

Overview of COM Technologies

[Topic groups](#) [See also](#)

Delphi provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM objects, Automation servers and clients (controllers), Active Server Pages, ActiveX controls, or ActiveForms.

COM is a language independent software component model designed by Microsoft to enable interaction between software components and applications. Microsoft extends this technology with ActiveX, which is a primarily used for Intranet development.

The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface to those features.

Applications can access COM components and their interfaces that exist on the same computer as the application, or that exist on another computer on the network using a mechanism called Distributed COM or DCOM. For more information on clients, servers, and interfaces see [Parts of a COM application](#).

The topics in this topic group provide a conceptual overview of the underlying technology on which Automation and ActiveX controls are built.

COM as a specification and implementation

COM is both a specification and an implementation. The COM specification defines object creation and communication between objects. According to this specification, COM objects can be written in different languages, run in different process spaces and on different platforms. As long as the objects adhere to the written specification, they can communicate. This allows you to integrate legacy code as a component with new components implemented in object-oriented languages.

The COM implementation is the COM library (including OLE32.dll and OLEAut32.dll), which provides a number of core services that support the written specification. The COM library contains a set of standard interfaces that define the core functionality of a COM object, and a small set of API functions designed for the purpose of creating and managing COM objects.

Note: Delphi's interface objects and language follow the COM specification. Delphi's implementation of the COM specification is called the Delphi ActiveX framework (DAX). Most of the implementation is found in the AxCtrls unit.

When you use Delphi wizards and VCL objects in your application, you are using Delphi's implementation of the COM specification. In addition, Delphi provides some wrappers for COM services for those features that it does not implement directly (such as Active Documents). You can find these wrappers defined in the ComObj unit and the API definitions are found in the AxCtrls unit.

COM extensions

As COM has evolved, it has been extended beyond the basic COM services. COM serves as the basis for other technologies such as Automation, ActiveX controls, Active Server Pages, and Active Documents. For details, see [COM extensions](#).

In addition, you can create a COM object that can work within the Microsoft Transaction Server (MTS) environment. MTS is a component-based transaction processing system for building, deploying, and managing large intranet and Internet server applications. Even though MTS is not architecturally part of COM, it is designed to extend the capabilities of COM in a large, distributed environment. For more information on MTS, see [Creating MTS objects](#).

Delphi provides wizards to easily implement applications that incorporate the above technologies in the Delphi environment. For details, see [Implementing COM objects with wizards](#).

Parts of a COM application

[Topic groups](#)

When implementing a COM application, you supply the following:

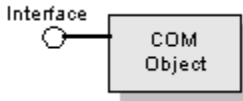
- COM Interface** The way in which an object exposes its services externally to clients. A COM object provides an interface for each set of related methods (member functions) and properties (data members and or content).
- COM server** A module, either an EXE, DLL, or OCX, that contains the code for a COM object. Object implementations reside in servers. A COM object implements one or more interfaces.
- COM client** The code that calls the interfaces to get the requested services from the server. Clients know what they want to get from the server (through the interface); clients do not know the internals of how the server provides the services. The most common COM client to implement is an **Automation controller**. Delphi eases the process in creating a client by allowing you to install COM servers (such as a Word document or Powerpoint slide) as components on the Component Palette. This allows you to connect to the server and hook its events through the Object Inspector.

COM Interfaces

[Topic groups](#) [See also](#)

COM clients communicate with objects through COM interfaces. Interfaces are groups of logically or semantically related routines which provide communication between a provider of a service (server object) and its clients. The standard way to depict a COM interface is as follows:

A COM interface



For example, every COM object implements the basic interface, *IUnknown*, which tells the client what interfaces are available on the client.

Objects can have multiple interfaces, where each interface implements a feature. An interface provides a way to convey to the client what service it provides, without providing implementation details of how or where the object provides this service.

Key aspects of COM interfaces are as follows:

- Once published, interfaces are immutable; that is, they do not change. You can rely on an interface to provide a specific set of functions. Additional functionality is provided by additional interfaces.
- By convention, COM interface identifiers begin with a capital I and a symbolic name that defines the interface, such as *IMalloc* or *IPersist*.
- Interfaces are guaranteed to have a unique identification, called a **Globally Unique Identifier (GUID)**, which is a 128-bit randomly generated number. Interface GUIDs are called **Interface Identifiers (IIDs)**. This eliminates naming conflicts between different versions of a product or different products.
- Interfaces are language independent. You can use any language to implement a COM interface as long as the language supports a structure of pointers, and can call a function through a pointer either explicitly or implicitly.
- Interfaces are not objects themselves; they provide a way to access an object. Therefore, clients do not access data directly; clients access data through an interface pointer.
- Interfaces are always inherited from the fundamental interface, *IUnknown*.
- Interfaces can be redirected by COM through proxies to enable interface method calls to call between threads, processes, and networked machines, all without the client or server objects ever being aware of the redirection. For more information, see in-process, out-of-process, and remote servers.

The fundamental COM interface, IUnknown

[Topic groups](#) [See also](#)

All COM objects must support the fundamental interface, called *IUnknown*, which contains the following routines:

<i>QueryInterface</i>	Provides pointers to other interfaces that the object supports.
<i>AddRef</i> and <i>Release</i>	Simple reference counting methods that keep track of the object's lifetime so that an object can delete itself when the client no longer needs its service.

Clients obtain pointers to other interfaces through the *IUnknown* method, *QueryInterface*. *QueryInterface* knows about every interface in the server object and can give a client a pointer to the requested interface. When receiving a pointer to an interface, the client is assured that it can call any method of the interface.

Objects track their own lifetime through the *IUnknown* methods, *AddRef* and *Release*, which are simple reference counting methods. As long as an object's reference count is nonzero, the object remains in memory. Once the reference count reaches zero, the interface implementation can safely dispose of the underlying object(s).

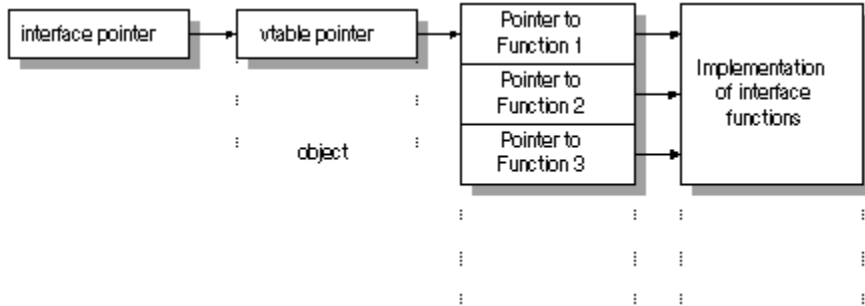
COM interface pointers

[Topic groups](#) [See also](#)

An interface pointer is a 32-bit pointer to an object instance that points, in turn, to the implementation of each method in the interface. The implementation is accessed through an array of pointers to these methods, which is called a **vtable**. Vtables are similar to the mechanism used to support virtual functions in ObjectPascal.

The vtable is shared among all instances of an object class, so for each object instance, the object code allocates a second structure that contains its private data. The client's interface pointer, then, is a pointer to *the pointer* to the vtable as shown in the following diagram.

Interface vtable



COM servers

[Topic groups](#) [See also](#)

A COM server is an application or a library that provides services to a client application or library. A COM server consists of one or more COM objects, where a COM object is a set of properties (data members or content) and methods (or member functions).

Clients do not know *how* a COM object performs its service; the object's implementation remains encapsulated. An object makes its services available through its **interfaces**.

In addition, clients do not need to know *where* a COM object resides. COM provides transparent access regardless of the object's **location**.

When a client requests a service from a COM object, the client passes a class identifier (CLSID) to COM. A CLSID is simply a GUID that references a COM object. COM uses this CLSID to locate the appropriate server implementation, brings the code into memory, and has the server instantiate an object instance for the client. Thus, a COM server must provide a class factory object (*IClassFactory*) that creates instances of objects on demand. (The CLSID is based on the interface's GUID.)

In general, a COM server must perform the following:

- Register entries in the system registry that associate the server module with the class identifier (CLSID).
- Implement a class factory object, which is a special type of object that manufactures another object of a particular CLSID.
- Expose the class factory to COM.
- Provide an unloading mechanism through which a server that is not servicing clients can be removed from memory.

Note: Delphi wizards automate the creation of COM objects and servers; for details see [Implementing COM objects with wizards](#).

Class factories

[See also](#)

A COM object is an instance of a **CoClass**, which is a class that implements one or more COM interfaces. The COM object provides the services as defined by its CoClass interfaces.

CoClasses are instantiated by a special type of object called a *class factory*. Whenever an object's services are requested by a client, a class factory creates and registers an object instance for that particular client. If another client requests the object's services, the class factory creates another object instance to service the second client.

A CoClass must have a class factory and a class identifier (CLSID) so that its COM object can be instantiated externally, that is, from another module. Using these unique identifiers for CoClasses means that they can be updated whenever new interfaces are implemented in their class. A new interface can modify or add methods without affecting older versions, which is a common problem when using DLLs.

Delphi wizards take care of implementing and instantiating class factories.

In-process, out-of-process, and remote servers

See also

With COM, a client does not need to know where an object resides, it simply makes a call to an object's interface. COM performs the necessary steps to make the call. These steps differ depending on whether the object resides in the same process as the client, in a different process on the client machine, or in a different machine across the network. The different types of servers are known as:

- In-process server** A library (DLL) running in the *same process space* as the client, for example, an ActiveX control embedded in a Web page viewed under Internet Explorer or Netscape. Here, the ActiveX control is downloaded to the client machine and invoked within the same process as the Web browser.

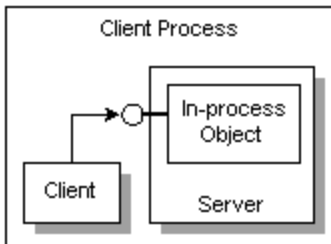
The client communicates with the in-process server using direct calls to the COM interface.
- Out-of-process server (or local server)** Another application (EXE) running in a *different process space* but on the *same machine* as the client. For example, an Excel spreadsheet embedded in a Word document are two separate applications running on the same machine.

The local server uses COM to communicate with the client.
- Remote server** A DLL or another application running on a *different machine* from that of the client. For example, a Delphi database application is connected to an application server on another machine in the network.

The remote server uses distributed COM (DCOM) interfaces to communicate with the application server.

As shown in the following figure, for in-process servers, pointers to the object interfaces are in the same process space as the client, so COM makes direct calls into the object implementation.

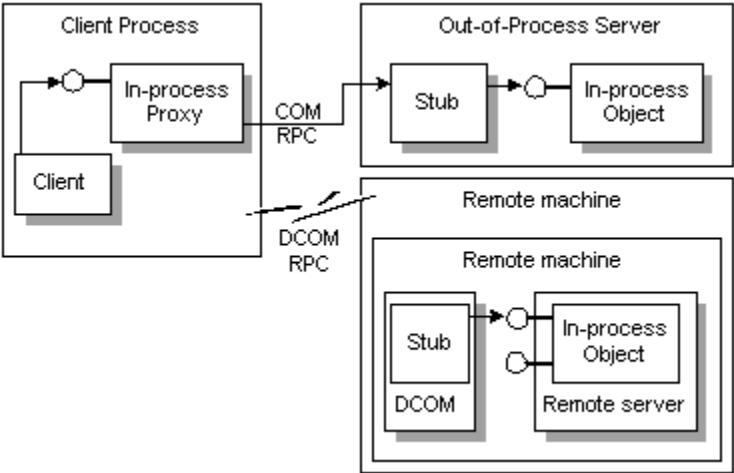
In-process servers



As shown in the following figure, when the process is either in a different process or in a different machine altogether, COM uses a proxy to initiate remote procedure calls. The **proxy** resides in the same process as the client, so from the client's perspective, all interface calls look alike. The proxy intercepts the client's call and forwards it to where the real object is running. The mechanism that enables the client to access objects in a different process space, or even different machine, as if they were in their own process, is called **marshaling**.

The difference between out-of-process and remote servers is the type of interprocess communication used. The proxy uses COM to communicate with an out-of-process server, it uses distributed COM (DCOM) to communicate with a remote machine.

Out-of-process and remote servers



The marshaling mechanism

[See also](#)

Marshaling is the mechanism that allows a client to make interface function calls to remote objects in another process or on a different machine. Marshaling

- Takes an interface pointer in the server's process and makes a proxy pointer available to code in the client process.
- Transfers the arguments of an interface call as passed from the client and places the arguments into the remote object's process space.

For any interface call, the client pushes arguments onto a stack and makes a function call through the interface pointer. If the call to the object is not in-process, the call gets passed to the proxy. The proxy packs the arguments into a marshaling packet and transmits the structure to the remote object. The object's stub unpacks the packet, pushes the arguments onto the stack, and calls the object's implementation. In essence, the object recreates the client's call in its own address space.

What type of marshaling occurs depends on what the COM object implements. Objects can use a standard marshaling mechanism provided by the *IDispatch* interface. This is a generic marshaling mechanism that enables communication through a system-standard remote procedure call (RPC). For details on the *IDispatch* interface, see [Automation Interfaces](#).

Note: [Microsoft Transaction Server \(MTS\)](#) provides additional support for remote objects.

COM Clients

[Topic groups](#) [See also](#)

It is important to design a COM application where clients can query the interfaces of an object to determine what an object is capable of providing. Server objects should have no expectations about the client using its objects. The client can determine what an object can provide through its interfaces. In addition, clients don't need to know how (or even where) an object provides the services; it just needs to rely on the object to provide the service it advertised through the interface.

A typical COM client is the Automation Controller. An Automation controller is the part of the application that has the broad perspective of the application's intended use. It knows the kinds of information it needs from various objects on the server, and it requests services when necessary.

Delphi makes it easier for you to develop an Automation Controller by allowing you to import an Automation server's type library and install it on the Component palette.

COM extensions

Topic groups

COM was originally designed to provide core communication functionality and to enable the broadening of this functionality through extensions. COM itself has extended its core functionality by defining specialized sets of interfaces for specific purposes.

ActiveX is a technology that allows COM components, especially controls, to be more compact and efficient. This is especially necessary for controls that are intended for Intranet applications that need to be downloaded by a client before they are used.

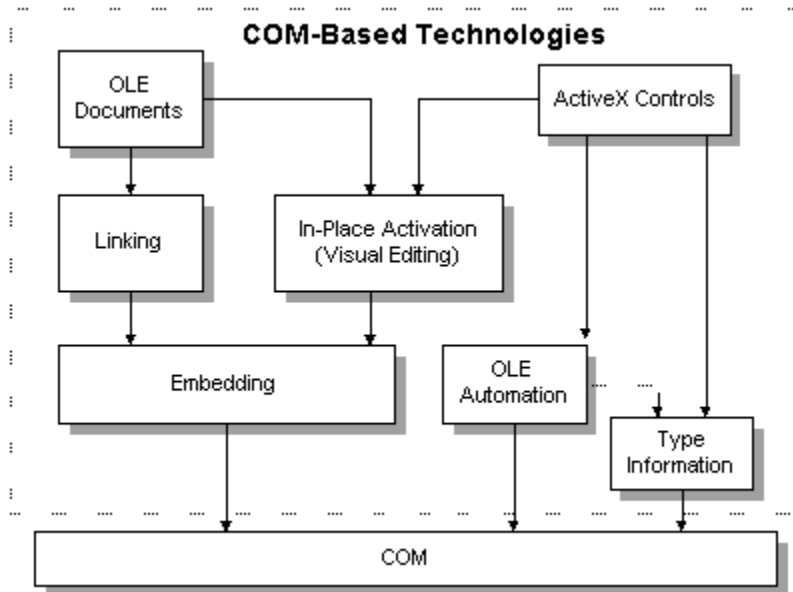
Microsoft is incorporating some of the MTS technologies for building complex Internet and intranet applications within the COM framework. This next evolution of COM, which also incorporates other new features, is currently called "COM+" (COM Plus) and should be available with the release of Windows 2000.

The following lists the vast array of services COM extensions currently provide.

<u>Automation servers</u>	Automation refers to the ability of an application to control the objects in another application programmatically. Automation servers are the objects that can be controlled by other executables at runtime.
<u>Automation controllers (or COM clients)</u>	Clients of Automation servers. Controllers provide a programming environment in which a developer or user can write scripts (controls) to drive Automation servers.
<u>ActiveX controls</u>	ActiveX controls are specialized in-process COM servers, typically intended for embedding in a client application. The controls offer both design and runtime behaviors as well as events.
<u>Type libraries</u>	A collection of static data structures, often saved as a resource, that provides detailed type information about an object and its interfaces. Clients of Automation servers and ActiveX controls expect type information to be available.
<u>Active Server Pages</u>	Active Server Pages are ActiveX components that allow you to create dynamic Web pages.
<u>Active Documents</u>	Objects that support linking and embedding, drag-and-drop, visual editing, and in-place activation. Word documents and Excel spreadsheets are examples of Active Documents.
<u>Visual cross-process objects</u>	Objects that can be manipulated across different processes.

The following diagram illustrates the relationship of the COM extensions and how they are built upon COM:

COM-based technologies



Using COM objects introduces both features and restrictions. These objects can be visual or non-visual. Some must run in the same process space as their clients; others can run in different processes or remote machines, as long as the objects provide marshaling support.

The following table summarizes the types of COM objects that you can create, whether they are visual, process spaces they can run in, how they provide marshaling, and whether they require a type library.

COM object requirements

<u>Object</u>	<u>Visual Object?</u>	<u>Process space</u>	<u>Communication</u>	<u>Type library</u>
Active Document	Usually	In-process, or out-of-process	OLE Verbs	No
Automation	Occasionally	In-process, out-of-process, or remote	Automatically marshaled using the <i>IDispatch</i> interface (for out-of-process and remote servers)	Required for automatic marshaling
ActiveX Control	Usually	In-process	Automatically marshaled using the <i>IDispatch</i> interface	Required
Custom interface object	Optionally	In-process	No marshaling required for in-process servers	Recommended
Custom interface object	Optionally	In-process, out-of-process, or remote	Automatically marshaled via a type library; otherwise, manually marshaled using custom interfaces	Recommended

Automation

[Topic groups](#) [See also](#)

Automation refers to the ability of an application to control the objects in another application programmatically, like a macro that can manipulate more than one application at the same time. The client of an Automation object is referred to as an Automation controller, and the server object being manipulated is called the Automation object.

Automation can be used on in-process, local, and remote servers.

Automation is characterized by two key points:

- The Automation object must be able to define a set of properties and commands, and to describe their capabilities through type descriptions. In order to do this they must have a way to provide information about the object's interfaces, the interface methods, and those methods' arguments. Typically this information is available in [type libraries](#). **The Automation server can also generate type information dynamically when queried.**
- Automation objects must make these methods accessible so that other applications can use them. For this, they must implement the *IDispatch* interface. Through this interface an object can expose all of its methods and properties. Through the primary method of this interface, the object's methods can be invoked, once having been identified through type information.

Developers often use Automation to create and use non-visual OLE objects that run in any process space because the Automation *IDispatch* interface automates the marshaling process. Automation does, however, restrict the types that you can use.

For a list of types that are valid for type libraries in general, and Automation interfaces in particular, see [Valid types](#).

ActiveX controls

[Topic groups](#)

ActiveX controls are visual controls that run only in in-process servers, and can be plugged into an ActiveX control container application. They are not complete applications in themselves, but can be thought of as prefabricated OLE controls that are reusable in various applications. ActiveX controls make use of Automation to expose their properties, methods, and events. Features of ActiveX controls include the ability to fire events, bind to data sources, and support licensing.

An increasingly common use of ActiveX controls is on a Web site as interactive objects in a Web page. As such, ActiveX has become a standard that has especially targeted interactive content for the World Wide Web, including the use of ActiveX Documents used for viewing non-HTML documents through a Web browser. For more information about ActiveX technology, see the Microsoft ActiveX Web site.

Delphi wizards allow you to easily [create ActiveX controls](#).

Type libraries

[Topic groups](#) [See also](#)

Type libraries provide a way to get more type information about an object than can be determined from an object's interface. The type information contained in type libraries provides needed information about objects and their interfaces, such as what interfaces exist on what objects (given the CLSID), what member functions exist on each interface, and what arguments those functions require.

You can obtain type information either by querying a running instance of an object or by loading and reading type libraries. With this information, you can implement a client which uses a desired object, knowing specifically what member functions you need, and what to pass those member functions.

Clients of Automation servers and ActiveX controls expect type information to be available. Automation and ActiveX wizards generate a type library automatically. You can view or edit this type information by using the **Type Library Editor**.

This topic covers:

- [Contents of type libraries](#)
- [Creating type libraries](#)
- [When to use type libraries](#)
- [Accessing type libraries](#)
- [Benefits of using type libraries](#)
- [Using type library tools](#)

The content of type libraries

Type libraries contain *type information*, which indicates which interfaces exist in which COM objects, and the types and numbers of arguments to the interface methods. These descriptions include the unique identifiers for the CoClasses (CLSIDs) and the interfaces (IIDs), so that they can be properly accessed, as well as the dispatch identifiers (dispIDs) for Automation interface methods and properties.

Type libraries can also contain the following information:

- Descriptions of custom type information associated with custom interfaces
- Routines that are exported by the Automation or ActiveX server, but that are not interface methods
- Information about enumeration, record (structures), unions, alias, and module data types
- References to type descriptions from other type libraries

Creating type libraries

With traditional development tools, you create type libraries by writing scripts in the Interface Definition Language (IDL) or the Object Description Language (ODL), then running that script through a compiler. However, Delphi automatically generates a type library when for you when you use either the Automation server or ActiveX control wizard. (You can also create a type library by choosing from the main menu, File|New|ActiveX|Type Library.) Then you can view the type library using Delphi's Type Library editor. You can easily edit your type library using the Type Library editor and Delphi automatically updates the appropriate source files when the type library is saved.

The [Type Library editor](#) automatically generates a standard type library, typically as a resource, along with a Delphi Interface File (.PAS file) that contains the type declarations in Object Pascal syntax.

When to use type libraries

It is important to create a type library for each set of objects that is exposed to external users, for example,

- ActiveX controls require a type library, which must be included as a resource in the DLL that contains the ActiveX controls.
- Exposed objects that support vtable binding of custom interfaces must be described in a type

library because vtable references are bound at compile time. For more information about vtable and compile time binding, see [Automation interfaces](#).

- Applications that implement Automation servers should provide a type library so that clients can early bind to it.
- Objects instantiated from classes that support the *IProvideClassInfo* interface, such as all descendants of the VCL *TTypedComObject* class, must have a type library.
- Type libraries are not required, but are useful for identifying the objects used with OLE drag-and-drop.

When defining interfaces for internal use only (within an application) you do not need to create a type library. For more information on how to define an interface directly in Object Pascal, see [Interface types](#).

Accessing type libraries

The binary type library is normally a part of a resource file (.RES) or a stand-alone file with a .TLB file-name extension. Once a type library has been created, object browsers, compilers, and similar tools can access type libraries through special type interfaces:

Interface	Description
<i>ITypeLib</i>	Provides methods for accessing a library of type descriptions.
<i>ITypeInfo</i>	Provides descriptions of individual objects contained in a type library. For example, a browser uses this interface to extract information about objects from the type library.
<i>ITypeComp</i>	Provides a fast way to access information that compilers need when binding to an interface.

Delphi can import and use type libraries from other applications. Most of the VCL classes used for COM applications support the essential interfaces that are used to store and retrieve type information from type libraries and from running instances of an object. The VCL class *TTypedComObject* supports interfaces that provide type information, and is used as a foundation for the ActiveX object framework.

Benefits of using type libraries

Even if your application does not require a type library, you can consider the following benefits of using one:

- Type checking can be performed at compile time.
- You can use early binding with Automation (as opposed to calling through Variants), and controllers that do not support vtables or dual interfaces can encode dispIDs at compile time, improving runtime performance.
- Type browsers can scan the library, so clients can see the characteristics of your objects.
- The *RegisterTypeLib* function can be used to register your exposed objects in the registration database.
- The *UnRegisterTypeLib* function can be used to completely uninstall an application's type library from the system registry.
- Local server access is improved because Automation uses information from the type library to package the parameters that are passed to an object in another process.

Using type library tools

The tools for working with type libraries are listed below.

- The TLIBIMP (Type Library Import) tool, which takes existing type libraries and creates Delphi Interface files, is incorporated into the Type Library editor. TLIBIMP provides additional configuration options not available inside the Type Library editor.
- The Microsoft IDL compiler (MIDL) compiles IDL scripts to create a type library.
- MKTYPLIB is an ODL compiler that compiles ODL scripts to create a type library, found in the MS

Win32 SDK.

- OLEView is a type library browser tool, found on Microsoft's Web site.
- TRegSvr is a tool for registering and unregistering servers and type libraries, which comes with Delphi. The source to TRegSvr is available as an example in the Examples directory.
- RegSvr32.exe is a tool for registering and unregistering servers and type libraries, which is a standard Windows utility.

Active Server Pages

[Topic groups](#) [See also](#)

The Active Server Pages (ASP) technology allows you to build Web pages dynamically by using ActiveX server components. With Active Server Pages, you can embed ActiveX controls in a Web page that get called every time the server loads the Web page. For example, you can write an Object Pascal program, such as one to create a bitmap or connect to a database, and this control accesses data that gets updated every time the server loads the Web page.

Active Server Pages relies on the Microsoft Internet Information Server (IIS) environment to serve your Web pages.

On the server side, the Active Server Pages are ActiveX components which you can develop using Delphi or most other languages including C++, Java, or Visual Basic. On the client side, the ASP is a standard HTML document and can be viewed by users on any platform using any Web browser.

Delphi wizards allow you to easily [create Active Server Pages](#).

Active Documents

[Topic groups](#)

Active Documents (previously referred to as OLE documents) are a set of COM services that supports linking and embedding, drag-and-drop, and visual editing. Active Documents can seamlessly incorporate data or objects of different formats, for example, sound clips, spreadsheets, text, and bitmaps.

Unlike ActiveX controls, Active Documents are not limited to in-process servers; they can be used in cross-process applications.

Unlike Automation objects, which are almost never visual, Active Document objects can be visually active in another application. So, Active Document objects are associated with two types of data: presentation data, used for visually displaying the object on a display or output device, and native data, used to edit an object.

Active Document objects can be document containers or document servers. While Delphi does not provide an automatic wizard for creating Active Documents, you can use the VCL class, *[TOleContainer](#)*, to support linking and embedding in existing Active Documents.

You can also use *TOleContainer* as a basis for an Active Document container. To create objects for Active Document servers, use one of the VCL COM base classes and implement the appropriate interfaces for that object type, depending on the services the object needs to support. For more information about creating and using Active Document servers, see the Microsoft ActiveX Web site.

Note: While the specification for Active Documents has built-in support for marshaling for cross-process applications, Active Documents do not run on remote servers because they use types that are specific to a system on a given machine, for example, window handles, menu handles, and so on.

Visual cross-process objects

[Topic groups](#) [See also](#)

Automation objects, Active Documents and ActiveX controls are commonly used objects. Less common are OLE/ActiveX objects that are visually displayed and manipulated in a cross-process application. These types of objects are more difficult to create because the communication protocol involved in visually manipulating objects in cross-process applications is only standardized for visual objects that use Active Document interfaces. Therefore, you need to write one or more custom interfaces that your object implements, and then handle the marshaling interfaces yourself.

This can be done by either

- Using a dual interface *IDispatch*, which provides automatic marshaling. This is the recommended way. It is the easiest approach, and the Automation wizard creates dual interfaces by default when you create an Automation object.
- Writing the marshaling classes by hand by implementing *IMarshal* or related interfaces.

Implementing COM objects with wizards

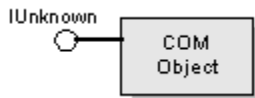
Topic groups

Delphi makes it easier to write COM applications by providing wizards to help create Delphi applications that run in the COM environment. It provides separate wizards to create the following:

- A simple COM object
- An Automation object
- An ActiveX control
- An Active server page
- An ActiveX Form
- ActiveX library
- Property page
- Type library
- MTS object

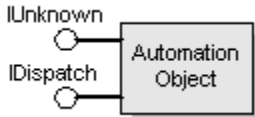
The wizards automate the tasks involved in creating each type of COM object. They provide the required COM interfaces for each type of object. With a simple COM object, the wizard implements the one required COM interface, *IUnknown*, which provides an interface pointer to the object.

Simple COM object interface



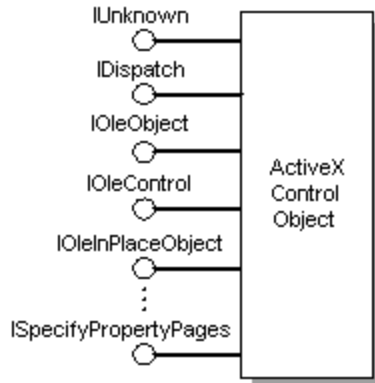
For Automation objects, the wizard implements *IUnknown* and *IDispatch*, which provides automatic marshaling.

Automation object interface



For ActiveX control objects, the wizard implements all the required ActiveX control interfaces, from *IUnknown*, *IDispatch*, *IObject*, *IObjectControl*, and so on. For a complete list of interfaces, see the reference page for [TActiveXControl](#) object.

ActiveX control interface



You can think of the wizards as providing an increasing degree of implementation. The various wizards implement the following COM interfaces:

Wizard	Implemented interfaces	What the wizard does
COM server	<i>IUnknown</i>	Exports the necessary routines that handle server registration, class registration, loading and unloading the server, and object instantiation. Creates and manages the class factories for objects implemented on the server.

		Instructs COM to invoke object interfaces based on a specified threading model. Provides a type library, if requested.
Automation server	<i>IUnknown, IDispatch</i>	Performs the tasks of a COM server wizard (described above), plus: Implements the interface that you specify, either dual or dispatch. Provides a type library automatically.
ActiveX Control	<i>IUnknown, IDispatch, IPersistStreamInit, IOleInPlaceActiveObject, IPersistStorage, IViewObject, IOleObject, IViewObject2, IOleControl, IPerPropertyBrowsing, IOleInPlaceObject, ISpecifyPropertyPages</i>	Performs the tasks of the COM server and Automation server wizards (described above), plus: Implements the properties, methods, and events for all the interfaces in the TActiveXControl. Leaves you in the source code editor so that you can modify the object.
ActiveForm	Same interfaces as ActiveX Control	Performs the tasks of the ActiveX control wizard, plus: Implements the properties, methods, and events for all the interfaces in the TActiveXControl. Leaves you with a form so that you can design an application.
Active Server Object	<i>IUnknown, IDispatch</i>	Performs the tasks of an Automation object wizard (described above) and generates an .ASP page which can be loaded into a Web browser. It leaves you in the Type Library editor so that you can modify the object's properties and methods if needed. If you set the Active Server Type to Page-Level event methods, it implements <i>OnStartPage</i> and <i>OnEndPage</i> automatically for you.
ActiveX library	None, by default	Creates a new ActiveX or Com server DLL and exposes the necessary export functions.
Property Page	<i>IUnknown, IPropertyPage</i>	Creates a new property page that you can design in the Forms designer.
Type Library	None, by default	Creates a new type library and associates it with the active project.
MTS object	The <i>IObjectControl</i> interface methods, <i>Activate</i> , <i>Deactivate</i> , and <i>CanBePooled</i> .	Adds a new unit to the current project containing the <u>MTS</u> object definition, so that clients can access this server within the MTS runtime environment. It leaves you in the Type Library editor so that you can modify the object's properties and methods if needed.

If you want to add any additional COM objects (or reimplement any existing implementation), you can do

so. To provide a new interface, you would create a descendant of the *IDispatch* interface and implement the needed methods in that descendant. To reimplement an interface, create a descendant of that interface and modify the descendant.

For details in developing COM applications using these wizards, see:

[Creating a COM object](#)

[Creating an Automation controller](#)

[Creating an Automation server](#)

[Creating an ActiveX control](#)

[Creating Active Server Pages](#)

[Working with type libraries](#)

[Creating MTS Automation objects](#)

Creating a simple COM object

[Topic groups](#) [See also](#)

Delphi provides wizards to help you create various COM objects. This is an overview of how to create a simple, lightweight COM object, such as a shell extension, in the Delphi environment. To create Automation client and server objects, see [Creating an Automation Controller](#), and [Creating an Automation server](#). To create an ActiveX control, see [Creating an ActiveX control](#). To create an Active Server Page, see [Creating Active Server Pages](#).

This topic is not intended to provide complete details of writing COM applications. For that information, refer to your Microsoft Developer's Network (MSDN) documentation. The Microsoft Web site also provides current information on this topic.

Overview of creating a COM object

Use the COM object wizard to create a simple, lightweight COM object, for example, a shell extension. A COM object can be implemented as either an in-process server, out-of-process server, or remote server.

The COM object wizard performs the following tasks:

- Creates a new unit.
- Defines a new class that descends from *TComObject* and sets up the class factory constructor.

The process of creating a COM object involves these steps:

- 1 [Design](#) the COM object.
- 2 [Use the COM object wizard](#) to create a COM object.
- 3 [Register an ActiveX control](#).
- 4 Test the COM object.

Designing a COM object

[Topic groups](#)

When designing the COM object, you need to decide what COM interfaces you want to implement. The wizard supplies the *IUnknown* interface. If you want to implement any other COM interfaces, see the MSDN documentation.

You must decide whether the COM object is an in-process server, out-of-process server, or remote server. For in-process servers and for out-of-process and remote servers using a type library, COM marshals the data for you. Otherwise, you must consider how to marshal the data to out-of-process servers.

For information, see [In-process, out-of-process, and remote servers](#).

Creating a COM object

[Topic groups](#) [See also](#)

Before you create a COM object, create or open the project for an application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on your needs.

To bring up the COM object wizard,

- 1 Choose File|New to open the New Items dialog box.
- 2 Select the tab labeled, ActiveX.
- 3 Double-click the COM object icon.

In the wizard, specify the following:

ClassName	Specify the name for the object that you want to implement.
Instancing	Specify an <u>instancing</u> mode to indicate how your COM object is launched. Note: When your COM object is used only as an in-process server, instancing is ignored.
Threading model	Choose the <u>threading model</u> to indicate how client applications can call your COM object's interface. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.
Implemented interfaces	Specify the names of the COM interfaces that you want this COM object to implement.
Description	Enter a description of the COM object you are creating.
Include Type Library	Check this box to generate a type library for this object. A type library contains type information that allows you to expose any object interface and its methods and properties to client applications. Checking this box automatically causes Mark Interface OleAutomation to be checked.
Mark interface OleAutomation	Check this box to enable the <u>marshaling code</u> that gets generated when you create a type library. COM knows how to marshal all the Automation-compatible types in the type library and can set up the proxies and stubs for you so that you can pass parameters to out-of-process (.EXE) servers.

COM object instancing types

[Topic groups](#)

Note: Instancing is ignored when your COM object is used only as an in-process server.

When your COM application creates a new COM object, it can have any of the following instancing types:

<u>Instancing</u>	<u>Meaning</u>
Internal	The object can only be created internally. An external application cannot create an instance of the object directly. For example, a word processor application may have a document object that can only be created by calling a method of the application that can create the document object.
Single Instance	Allows only a single COM interface for each executable (application), so creating multiple instances results in creating multiple applications. Single instance specifies that once an application has connected to the object, it is removed from public view so that no other applications can connect to it. This option is commonly used for multiple document interface (MDI) applications. When a client requests services from a single instance object, all requests are handled by the same server. For example, any time a user requests to open a new document in a word processor application, typically the new document opens in the same application process.
Multiple Instance	Specifies that multiple applications can connect to the object. Any time a client requests service, a separate instance of the server gets invoked. (That is, there can be multiple instances in a single executable.) Any time a user attempts to open the Windows Explorer, a separate Explorer is created.

Choosing a threading model

[Topic groups](#) [See also](#)

When creating an object from the wizard, you select a threading model that your object agrees to support. By adding thread support to your COM object, you can improve its performance.

For more details, see

[Writing an object that supports the free threading model](#)

[Writing an object that supports the apartment threading model](#)

Threading models for COM objects

<u>Threading model</u>	<u>Description</u>	<u>Implementation pros and cons</u>
Single	No thread support. Client requests are serialized by the calling mechanism.	Clients are handled one at a time so no threading support is needed. No performance benefit.
Apartment (or Single-threaded apartment)	Clients can call an object's methods only from the thread on which the object was created. Different objects from the same server can be called on different threads, but each object is called only from that one thread.	Instance data is safe, global data must be protected using critical sections or some other form of serialization. The thread's local variables are reliable across multiple calls. Some performance benefits. Objects are easy to write, but clients can be tricky. Primarily used for controls for Web browsers.
Free (also called multi-threaded apartment)	Clients can call any object's methods from any thread at any time. Objects can handle any number of threads at any time.	Objects must protect all instance and global data using critical sections or some other form of serialization. Thread local variables are <i>not</i> reliable across multiple calls. Clients are easy to write, but objects are harder to write. Primarily used for distributed DCOM environments.
Both	Objects can support clients that use either apartment or free threading models.	Maximum performance and flexibility. Supports both threading models when clients may be either using single-threaded or free for improved performance.

Both the client and server sides of the application tell COM the rules it intends to follow for using threads when it initializes COM. COM compares the threading rules that each party has agreed to support. If both parties support the same rules, COM sets up a direct connection between the two and trusts that the parties follow the rules. If the two parties advertise different rules, COM sets up marshaling between the two parties so that each sees only the threading rules that it says it knows how to handle. Of course, marshaling affects performance somewhat, but it allows parties with different threading models to work together.

Note: The threading model you choose in the wizard determines how the object is advertised in the registry. You must make sure that your object implementation adheres to the threading model you have chosen.

The threading model is valid for in-process servers only. Setting the threading model in the wizard sets the threading model key in the CLSID registry entry, InProcessServer32.

Out-of-process servers are registered as EXE and it is up to you to implement the threading model yourself. If the EXE contains a free threaded server, Delphi will initialize COM for free threading, which

means that it can provide the expected support for any free-threaded or apartment-threaded objects contained in the EXE. To manually override threading behavior in EXEs, see [ColnitFlags](#).

Note: Local variables are always safe, regardless of the threading model. This is because local variables are stored on the stack and each thread has its own stack.

Writing an object that supports the free threading model

Use the free threading model rather than apartment threading whenever the object needs to be accessed from more than one thread. A common example is a client application connected to an object on a remote machine. When the remote client calls a method on that object, the server receives the call on a thread from the thread pool on the server machine. This receiving thread makes the call locally to the actual object; and, because the object supports the free threading model, the thread can make a direct call into the object.

If the object supported the apartment threading model instead, the call would have to be transferred to the thread on which the object was created, and the result would have to be transferred back into the receiving thread before returning to the client. This approach requires extra marshaling.

To support free threading, you must consider how instance data can be accessed for *each* method. If the method is writing to instance data, you must use critical sections or some other form of serialization, to protect the instance data. Likely, the overhead of serializing critical calls is less than executing COM's marshaling code.

Note that if the instance data is read-only, serialization is not needed.

Writing an object that supports the apartment threading model

To implement the (single-threaded) apartment threading model, you must follow a few rules;

- The first thread in the application that gets created is COM's main thread. This is typically the thread on which WinMain was called. This must also be the last thread to uninitialized COM.
- Each thread in the apartment threading model must have a message loop, and the message queue must be checked frequently.
- When a thread gets a pointer to a COM interface, that interface's methods may only be called from that thread.

The single-threaded apartment model is the middle ground between providing no threading support and full, multi-threading support of the free threading model. A server committing to the apartment model promises that the server has serialized access to all of its global data (such as its object count). This is because different objects may try to access the global data from different threads. However, the object's instance data is safe because the methods are always called on the same thread.

Typically, controls for use in Web browsers use the apartment threading model because browser applications always initialize their threads as apartment.

For general information on threads, see [Writing multi-threaded applications](#).

Registering a COM object

[Topic groups](#)

After you have created your COM object, you must register it so that other applications can find and use it.

Note: Before you remove a COM object from your system, you should unregister it.

To register a COM object,

- Choose Run|Register ActiveX Server.

To unregister a COM object,

- Choose Run|Unregister ActiveX Server.

As an alternative, you can use the **regsvr** command from the command line or run the regsvr32.exe from the operating system.

Testing a COM object

[Topic groups](#) [See also](#)

Testing your COM object depends on the COM object you have designed. Once you have created the COM object, test it by using the interfaces you implemented to access the methods of the interfaces.

To test and debug a COM object,

- 1 Turn on debugging information using the Compiler tab on the Project|Options dialog box, if necessary. Also, turn on Integrated Debugging in the Tools|Debugger Options dialog.
- 2 For an in-process server, choose Run|Parameters.
- 3 In the Host Application box, type the name of the client application which will be requesting the service of this COM object, and choose OK.
- 4 Choose Run|Run.
- 5 Set breakpoints in the COM object.
- 6 Use the client application to interact with the COM object.

The COM object pauses when the breakpoints are reached.

Creating an Automation controller

[Topic groups](#) [See also](#)

Automation is a COM protocol that defines how one application accesses an object that resides within another application or DLL. An *Automation controller* is a client application that controls an Automation server through one or more server-provided objects which implement the *IDispatch* interface.

Automation controllers can be written in any language for which an implementation of COM and Automation is provided. Most automation controllers are currently written in C++, Object Pascal (Delphi), or Visual Basic.

Examples of Automation servers include Microsoft Word, Microsoft Excel, and Internet Explorer. These applications can be controlled by Delphi applications or by other Automation controllers.

Delphi gives you the flexibility to integrate applications and DLLs with a variety of applications as either Automation servers or as controllers.

This series of topics describes how to create an Automation controller in the Delphi environment. It is not intended to provide complete details of controller application development for every type of server. For specific information on a server application, you should consult that application's documentation.

In Delphi, you create an Automation controller by importing an Automation server's type library and installing it on the Component palette.

Creating an Automation controller by importing a type library

[Topic groups](#) [See also](#)

You can create an Automation controller by importing an Automation server's type library and using the automatically generated classes to control the Automation server. With the Import Type Library dialog, you install the server that the type library describes to the Component palette, which allows you to connect to the server and hook up its events using the Object Inspector. You can then manipulate the server properties programmatically.

To import an Automation controller's type library,

- 1 Choose Project|Import Type Library.
- 2 Select the type library from the list.

The dialog lists all the libraries registered on this system. If the type library is not in the list, choose the Add button, find and select the type library file, choose OK, and repeat step 2. Note that the type library could be a standalone type library file (.TLB, .OLB), or a server that provides a type library (.DLL, .OCX, .EXE).

- 3 Choose the Palette page on which this server will reside.
- 4 Check Generate Component Wrapper, which creates a *TComponent* wrapper that allows you to install the server that the type library describes on the Component palette.
- 5 Choose Install.

Specify where you want the type library to reside, either in an existing package or new package. This button is grayed out if no component can be created for that type library.

A server that the type library described now resides on the Component palette. You can use the Object Inspector to write an event handler for the server.

To control the Automation server through this controller, you add code to the implementation unit to provide support for early (VTable) binding and/or late (dispatch) binding. The unit that you just created includes interface wrappers for both VTable binding for all exposed interfaces, and dispatch binding for dual and dispatch interfaces.

Handling events in an Automation Controller

[See also](#)

Once you have installed a server on the Component palette, you can use the Object Inspector to write an event handler.

To support events,

- 1 From the Component palette, drop the desired server component onto your form.
- 2 Select an object, and click the Events tab on the Object Inspector. You'll see a list of the object's events.
- 3 Double-click in the space to the right of an event and Delphi opens the Code editor, displaying the skeleton event handler for you to complete.

After implementing your event handler, you can [connect to a server](#).

Connecting to and disconnecting from a server

[See also](#)

Typically, you connect to a server through its main interface. For example, you would connect to Microsoft Word through the WordApplication component. Once you've connected to the main interface, you can connect to any of the application's components (such as a WordDocument or WordParagraphFormat) by using the *ConnectTo* method.

Here is the event handler for the ExcelApplication event, OnNewWorkBook. In it, we are assigning the workbook to the ExcelWorkbook component.

```
procedure TForm1.XLappNewWorkbook(Sender: TObject; var Wb:OleVariant);
begin
    ExcelWorkbook1.ConnectTo((iUnknown(Wb) as ExcelWorkBook));
end;
```

After importing the type library, add code to the implementation unit to control the server with either a dual interface (most typical) or a dispatch interface.

Note: The import files created when a type library is imported (*libname_TLB.CPP* and *libname_TLB.H*) should be considered to be read-only, and are not intended for modification of any kind. These files are regenerated each time the type library is refreshed, so any changes are overwritten.

For servers that expose a Quit method (such as WordApplication and ExcelApplication), code is generated to call this method in the Disconnect method. Quit typically exposes functionality that is equivalent to clicking on File to quit the application. If AutoConnect is set to True, the application server will Quit when the client exits. Therefore, hitting F1 when AutoQuit is highlighted in the Object Inspector does nothing.

Controlling an Automation server using a dual interface

[See also](#)

When you create an Automation controller by selecting an object from the Component Palette, it automatically provides a [dual interface](#) because Delphi can determine the VTable layout from information in the type library. Calling a method on the class automatically connects to an instance of Word.

For example, for a Word server, you call on a method of class TWordApplication as follows:

```
foo := TWordApplication      {New way}
foo.DoSomething;
```

Of course, the former way of controlling an Automation server with a dual interface still works, but it is more cumbersome. You must first, declare an interface and initialize it with the Create method of a CoClass client proxy class. Then you call methods of the smart interface object. For example,

```
foo : _Application
foo := CoWordApplication.Create      {Old way}
foo.DoSomething;
```

The interface and CoClass client proxy class are defined in the unit that is generated automatically when you import a type library. The names that start with "I" are smart interfaces. For example, the main smart interface for Microsoft Word is "IWordBasic". (A smart interface automatically frees its wrapped interface when it is destroyed.) The names of the CoClass client proxy classes start with "Co". For example, the main CoClass client proxy class for Microsoft Word is CoApplication.

Controlling an Automation server using a dispatch interface

[See also](#)

Typically, you will use the [dual interface](#) to control the Automation server. However, you may find a need to control an Automation server with a smart [dispatch interface](#) object. To do so,

- 1 In the Automation controller's implementation unit, declare a dispinterface.
- 2 Control the Automation server by calling methods of the dispatch interface object.

For information on dispatch interfaces, see "Automation interfaces" on page 8.

Example: Printing a document with Microsoft Word

[Topic groups](#) [See also](#)

The following steps show how to create an Automation controller that prints a document using Microsoft Word 8 from Office 97.

Before you begin: Create a new project that consists of a form, a button, and an open dialog box (*TOpenDialog*). These controls constitute the Automation controller.

Step 1: Prepare Delphi for this example

For your convenience, Delphi has provided many common servers, such as Word, Excel, and Powerpoint, on the Component Palette. To demonstrate how to import a server, we use Word. Since it already exists on the Component Palette, this first step asks you to remove the package containing Word so that you can install it on the palette. Step 4 describes how to return the Component Palette to its normal state.

To remove Word from the Component palette,

- 1 Choose Component|Install packages.
- 2 Click Borland Sample Automation Server components and choose Remove.

The Servers page of the Component Palette no longer contains any of the servers supplied with Delphi. (If no other servers have been imported, the Servers page also disappears.)

Step 2: Import the Word type library

To import the Word type library,

- 1 Choose Project|Import Type Library.
- 2 In the Import Type Library dialog,
 - 1 Select Microsoft Office 8.0 Object Library.

If Word (Version 8) is not in the list, choose the Add button, go to Program Files\Microsoft Office\Office, select the Word type library file, MSWord8.olb choose Add, and then select Word (Version 8) from the list.

- 2 In Palette Page, choose Servers.
- 3 Choose Install.

The Install dialog appears. Select the Into New Packages tab and type WordExample to create a new package containing this type library.

- 3 Go to the Servers Palette Page, select WordApplication and place it on a form.
- 4 Write an event handler for the button object as described in the next step.

Step 3: Use a VTable or dispatch interface object to control Microsoft Word

You can use either a VTable or a dispatch object to control Microsoft Word.

Using a VTable interface object

By dropping an instance of the WordApplication object onto your form, you can easily access the control using a VTable interface object. You simply call on methods of the class you just created. For Word, this is the TWordApplication class.

- 1 Select the button, double-click its *OnQuit* event handler and supply the following event handling code:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FileName: OleVariant;
begin
  if OpenDialog1.Execute then
  begin
    FileName := OpenDialog1.FileName;
    WordApplication1.Documents.Open(FileName,
      EmptyParam, EmptyParam, EmptyParam,
      EmptyParam, EmptyParam, EmptyParam,
```

```

    EmptyParam, EmptyParam, EmptyParam) ;
WordApplication1.ActiveDocument.PrintOut (
    EmptyParam, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam) ;
    end;
end;

```

- 2 Build and run the program. By clicking the button, Word prompts you for a file to print.

Using a dispatch interface object

As an alternate, you can use a dispatch interface for late binding. To use a dispatch interface object, you create and initialize the Application object using the `_ApplicationDisp` dispatch wrapper class as follows. Notice that dispinterface methods are "documented" by the source as returning Vtable interfaces, but, in fact, you must cast them to dispatch interfaces.

- 1 Select the button, double-click its *OnQuit* event handler and supply the following event handling code:

```

procedure TForm1.Button1Click(Sender: TObject);
var
MyWord : _ApplicationDisp;
FileName : OleVariant;
begin
    if OpenFileDialog1.Execute then
    begin
        FileName := OpenFileDialog1.FileName;
        MyWord := CoWordApplication.Create as
            _ApplicationDisp;
        (MyWord.Documents as
            DocumentsDisp).Open(FileName, EmptyParam,
            EmptyParam, EmptyParam, EmptyParam, EmptyParam,
            EmptyParam, EmptyParam, EmptyParam);
        (MyWord.ActiveDocument as
            _DocumentDisp).PrintOut(EmptyParam, EmptyParam,
            EmptyParam, EmptyParam, EmptyParam, EmptyParam,
            EmptyParam, EmptyParam, EmptyParam, EmptyParam,
            EmptyParam, EmptyParam, EmptyParam, EmptyParam);
        MyWord.Quit(EmptyParam, EmptyParam, EmptyParam);
    end;
end;

```

- 2 Build and run the program. By clicking the button, Word prompts you for a file to print.

Step 4: Clean-up the example

After completing this example, you will want to restore Delphi to its original form.

- 1 Delete the objects on this Servers page:
 - 1 Choose Component|Configure Palette and select the Servers Page.
 - 2 Select all objects on the page that you just added, choose Hide.
 - 3 The Servers page no longer appears.
- 2 Return the Borland Sample Automation Server Components package:
 - 1 Choose Component|Install Packages.
 - 2 From the list, select the WordExample package and click remove.
 - 3 From the list, select Borland Sample Automation Server Components package, and click Add.

The Servers tab is returned to the Component Palette.

Getting more information

For the most up-to-date information about the dispatch interface, dual interface, and CoClass client proxy classes, refer to the comments in the automatically-generated source file.

Creating an Automation server

[Topic groups](#) [See also](#)

An Automation server is an application that exposes its functionality for client applications, called Automation controllers, to use. Controllers can be any applications that support Automation, such as Delphi, Visual Basic, or C++Builder. An Automation server can be an application or library.

This topic shows how to create an Automation server using the Delphi Automation server wizard. With this, you can expose properties and methods of an existing application for Automation control.

Here are the steps for creating an Automation server from an existing application:

- [Create an Automation object](#) for the application.
- [Expose the application's properties and methods for Automation.](#)
- [Register](#) the application as an Automation server.
- [Test and debug](#) the application.

For information about creating an Automation controller, see [Creating an Automation Controller](#). For general information about the COM technologies, see [Overview of COM technologies](#).

Creating an Automation object for an application

[Topic groups](#) [See also](#)

An Automation object is an ObjectPascal class descending from *TAutoObject* that supports OLE Automation, exposing itself for other applications to use. Since *TAutoObject* supports OLE Automation, any object derived from it gets Automation support automatically. You create an Automation object using the Automation Object wizard.

Before you create an Automation object, create or open the project for an application containing functionality that you want to expose. The project can be either an application or ActiveX library, depending on your needs.

To display the Automation wizard:

- 1 Choose File|New.
- 2 Select the tab labeled, ActiveX.
- 3 Double-click the Automation Object icon.

In the wizard dialog, specify the following:

CoClass Name	Specify the class whose properties and methods you want to expose to client applications. (Delphi prepends a T to this name.)
Instancing	Specify an <u>instancing</u> mode to indicate how your Automation server is launched. Note: When your Automation object is used only as an in-process server, instancing is ignored.
Threading Model	Choose the <u>threading model</u> to indicate how client applications can call your object's interface. This is the threading model that you commit to implementing in the Automation object. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.
Generate event support code	Check this box to tell the wizard to implement a separate interface for <u>managing events</u> of your Automation object.

When you complete this procedure, a new unit is added to the current project that contains the definition for the Automation object. In addition, the wizard adds a type library project and opens the type library. Now you can expose the properties and methods of the interface through the type library as described in [Exposing the application's properties and methods for Automation](#).

The Automation object implements a **dual interface**, which supports both early (compile-time) binding through the *VTable* and late (runtime) binding through the *IDispatch* interface.

Managing events in your Automation object

[Topic groups](#) [See also](#)

The Automation wizard automatically generates event code if you check the option, Generate Support Code in the Automation Object wizard dialog box.

For a server to support events, it provides a definition of an outgoing interface which is implemented by a client. The client determines what outgoing interfaces are available by querying the server's *IConnectionPointContainer* interface, and it uses methods provided by the server's *IConnectionPoint* interface to pass the server a pointer to the client's implementation of the events (known as a sink). The server must maintain a list of such sinks and call methods on them when an event occurs. When you select Generate Event Support Code, Delphi automatically generates the code necessary to support *IConnectPoint* and *IConnectPointContainer*.

Exposing an application's properties, methods, and events

[Topic groups](#) [See also](#)

When you build the Automation server with the Automation wizard, it automatically generates a type library, which provides a way for host applications to find out what the object can do. To expose the properties, methods, and events of an application, you modify the Automation server's type library as described below.

Exposing a property for Automation

A property is a member function that sets or returns information about the state of the object, such as color or font. For example, a button control might have a property declared as follows:

```
property Caption: WideString;
```

To expose a property for Automation,

- 1 In the type library editor, select the default interface for the Automation object.
The default interface should be the name of the Automation object preceded by the letter "I". To determine the default, in the Type Library editor, choose the CoClass and Implements tab, and check the list of implemented interfaces for the one marked, "Default."
- 2 To expose a read/write property, click the Property button on the toolbar; otherwise, click the arrow next to the Property button on the toolbar, and then click the type of property to expose.
- 3 In the Attributes pane, specify the name of the property.
- 4 In the Parameters pane, specify the property's return type and add the appropriate parameters.
- 5 On the toolbar, click the Refresh button.

A definition and dummy implementation for the property is inserted into the Automation object's unit files.

- 6 In the dummy implementation for the property, add code (between **try** and **finally** statements) that provides the expected functionality. In many cases, this code will simply call an existing function inside the application.

Exposing a method for Automation

A method can be a procedure or a function. To expose a method for Automation,

- 1 In the Type Library editor, select the default interface for the Automation object.
The default interface should be the name of the Automation object preceded by the letter "I". To determine the default, in the Type Library editor, choose the CoClass and Implements tab, and check the list of implemented interfaces for the one marked, "Default."
- 2 Click the Method button.
- 3 In the Attributes pane, specify the name of the method.
- 4 In the Parameters pane, specify the method's return type and add the appropriate parameters.
- 5 On the toolbar, click the Refresh button.

A definition and dummy implementation for the method is inserted into the Automation object's unit files.

- 6 In the dummy implementation for the method, add code between **try** and **finally** statements that provides the expected functionality. In many cases, this code will simply call an existing function inside the application.

Exposing an event for Automation

To expose an event for Automation,

- 1 In the Automation wizard, check the box, Generate event support code.
The wizard creates an Automation object that includes an Events interface.
- 2 In the Type Library editor, select the Events interface for the Automation object.
The Events interface should be the name of the Automation object preceded by the letter "I" and succeeded by the word "Events."
- 3 Click the Method button from the Type Library toolbar.

- 4 In the Attributes pane, specify the name of the method, such as MyEvent.
- 5 On the toolbar, click the Refresh button.

A definition and dummy implementation for the event is inserted into the Automation object's unit files.

- 6 In the Code Editor, create an event handler inside the *TAutoObject* descendant in the Automation object class. For example,

```
unit ev;
interface
uses
  ComObj, AxCtrls, ActiveX, Project1_TLB;
type
  TMyAutoObject = class (TAutoObject, IConnectionPointContainer, IMyAutoObject)
  private
    .
    .
    .
  public
    procedure Initialize; override;
    procedure EventHandler; { Add an event handler}
```

- 7 At the end of the *Initialize* method, assign an event to the event handler you just created. For example,

```
procedure TMyAutoObject.Initialize;
begin
  inherited Initialize;
  FConnectionPoints:= TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo <> nil then
    FConnectionPoints.CreateConnectionPoint (AutoFactory.EventIID,
      ckSingle, EventConnect);
  OnEvent = EventHandler; { Assign an event to the event handler }
end;
```

- 8 Add the necessary code to call the method implemented by the sink. For example, provide the following code substituting the name of your event for "MyEvent."

```
procedure TMyAutoObject.EventHandler;
begin
  if FEvents <> nil then FEvents.MyEvent; { Call method implemented by the
sink.}
end;
```

Getting more information

Press F1 anywhere in the Type Library Editor to get more information on using the editor.

Registering an application as an Automation server

[Topic groups](#)

You can register the Automation server as an in-process or an out-of-process server. For more information on the server types, see [In-process, out-of-process, and remote servers](#).

Note: When you want to remove the Automation server from your system, it is recommended that you first unregister it, removing its entries from the Windows registry.

Registering an in-process server

To register an in-process server (DLL or OCX),

- Choose Run|Register ActiveX Server.

To unregister an in-process server,

- Choose Run|Unregister ActiveX Server.

Registering an out-of-process server

To register an out-of-process server,

- Run the server with the **/regserver** command-line option.
You can set command-line options with the Run|Parameters dialog box.
You can also register the server by running it.

To unregister an out-of-process server,

- Run the server with the **/unregserver** command-line option.

Testing and debugging the application

[Topic groups](#) [See also](#)

To test and debug an Automation server,

- 1 Turn on debugging information using the Compiler tab on the Project|Options dialog box, if necessary. Also, turn on Integrated Debugging in the Tools|Debugger Options dialog.
- 2 For an in-process server, choose Run|Parameters, type the name of the Automation controller in the Host Application box, and choose OK.
- 3 Choose Run|Run.
- 4 Set breakpoints in the Automation server.
- 5 Use the Automation controller to interact with the Automation server.

The Automation server pauses when the breakpoints are reached.

Automation interfaces

Topic groups

Delphi wizards implement the dual interface by default, which means that the Automation object supports both

- Late binding at runtime, which is through the *IDispatch* interface. This is implemented as a dispatch interface, or **dispinterface**.
- Early binding at compile-time, which is accomplished through directly calling one of the member functions in the object's virtual function table (VTable). This is referred to as a **custom interface**.

Dual interfaces

Topic groups

A dual interface is a custom interface and a dispinterface at the same time. It is implemented as a COM VTable interface that derives from *IDispatch*. For those controllers that can access the object only at runtime, the dispinterface is available. For objects that can take advantage of compile-time binding, the more efficient VTable interface is used.

Dual interfaces offer the following combined advantages of VTable interfaces and dispinterfaces:

- For VTable interfaces, the compiler performs type checking and provides more informative error messages.
- For Automation controllers that cannot obtain type information, the dispinterface provides runtime access to the object.
- For in-process servers, you have the benefit of fast access through VTable interfaces.
- For out-of-process servers, COM marshals data for both VTable interfaces and dispinterfaces.

COM provides a generic proxy/stub implementation that can marshal the interface based on the information contained in a type library.

The first three entries of the VTable for a dual interface refer to the *IUnknown interface*, the next four entries refer to the *IDispatch interface*, and the remaining entries are COM entries for direct access to members of the custom interface.

Dispatch interfaces

[Topic groups](#)

Automation controllers are clients that use the COM *IDispatch* interface to access the COM server objects. The controller must first create the object, then query the object's *IUnknown* interface for a pointer to its *IDispatch* interface. *IDispatch* keeps track of methods and properties internally by a dispatch identifier (dispID), which is a unique identification number for an interface member. Through *IDispatch*, a controller retrieves the object's type information for the dispatch interface and then maps interface member names to specific dispIDs. These dispIDs are available at runtime, and controllers get them by calling the *IDispatch* method, *GetIDsOfNames*.

Once it has the dispID, the controller can then call the *IDispatch* method, *Invoke*, to execute the appropriate code (property or method), packaging the parameters for the property or method into one of the *Invoke* parameters. *Invoke* has a fixed compile-time signature that allows it to accept any number of arguments when calling an interface method.

The Automation object's implementation of *Invoke* must then unpackage the parameters, call the property or method, and be prepared to handle any errors that occur. When the property or method returns, the object passes its return value back to the controller.

This is called late binding because the controller binds to the property or method at runtime rather than at compile time.

Custom interfaces

[Topic groups](#)

Custom interfaces are user-defined interfaces that allow clients to invoke interface methods based on their order in the VTable and knowledge of the argument types. The VTable lists the addresses of all the properties and methods that are members of the object, including the member functions of the interfaces that it supports. If the object does not support *IDispatch*, the entries for the members of the object's custom interfaces immediately follow the members of *IUnknown*.

If the object has a type library, you can access the custom interface through its VTable layout, which you can get using the Type Library editor. If the object has a type library and also supports *IDispatch*, a client can also get the dispIDs of the *IDispatch* interface and bind directly to a VTable offset. Delphi's type library importer (TLIBIMP) retrieves dispIDs at import time, so clients that use dispinterface wrappers can avoid calls to *GetIDsOfNames*; this information is already in the `_TLB` file. However, clients still need to call *Invoke*.

Marshaling data

[Topic groups](#)

For out-of-process and remote servers, you must consider how COM marshals data outside the current process. You can provide marshaling:

- Automatically, by using the *IDispatch* interface.
- Automatically, by creating a type library with your server and marking the interface with the Ole Automation flag. COM knows how to marshal all the **Automation-compatible** types in the type library and can set up the proxies and stubs for you. Some type restrictions apply to enable automatic marshaling.
- Manually by implementing all the methods of the *IMarshal* interface. This is called **custom marshaling**.

Automation compatible types

Function result and parameter types of the methods declared in dual and dispatch interfaces must be *Automation-compatible* types. The following types are OLE Automation-compatible:

- The predefined valid types such as *Smallint*, *Integer*, *Single*, *Double*, *WideString*. For a complete list, see [Valid types](#).
- Enumeration types defined in a type library. OLE Automation-compatible enumeration types are stored as 32-bit values and are treated as values of type *Integer* for purposes of parameter passing.
- Interface types defined in a type library that are OLE Automation safe, that is, derived from *IDispatch* and containing only OLE Automation compatible types.
- Dispinterface types defined in a type library.
- *IFont*, *IStrings*, and *IPicture*. Helper objects must be instantiated to map
 - an *IFont* to a *TFont*
 - an *IStrings* to a *TStrings*
 - an *IPicture* to a *TPicture*

The ActiveX control and ActiveForm wizards create these helper objects automatically when needed. To use the helper objects, call the global routines, [GetOleFont](#), [GetOleStrings](#), [GetOlePicture](#), respectively.

Type restrictions for automatic marshaling

For an interface to support automatic marshaling, the following restrictions apply. When you edit your Automation object using the type library editor, the editor enforces these restrictions:

- Types must be compatible for cross-platform communication. For example, you cannot use data structures (other than implementing another property object), unsigned arguments, wide strings, and so on.
- String data types must be transferred as BSTR. PChar and AnsiString cannot be marshaled safely.
- All members of a dual interface must pass an HRESULT as the function's return value.
- Members of a dual interface that need to return other values should specify these parameters as **var** or **out**, indicating an output parameter that returns the value of the function.

Note: One way to bypass the Automation types restrictions is to implement a separate *IDispatch* interface and a custom interface. By doing so, you can use the full range of possible argument types. This means that COM clients have the option of using the custom interface, which Automation controllers can still access. In this case, though, you must implement the marshaling code manually.

Custom marshaling

Typically, you will use automatic marshaling in your out-of-process and remote servers because it is easier--COM does the work for you. However, you may decide to provide custom marshaling if you think you can improve marshaling performance.

Creating an ActiveX control

[Topic groups](#) [See also](#)

An ActiveX control is a software component that integrates into and extends the functionality of any host application that supports ActiveX controls, such as C++Builder, Delphi, Visual dBASE, Visual Basic, Internet Explorer, and Netscape Navigator.

For example, Delphi comes with several ActiveX controls, including charting, spreadsheet, and graphics controls. You can add these controls to the component palette in the IDE, and then use them like any standard VCL component, dropping them on forms and setting their properties using the Object Inspector.

An ActiveX control can also be deployed on the Web, allowing it to be referenced in HTML documents and viewed with ActiveX-enabled Web browsers.

This is an overview of how to create an ActiveX control in the Delphi environment. It is not intended to provide complete implementation details of writing ActiveX control. For that information, refer to your Microsoft Developer's Network (MSDN) documentation or search the Microsoft Web site for ActiveX information.

Overview of ActiveX control creation

Delphi provides two wizards for ActiveX development:

- The ActiveX Control wizard allows you to convert an existing VCL or custom-VCL control into an ActiveX control by adding an ActiveX class wrapper around the VCL control.
- The ActiveForm wizard allows you to create a new ActiveX application from the start. The wizard sets up the project and adds a blank form so that you can begin to add the controls to design the form.

The ActiveX Control wizard generates an implementation unit that descends from two objects, *TActiveXControl* for the ActiveX control specifics, and the VCL object of the control that you choose to encapsulate. The ActiveForm wizard generates an implementation unit that descends from *TActiveForm*.

To create a new ActiveX control, you must perform the following steps:

- 1 [Design](#) and create the custom VCL control that forms the basis of your ActiveX control.
- 2 Use the [ActiveX control wizard](#) to create an ActiveX control from a VCL control
or,
Use the [ActiveForm wizard](#) to create an ActiveX control based on a VCL form for Web deployment.
- 3 Use the [Use the ActiveX property page wizard](#) to create one or more property pages for the control (optional).
- 4 [Associate a property page with an ActiveX control](#)(optional).
- 5 [Register an ActiveX control](#).
- 6 [Test an ActiveX control](#) with all potential target applications.
- 7 [Deploy](#) an ActiveX control on the Web.

An ActiveX control consists of the VCL control from which it is built, as well as properties, methods, and events that are listed in the control's type library. For details, see [Elements of an ActiveX control](#).

Elements of an ActiveX control

[Topic groups](#) [See also](#)

An ActiveX control involves many elements which each perform a specific function. The elements include a VCL control, properties, methods, events, and one or more associated type libraries.

VCL control

An ActiveX control in Delphi is simply a VCL control that has been made accessible to applications and objects that support ActiveX controls. When you create an ActiveX control, you must first design or choose the VCL control from which you will make your ActiveX control.

Note: The controls available from the wizard list are controls derived from TWinControl. Some controls, such as EditControl, are registered as a NonActiveX control, therefore, they do not appear in the list.

Type library

A type library contains the type definitions for the control and is created automatically by the ActiveX control wizard. This type information, which provides more details than the interface, provides a way for controls to advertise its services to host applications. When you design your control, type library information is stored in a file with the TLB extension and a corresponding Pascal file containing the Pascal translations. When you build the ActiveX control, the type library information is automatically compiled into the ActiveX control DLL as a resource.

Properties, methods, and events

The properties, events, and methods of the VCL control become those of the ActiveX control.

- A property is an attribute, such as a color or label.
- A method is a request to a control to perform some action.
- An event is a notification from a control to the container that something has happened.

Property page

The property page allows the user of a control to view and edit its properties. You can group several properties on a page, or use a page to provide a dialog-like interface for a property. For information on how to create property pages, see [creating a property page for an ActiveX control](#).

Designing an ActiveX control

[Topic groups](#) [See also](#)

When designing an ActiveX control, you start by creating a custom VCL control. This forms the basis of your ActiveX control. For information on creating the custom VCL control, see the online manual, [Creating custom components](#).

When designing ActiveX controls from either existing VCL controls or from new ActiveForms, keep in mind that you are implementing an ActiveX control that will be embedded in another application; this control is not an application in itself.

For this reason, you probably do not want to use elaborate dialog boxes or other major user-interface components. Your goal is typically to make a simple control that works inside of, and follows the rules of the main application.

Whether you create an ActiveX control from a VCL control or ActiveForm depends on whether you already have an existing control that you simply want to encase in an ActiveX control wrapper. In this case, use the ActiveX control wizard as described in [Generating an ActiveX control from a VCL control](#).

If you are creating a more complete ActiveX application, use the ActiveForm wizard as described in [Generating an ActiveX control based on a form](#). Typically, you use an ActiveForm to create and supply ActiveX applications for the Web.

The wizards implement all the necessary ActiveX interfaces required using the VCL objects, *TActiveXControl* or *TActiveForm*. You need only implement any additional interfaces you may have added to your control.

Once you have generated the control using either wizard, you can modify the control's properties, methods, and events using the Type Library editor.

Generating an ActiveX control from a VCL control

[Topic groups](#) [See also](#)

You generate an ActiveX control from a VCL control by using the ActiveX Control wizard. The properties, methods, and events of the VCL control become the properties, methods, and events of the ActiveX control.

The ActiveX control wizard actually puts an ActiveX class wrapper around a VCL control and builds the ActiveX control that contains the object. This ActiveX wrapper exposes the capabilities of the VCL control to other objects and servers.

Before using the ActiveX control wizard, you must select the VCL control from which to build the ActiveX control.

To bring up the ActiveX control wizard,

- 1 Choose File|New to open the New Items dialog box.
- 2 Select the tab labeled ActiveX.
- 3 Double-click the ActiveX Control icon.

In the wizard, specify the following:

VCL ClassName	Choose the VCL control from the drop-down list. For example, to create an ActiveX control that allows client applications to use a <i>TButton</i> object, select <i>TButton</i> . For <i>ActiveForms</i> , the VCL class name option is dimmed because Active forms are always based on <i>TActiveForm</i> .
New Name	The wizard supplies a default name that clients will use to identify your ActiveX control. Change this name to provide a different OLE class name.
Implementation Unit	The wizard supplies a default name for the unit that contains code to implement the behavior of the ActiveX control. Accept the default or type in a new name.
Project Name	The wizard supplies a default name for the ActiveX library project for your ActiveX control, if no project is currently open. If you have an ActiveX library open, this option is disabled.
Threading Model	Choose the <u>threading model</u> to indicate how client applications can call your control's interface. This is the threading model that you commit to implementing in the control. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.

Specify the following ActiveX control options:

Make Control Licensed	Check this box to <u>enable licensing</u> for the ActiveX controls that you design and distribute (unless the control is distributed free of charge). The wizard generates both a design-time license for designers of the control, and a runtime license for users of the control.
Include Version Information	Check this box to include version information, such as a copyright or a file description, in the ActiveX control. This information can be viewed in a browser. Specify version information by choosing Project Options and selecting the <u>Version Info page</u> . Note: Version information is required for registering a control in Visual Basic 4.0.

Include About Box

When this box is checked, an About box is included in the project. The user of the control can display the About box in a development environment. The About box is a separate form that you can modify. By default, the About box includes the name of the ActiveX control, an image, copyright information, and an OK button.

The wizard generates the following files:

- An ActiveX Library project file, which contains the code required to start an ActiveX control. You usually don't change this file.
- A type library (with the TLB extension), which defines and implements the properties, methods, and events that the ActiveX control exposes for Automation. For more information about the type library, see [Working with type libraries](#).
- An ActiveX implementation unit, which defines and implements the ActiveX control using the Delphi ActiveX framework (DAX). DAX is Delphi's implementation of the COM specification for ActiveX controls. You can modify the implementation to suit your needs.
- An About box form and unit (if Include About Box is checked).

Licensing ActiveX controls

[Topic groups](#) [See also](#)

Licensing an ActiveX control consists of providing a license key at design-time and supporting the creation of licenses dynamically, when the control is created at runtime.

To provide design-time licenses, the ActiveX wizard creates a key for the control that is stored in a file with the same name as the project and with the LIC extension. The user of the control must have a copy of the LIC file to open the control in a development environment. Each control in the project that has Make Control Licensed checked will have a separate key entry in the LIC file.

To support the creation of runtime licenses, the license is generated by querying the control for the license (at design time), storing it, and then passing the license to the control when it is created in the context of an EXE. When the runtime license is passed to the control, the design time license is no longer required.

Runtime licenses for the Internet Explorer requires an extra level of indirection because users can view HTML source code for any Web page, and because an ActiveX control is copied to the user's computer before it is displayed. To create runtime licenses for controls used in IE, you must first generate a license package file (LPK file) and embed this file in the HTML page that contains the control.

The LPK file is essentially an array of ActiveX control CLSIDs and license keys.

Note: To generate the LPK file, use the utility, LPK_TOOL.EXE, which you can download from the Microsoft Web site (www.microsoft.com).

To embed the LPK file in a Web page, use the HTML objects, <OBJECT> and <PARAM> as follows:

```
<OBJECT CLASSID="clsid:6980CB99-f75D-84cf-B254-55CA55A69452">  
  <PARAM NAME="LPKPath" VALUE="ctrllic.lpk">  
</OBJECT>
```

The CLSID identifies the object as a license package and PARAM specifies the relative location of the license package file with respect to the HTML page.

When IE tries to display the Web page containing the control, it parses the LPK file, extracts the license key, and if the license key matches the control's license, it renders the control on the page. If more than one LPK is included in a Web page, IE ignores all but the first.

For more information, search for Licensing ActiveX Controls on the Microsoft Web site.

Generating an ActiveX control based on a VCL form

[Topic groups](#) [See also](#)

The ActiveForm wizard generates an ActiveX control based on a VCL form which you design when the wizard leaves you in the Form Designer. You can use the ActiveForm to create applications that you can deploy on the Web.

When an ActiveForm is deployed, an HTML page is created to contain the reference to the ActiveForm and specify its location within the page. The ActiveForm is then displayed and run from within a Web browser. Inside the Web browser, the form behaves just like a stand-alone Delphi form. The form may contain any VCL or ActiveX components, including custom-built VCL controls.

To start the ActiveForm wizard,

- 1 Choose File|New to open the New Items dialog box.
- 2 Select the tab labeled ActiveX.
- 3 Double-click the ActiveForm icon.

In the wizard, specify the following:

VCL ClassName	The VCL class name option is dimmed because Active forms are always based on <i>TActiveForm</i> .
New Name	The wizard supplies a default name that clients will use to identify your ActiveX control. Change this name to provide a different OLE class name.
Implementation Unit	The wizard supplies a default name for the unit that contains code to implement the behavior of the ActiveX control. Accept the default or type in a new name.
Project Name	The wizard supplies a default name for the ActiveX library project for your ActiveX control, if no project is currently open. If you have an ActiveX library open, this option is disabled.
Threading Model	Choose the <u>threading model</u> to indicate how client applications can call your control's interface. This is the threading model that you commit to implementing in the control. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.

Specify the following ActiveX control options:

Make Control Licensed	Check this box to <u>enable licensing</u> for the ActiveX controls that you design and distribute (unless the control is distributed free of charge). The wizard generates both a design-time license for designers of the control, and a runtime license for users of the control.
Include Version Information	Check this box to include version information, such as a copyright or a file description, in the ActiveX control. This information can be viewed in a browser. Specify version information by choosing Project Options and selecting the <u>Version Info page</u> . Note: Version information is required for registering a control in Visual Basic 4.0.
Include About Box	When this box is checked, an About box is included in the project. The user of the control can display the About box in a development environment. The About box is a separate form that you can modify. By default, the About box includes the name of the ActiveX control, an image, copyright information,

and an OK button.

The wizard generates the following files:

- An ActiveX Library project file, which contains the code required to start an ActiveX control. You usually don't change this file.
- A form.
- A type library (with the TLB extension), which defines and implements the properties, methods, and events that the ActiveX control exposes for Automation. For more information about the type library, see [Working with type libraries](#).
- An ActiveX implementation unit, which defines and implements the properties, methods, and events of *TActiveForm* using the Delphi ActiveX framework (DAX). DAX is Delphi's implementation of the COM specification for ActiveX controls. You can modify the implementation to suit your needs.
- An About box form and unit (if Include About Box is checked).

At this point, the wizard adds a blank form to your ActiveX library project. Now you can add controls and design the form as you like.

After you have designed and compiled the ActiveForm project into an ActiveX library (which has the OCX extension), you can [deploy](#) the project to your Web server and Delphi will create a test HTML page with a reference to the ActiveForm.

Working with properties, methods, and events in an ActiveX control

[Topic groups](#) [See also](#)

The ActiveX Control and ActiveForm wizards generate a type library that defines and implements the properties, methods, and events of the original VCL control or form, with the following exceptions:

- Any property, method, or event that uses a non-OLE type.
- Any property that is data-aware.

Therefore, you may need to add some properties, methods, and events manually.

Note: Since ActiveX controls have a different mechanism for making controls data-aware than a VCL control, the wizards do not convert the properties related to data. You may want to add some of the ActiveX data aware properties to your control as described in [Enabling simple data binding with the type library](#).

You can add, edit, and remove the properties, methods, and events in an ActiveX control by editing the type library using either of the following ways:

- Choose Edit|Add To Interface from the IDE. For details, see [Adding additional properties, methods, and events](#).
- Use the Type Library editor as described in [Using the Type Library Editor](#).

Note: Any changes you make to the type library will be lost if you regenerate the ActiveX control from the original VCL control or form.

Note: Non-published properties that are manually added to the type library will appear in a development environment, but changes made to them will not persist. That is, when the user of the control changes the value of a property, the changes will not be reflected when the control is run. If the source is a VCL object and the property is not already published, you must create a descendant of the VCL object and publish the property in the descendant.

Adding additional properties, methods, and events

[See also](#)

You can add additional properties, methods, and events to the control as follows.

- 1 Choose Edit|Add To Interface. The ActiveX controls implementation unit must be open and selected for the menu item to be available.

This brings up the Add to Interface dialog box.

- 2 Choose Method or Event from the Interface drop-down list. Next to Properties/Methods or Events is the name of the interface to which the member will be added.
- 3 Enter the declaration for the property, method, or event. For example, if you were creating an ActiveX control with the *TButton* VCL control, you could add the following property:

```
property Top:integer;
```

If you check the Syntax helper check box, pop-up windows appear as you type, prompting you for what is expected at each point.

- 4 Choose OK.

The declaration is automatically added to the control's implementation unit, type library (TLB) file, and type library unit. The specifics of what Delphi supplies actually depends on whether you have added a property, method, or event.

How Delphi adds properties

[See also](#)

Since the interface is a dual interface, the properties are implemented using read and write access methods in the Pascal file (with the PAS extension). When adding a Caption property, the wizard would actually add the following declaration to the implementation unit:

```
property Caption: Integer read Get_Caption write Set_Caption;
```

The read and write access method declarations and implementations for the property would be the following:

```
function Get_Caption: Integer; safecall;  
procedure Set_Caption(Value: Integer); safecall;  
function TButtonX.Get_Caption: Integer;  
begin  
    Result := FDelphiControl.Caption;  
end;  
procedure TButtonX.Set_Caption(Value: Integer);  
begin  
    FDelphiControl.Caption := Value;  
end;
```

Note: Because the Automation interface methods are declared **safecall**, you do not have to implement COM exception code for these methods—the Delphi compiler handles this for you by generating code around the body of **safecall** methods to catch Delphi exceptions and to convert them into COM error info structures and return codes.

The interface implementation methods simply pass through the behavior to the VCL control. In some cases, you may need to add code to convert the COM data types to native Delphi types.

How Delphi adds methods

[See also](#)

When you add a method, an empty implementation is added for you to add its functionality. For example, if you added:

```
procedure Move;
```

the following declaration and code would be added to the unit:

```
procedure Move; safecall;  
procedure TButtonX.Move;  
begin  
end;
```

Note: Because the Automation interface methods are declared **safecall**, you do not have to implement COM exception code for these methods—the Delphi compiler handles this for you by generating code around the body of **safecall** methods to catch Delphi exceptions and to convert them into COM error info structures and return codes.

How Delphi adds events

[See also](#)

An ActiveX control can fire events to its container. You add events to specify which events can be fired by the control. The following items are created for each event on the control:

- A dispatch interface declaration and implementation entries in the implementation unit.

```
procedure KeyPressEvent(Sender: TObject; var Key: Char);  
  
procedure TButtonX.KeyPressEvent(Sender: TObject; var Key: Char);  
var  
    TempKey: Smallint;  
begin  
    TempKey := Smallint(Key);  
    if FEvents <> nil then FEvents.OnKeyPress(TempKey);  
    Key := Char(TempKey);  
end;
```
- A type library entry for the event interface with dispinterface checked on its attributes page.
- An Object Pascal version of the type library's interface source file.

Unlike the Automation interface, the events interface is not added to the control's interface list. The control does *not* receive these events—the *container* does.

Enabling simple data binding with the type library

[Topic groups](#) [See also](#)

With simple data binding, you can bind a property of your ActiveX control to a specific field within a database. You can do so by setting the property's binding flags using the Type Library editor.

This topic describes how to bind data-aware properties in an ActiveX control. For information on binding data-aware properties in a Delphi container, see [Enabling simple data binding of ActiveX controls in the container](#).

By marking a property bindable, when a user modifies the property (such as a field in a database), the control notifies the database that the value has changed and requests that the database record be updated. The database then notifies the control whether it succeeded or failed to update the record.

Use the type library to enable simple data binding,

- 1 On the toolbar, click the property that you want to bind.
- 2 Choose the flags page.
- 3 Select the following binding attributes:

<u>Binding attribute</u>	<u>Description</u>
Bindable	Indicates that the property supports data binding. If marked bindable, the property notifies its container when the property value has changed.
Request Edit	Indicates that the property supports the OnRequestEdit notification. This allows the control to ask the container if its value can be edited by the user.
Display Bindable	Indicates that the container can show users that this property is bindable.
Default Bindable	Indicates the single, bindable property that best represents the object. Properties that have the default bind attribute must also have the bindable attribute. Cannot be specified on more than one property in a dispinterface.
Immediate Bindable	Allows individual bindable properties on a form to specify this behavior. When this bit is set, all changes will be notified. The bindable and request edit attribute bits need to be set for this new bit to have an effect.

- 4 Click the Refresh button on the toolbar to update the type library.

To test a data-binding control, you must [register](#) it first.

For example, to make a *TEdit* control a data-bound ActiveX control, create the ActiveX control from a *TEdit* and then change the Text property flags to Bindable, Display Bindable, Default Bindable, and Immediate Bindable. After the control is registered and imported, it can be used to display data.

Enabling simple data binding of ActiveX controls in the Delphi container

[Topic groups](#) [See also](#)

After installing a data-aware ActiveX control in the ActiveX tab of the Palette, and placing the control in the form designer, right-click the data-aware ActiveX control to display a list of options. In addition to the basic [Form context menu](#) options, the additional DataBindings item appears.

Note: You must set the data source property to the data source component on the form before invoking the Data Bindings Editor. In doing so, the dialog supplies the Field Name and Property fields from the data source component. The editor lists only those properties from the data source component that can be data-bound properties of the ActiveX control.

To bind a field to a property,

- 1 In the ActiveX Data Bindings Editor dialog, select a field and a property name.
Field Name lists the fields of the database and Property Name lists the ActiveX control properties that can be bound to a database field. The DispID of the property is in parentheses, for example, Value(12).
- 2 Click Bind and OK.

Note: If no properties appear in the dialog, the ActiveX control contains no data-aware properties. To enable simple data binding for a property of an ActiveX control, use the [type library](#).

The following example walks you through the steps of using a data-aware ActiveX control in the Delphi container. In this example, we use the Microsoft Calendar Control, which is available if you have Microsoft Office 97 installed on your system.

- 1 From the Delphi main menu, choose Component|Import ActiveX.
- 2 Select a data-aware ActiveX control, such as the Microsoft Calendar control 8.0, change its class name to *TCalendarAXControl*, and click Install.
- 3 In the Install dialog, click OK to add the control to the default user package, which makes the control available on the Palette.
- 4 Choose Close All and File|New Application to begin a new application.
- 5 From the ActiveX tab, drop a *TCalendarAXControl* object, which you just added to the Palette, onto the form.
- 6 From the Data Access tab, drop a DataSource and Table object onto the form.
- 7 Select the DataSource object and set its DataSet property to Table1.
- 8 Select the Table object and do the following:
 - Set the DataBaseName property to DBDEMOS
 - Set the TableName property to EMPLOYEE.DB
 - Set the Active property to True
- 9 Select the *TCalendarAXControl* object and set its DataSource property to DataSource1.
- 10 Select the *TCalendarAXControl* object, right-click, and choose Data Bindings to invoke the ActiveX Control Data Bindings Editor.
Field Name lists all the fields in the active database. Property Name lists those properties of the ActiveX Control that can be bound to a database field. The DispID of the property is in parentheses.
- 11 Select the HireDate field and the Value property name, choose Bind, and OK.
The field name and property are now bound.
- 12 From the Data Controls tab, drop a DBGrid object onto the form and set its DataSource property to DataSource1.
- 13 From the Data Controls tab, drop a DBNavigator object onto the form and set its DataSource property to DataSource1.
- 14 Run the application.
- 15 Test the application as follows:
With the HireDate field displayed in the DBGrid object, navigate through the database using the Navigator object. The dates in the ActiveX control change as you move through the database.

Creating a property page for an ActiveX control

[Topic groups](#) [See also](#)

A property page is a dialog box similar to the Delphi Object Inspector in which users can change the properties of an ActiveX control. A property page dialog allows you to group many properties for a control together to be edited at once. Or, you can provide a dialog box for more complex properties.

Typically, users access the property page by right-clicking the ActiveX control and choosing Properties.

The process of creating a property page is similar to creating a form, you

- 1 [Create a new property page.](#)
- 2 [Add controls to the property page.](#)
- 3 [Associate the controls the property page with the properties of an ActiveX control.](#)
- 4 [Connect the property page to the ActiveX control.](#)

Note: When adding properties to an ActiveX control or ActiveForm, you must publish the properties that you want to persist. For details see [Exposing properties of an ActiveX control.](#)

Creating a new property page

[Topic groups](#) [See also](#)

You use the Property Page wizard to create a new property page.

To create a new property page,

- 1 Choose File|New.
- 2 Select the ActiveX tab.
- 3 Double-click the Property Page icon.

The wizard creates a new form and implementation unit for the property page.

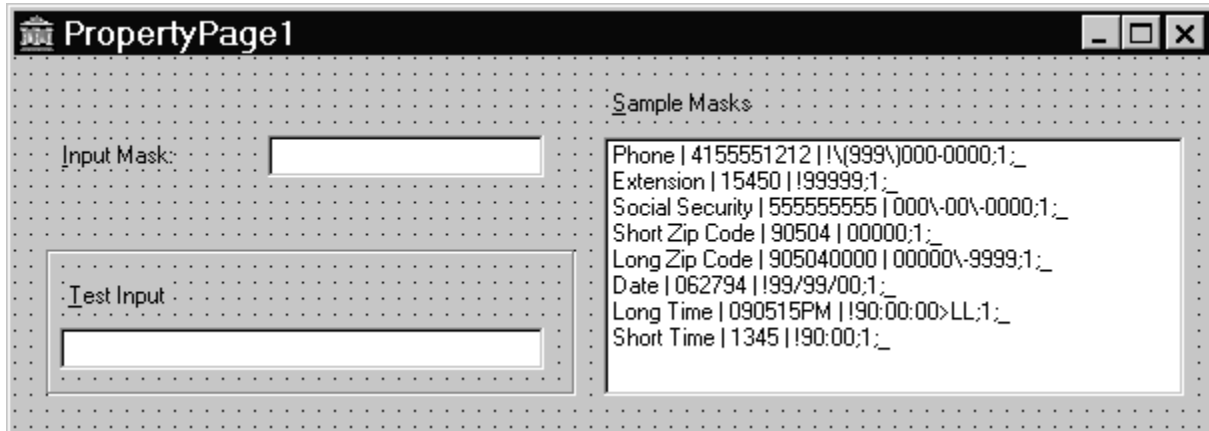
Adding controls to a property page

[Topic groups](#) [See also](#)

You must add a control to the property page for each property of the ActiveX control that you want the user to access.

For example, the following illustration shows a property page for setting the MaskEdit property of an ActiveX control.

Mask Edit property page in design mode



The list box allows the user to select from a list of sample masks. The edit controls allow the user to test the mask before applying it to the ActiveX control. You add controls to the property page the same as you would to a form.

Associating property page controls with ActiveX control properties

[Topic groups](#) [See also](#)

After you have added all the controls you need to the property page, you must associate each control with its corresponding property. You make this association by adding code to the property page's implementation unit. Specifically, you must add code to the [UpdatePropertyPage](#) and [UpdateObject](#) methods.

Updating the property page

[Topic groups](#) [See also](#)

Add code to the *UpdatePropertyPage* method to update the property's control on the property page when the properties of the ActiveX control change. You must add code to the *UpdatePropertyPage* method to update the property page with the current values of the ActiveX control's properties.

For example, the following code updates the property page's edit box control (InputMask) with the current value of the ActiveX control's (OleObject's) EditMask property:

```
procedure TPropertyPage1.UpdatePropertyPage;  
begin  
    { Update your controls from OleObjects }  
    InputMask.Text := OleObject.EditMask;  
end;
```

Updating the object

[Topic groups](#) [See also](#)

Add code to the *UpdateObject* method to update the property when the user changes the controls on the property page. You must add code to the *UpdateObject* method in order to set the properties of the ActiveX control to their new values.

For example, the following code sets the *EditMask* property of an ActiveX control (OleObject) using the value in the property page's edit box control (InputMask):

Note: Include the .TLB file that declares the *ICoClassNameDisp* interface.

```
procedure TPropertyPage1.UpdateObject;  
begin  
  {Update OleObjects from your controls }  
  OleObject.EditMask := InputMask.Text;  
end;
```

Connecting a property page to an ActiveX control

[Topic groups](#) [See also](#)

To connect a property page to an ActiveX control,

- 1 Add *DefinePropertyPage* with the GUID constant of the property page as the parameter to the *DefinePropertyPages* method implementation in the controls implementation for the unit. For example,

```
procedure TButtonX.DefinePropertyPages (DefinePropertyPage: TDefinePropertyPage);  
begin  
    DefinePropertyPage (Class_PropertyPage1);  
end;
```

The GUID constant, Class_PropertyPage1, of the property page can be found in the property pages unit.

The GUID is defined in the property page's implementation unit; it is generated automatically by the Property Page wizard.

- 2 Add the property page unit to the **uses** clause of the controls implementation unit.

Exposing properties of an ActiveX control

[Topic groups](#) [See also](#)

The properties of an ActiveX control or ActiveForm can appear in a development environment such as Visual Basic, Delphi, or C++Builder. The user of the control or form can then visually manipulate the properties of the control. For properties to appear, they must be defined in the control's type library. For property changes to persist, they must be published. You can add properties to the type library manually, or let Delphi add them automatically. Delphi automatically adds published properties to the type library for you.

Note: Non-published properties that are manually added to the type library will appear in a development environment, but changes made to them will not persist. That is, when the user of the control changes the value of a property, the changes will not be reflected when the control is run.

To expose a property for a control that you created using the ActiveX control wizard, go to the source of the VCL control to expose the necessary properties. If the source is a VCL object and the property is not already published, you must create a descendant of the VCL object and publish the property in the descendant. For example, to publish the *Align* property of *TButton*, you can add the following code to the implementation unit of the ActiveX control:

```
TAlignButton = class (TButton)
  published
    property Align;
end;
```

To expose a property for an ActiveForm,

- 1 Select Edit|Add to interface while the implementation unit is open in the editor.
- 2 Select Property/Method from the Procedure type drop-down list.
- 3 Enter the declaration for the property that will expose the control's property. For example, for the *Caption* property of a button control you could type,

```
property MyButtonCaption: WideString;
```

- 4 Choose OK.

This adds get and set methods to the implementation unit.

- 5 Add code to the methods to get and set the controls property. For example, with the button control above, the code would look as follows:

```
function TActiveFormX.Get_MyButtonCaption: WideString;
begin
  Result := Button1.Caption;
end;
procedure TActiveFormX.Set_MyButtonCaption(const Value: WideString);
begin
  Button1.Caption := Value;
end;
```

Since an ActiveForm is an ActiveX control, there is a type library associated with it, and you can add additional properties and methods to the ActiveForm. For information on how to do this, see [Working with properties, methods, and events in an ActiveX control](#).

Registering an ActiveX control

[Topic groups](#)

After you have created your ActiveX control, you must register it so that other applications can find and use it.

To register an ActiveX control:

- Choose Run|Register ActiveX Server.

Note: Before you remove an ActiveX control from your system, you should unregister it.

To unregister an ActiveX control:

- Choose Run|Unregister ActiveX Server.

As an alternative, you can use the **regsvr** command from the command line or run the regsvr32.exe from the operating system.

Testing an ActiveX control

[Topic groups](#) [See also](#)

To test your control, add it to a package and import it as an ActiveX control. This procedure adds the ActiveX control to the Delphi component palette. You can drop the control on a form and test as needed.

Your control should also be tested in all target applications that will use the control.

To debug the ActiveX control, select Run|Parameters and type the client name in the Host Application edit box.

The parameters then apply to the host application. Selecting Run|Run will run the host or client application and allow you to set breakpoints in the control.

Deploying an ActiveX control on the Web

[Topic groups](#) [See also](#)

Before the ActiveX controls that you create can be used by Web clients, they must be deployed on your Web server. Every time you make a change to the ActiveX control, you must recompile and redeploy it so that client applications can see the changes.

Before you can deploy your ActiveX control, you must have a Web Server that will respond to client messages. You can purchase a third-party Web Server, such as the Inprise Web Server that ships with the IntraBuilder product, or you can build your own Web Server if you have a version of Delphi that ships with socket components.

The process of deploying an ActiveX control,

- 1 Select Project|Web Deployment Options.
- 2 On the Project page, set the Target Dir to the location of the ActiveX control DLL as a path on the Web server. This can be a local path name or a UNC path, for example, C:\INETPUB\wwwroot.
- 3 Set the Target URL to the location as a Uniform Resource Locators (URL) of the ActiveX control DLL (without the file name) on your Web Server, for example, <http://mymachine.inprise.com/>. See the documentation for your Web Server for more information on how to do this.
- 4 Set the HTML Dir to the location (as a path) where the HTML file that contains a reference to the ActiveX control should be placed, for example, C:\INETPUB\wwwroot. This path can be a standard path name or a UNC path.
- 5 Set desired [Web deployment options](#).
- 6 Choose OK.
- 7 Choose Project|Web Deploy.

This creates a deployment code base that contains the ActiveX control in an ActiveX library (with the OCX extension). Depending on the options you specify, this deployment code base can also contain a cabinet (with the CAB extension) or information (with the INF extension).

The ActiveX library is placed in the Target Dir of step 2. The HTML file has the same name as the project file but with the HTM extension. It is created in the HTML Dir of step 4. The HTML file contains a URL reference to the ActiveX library at the location specified in step 3.

Note: If you want to put these files on your Web server, use an external utility such as ftp.

- 8 Invoke your ActiveX-enabled Web browser and view the created HTML page.

When this HTML page is viewed in the Web browser, your form or control is displayed and runs as an embedded application within the browser. That is, the library runs in the same process as the browser application.

Setting Web deployment options

[Topic groups](#) [See also](#)

Before deploying an ActiveX control, specify the Web deployment options that should be followed when creating the ActiveX library.

Web deployment options are listed in the tabs:

- [Project tab](#)
- [Packages tab](#)
- [Additional files tab](#)

Web deployment options include settings to allow you to set the following:

Compress files by setting CAB file	A cabinet is a single file, usually with a CAB file extension, that stores compressed files in a file library. Cabinet compression can dramatically decrease download time (up to 70%) of a file. During installation, the browser decompresses the files stored in a cabinet and copies them to the user's system. Each file that you deploy can be CAB file compressed.
Set options for packages	You can set specific options for each required package file as part of your deployment. These packages can each be put into CAB files. Packages that ship with Delphi are code signed with the Inprise signature.

If you choose different combinations of package and CAB file compression options for your ActiveX library, the resulting ActiveX library may be an OCX file, a CAB file containing an OCX file, or an INF file. See the [Option Combinations table](#) for details.

Web Deploy Options Default checkbox

Checking this box saves the current settings from the Project, Packages, and Additional Files pages of the Web Deployment Options dialog box as the default options. If it is not checked, the settings affect only the open ActiveX project.

To restore the original settings, delete or rename the DEFPROJ.DOF file.

INF file

If your ActiveX control depends on any packages or other additional files, these must be included when you deploy the ActiveX control. When the ActiveX control is deployed with additional packages or other additional files, a file with the INF extension (for INformation) is automatically created. This file specifies various files that need to be downloaded and set up for the ActiveX library to run. The syntax of the INF file allows URLs pointing to packages or additional files to download.

Option combinations

[Topic groups](#) [See also](#)

The following table summarizes the results of choosing different combinations of package and CAB file compression Web deployment options.

<u>Packages and/or additional files</u>	<u>CAB file compression</u>	<u>Result</u>
No	No	An ActiveX library (OCX) file.
No	Yes	A CAB file containing an ActiveX library file.
Yes	No	An INF file, an ActiveX library file, and any additional files and packages.
Yes	Yes	An INF file, a CAB file containing an ActiveX library, and a CAB file each for any additional files and packages.

Project tab

[Topic groups](#) [See also](#)

The project tab enables you to specify file locations, URL, and other deployment options for the project. The options on the project tab apply to the ActiveX library file or CAB file containing the ActiveX control and become the default options for any packages and additional files deployed with the project.

<u>Directories and URLs</u>	<u>Meaning</u>
Target Dir	Location as a path of the ActiveX library file on the Web server. This can be a standard path name or a UNC path. Example: C:\INETPUB\wwwroot
Target URL	Location as a Uniform Resource Locator (URL) of the path for the ActiveX library file on the Web Server. Example: http://mymachine.inprise.com/
HTML Dir	Location as a path of the HTML file that contains a reference to the ActiveX library. This can be a standard path name or a UNC path. Example: C:\INETPUB\wwwroot

Note: The locations specified are *paths* only, *not* complete file names.

In addition to specifying the locations for your ActiveX files, the project tab also allows you to specify whether to use CAB file compression, version numbers, and so on. The following table lists the options you can select:

<u>General options</u>	<u>Meaning</u>
Use CAB file compression	Compress the ActiveX library, and required packages and additional files, unless specified otherwise. Cabinet compression stores files in a file library, which can decrease a file's download time by up to 70%.
Include file version number	Include version number contained in Project Options VersionInfo.
Auto increment release number	Automatically increment the projects release number contained in Project Options VersionInfo.
Deploy required packages	If checked, deploy the project's required packages.
Deploy additional files	If checked, deploy the additional files specified on the Addition Files tab with the project.

Packages tab

[Topic groups](#) [See also](#)

The packages tab lets you specify how to deploy the packages used by the project. Each package can have its own settings. When deploying your ActiveX control, you can specify individual deployment options for each required package file used by the project as part of your deployment. Each of these packages can be put into CAB files. Packages that ship with Delphi are code signed with the Inprise signature.

To modify the settings for a particular package, select the package in the “Packages used by project” list box, and modify the settings as needed.

<u>CAB option</u>	<u>Meaning</u>
Compress in a separate CAB	Create a separate CAB file for the package. This is the default.
Compress in a project CAB	Include the package in the project CAB file.

Version Info

The Version Info check box lets you to specify if the package has version info. The version info comes from the resource in the package file and is entered into the INF file for the project.

<u>Directory and URL option</u>	<u>Meaning</u>
Target URL (or leave blank to assume file exists on target machines)	Location as a Uniform Resource Locators (URL) of the package on your Web Server. If left blank, the Web browser assumes the file exists on the target machine. If the package is not found on the target machine, the download of the ActiveX control fails.
Target directory (or leave blank to assume file exists on server)	Location as a path of the package on the Web server. This can be a standard path name or a UNC path. Leave blank to assume the file exists and ensure that it will not be overwritten.

Additional Files tab

[Topic groups](#) [See also](#)

Specifies other files associated with this project.

If the ActiveX control is deployed with any packages and/or additional files, an INF file is automatically created. This file specifies various files that must be downloaded and setup for the ActiveX control to run. The syntax for the INF file allows URLs pointing to files to download, and it allows for platform independence (by enumerating files for various platforms).

CAB option	Meaning
Compress in a separate CAB	Create a separate CAB file for the file. This is the default.
Compress in a project CAB	Include the file in the project CAB file.

Version Info

The Version Info checkbox lets you specify that the INF should contain version information. This information is fetched from the resource in the file.

Directory and URL option	Meaning
Target URL (or leave blank to assume file exists on target machines)	Location as a Uniform Resource Locators (URL) of the file on your Web Server. If left blank, the Web browser assumes the file exists on the target machine. If the file is not found on the target machine, the download of the ActiveX library fails.
Target directory (or leave blank to assume file exists on server)	Location as a path of the file on the Web server. This can be a standard path name or a UNC path. Leave blank to assume the file exists and ensure that it will not be overwritten.

Creating an Active Server Page

[Topic groups](#) [See also](#)

If you are using the Microsoft Internet Information Server (IIS) environment to serve your Web pages, you can use Active Server Pages (ASP) to create dynamic Web- based client-server applications. Active Server Pages allow you to embed controls in a Web page that get called every time the server loads the Web page. For example, you can write a Delphi Automation server, such as one to create a bitmap or connect to a database, and this control accesses data that gets updated every time the server loads the Web page.

Active Server Pages allow Web applications to be built using ActiveX server components (Automation objects). These are server-side components which you can develop using any Delphi or most other languages including C++, Java, Visual Basic. On the client side, the ASP is a standard HTML document and can be viewed by users on any platform using any Web browser.

Prior to this technology, client applications needed to be installed on every computer that would access the server. With this ASP model, most of the client application runs on the server; only the user interface presentation usually runs on the client. This makes it easier to update clients when an application changes.

You can also use Active Server Pages in conjunction with the Microsoft Transaction Server to automate the management of COM server components. For details on MTS, see [Creating MTS Automation objects](#).

This topic shows how to create an Active Server Page using the Delphi Active Server Page wizard. With this, you can expose properties and methods of an existing application for Automation control.

Here are the steps for creating an Active Server Page object from an existing application:

- [Create an Active Server Page object](#) for the application.
- [Expose the application's properties and methods for Automation.](#)
- [Register](#) the application as an Active Server Page object.
- [Test and debug](#) the application.

For more background on the COM technologies, see [Overview of COM technologies](#).

Creating an Active Server Page object

[Topic groups](#) [See also](#)

An Active Server Page object is an Automation object that has access to the interfaces of a Web request. Like other Automation objects, it is an ObjectPascal class descending from *TAutoObject* that supports Automation protocols, exposing itself for other applications to use. You create an Active Server Page object using the Active Server Object wizard.

Before you create an Active Server Page object, create or open the project for an application containing functionality that you want to expose. The project can be either an application or ActiveX library, depending on your needs.

You can use **Server.CreateObject** in an ASP page to launch either an in-process or out-of-process server, depending on your requirements. However, you should be aware of the [drawbacks of launching an out-of-process server](#).

To display the Active Server Object wizard:

- 1 Choose File|New.
- 2 Select the tab labeled, ActiveX.
- 3 Double-click the Active Server Object icon.

In the wizard, specify the following:

CoClass Name	Specify the class whose properties and methods you want to expose to client applications. (Delphi prepends a T to this name.)
Instancing	Specify an instancing mode to indicate how your Active Server Page object is launched. Note: When your Active Server Page object is used only as an in-process server, instancing is ignored.
Threading Model	Choose the threading model to indicate how client applications can call your object's interface. This is the threading model that you commit to implementing in the Active Server Page object. Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.
Active Server Type	Choose Page Level Event Methods with IIS 3 and IIS 4. It creates an Active Server Page object that implements the methods, <i>OnStartPage</i> and <i>OnEndPage</i> . These methods are called by the Web server when the page initializes and finishes execution. Use this with IIS 3 and IIS 4. Choose Object Context with IIS 5. It uses the functionality of MTS to retrieve the correct instance data of your object.
Generate a template test script for this object	Generate a simple .ASP page which creates the Delphi object based off its ProgID.
Generate event support code	Check this box to tell the wizard to implement a separate interface for managing events of your Active Server Page object.

When you complete this procedure, a new unit is added to the current project that contains the definition for the Active Server Page object. In addition, the wizard adds a type library project and opens the type library. Now you can expose the properties and methods of the interface through the type library as described in [Exposing the application's properties and methods for Automation](#).

The Active Server Page object, like any other Automation object, implements a **dual interface**, which supports both early (compile-time) binding through the VTable and late (runtime) binding through the *IDispatch* interface.

Creating ASPs for in-process or out-of-process servers

[Topic groups](#) [See also](#)

You can use **Server.CreateObject** in an ASP page to launch either an in-process or out-of-process server, depending on your requirements. However, launching in-process servers is more common.

In-process component DLLs are faster, more secure, and can be hosted by MTS, so they are better suited for server-side use.

Because out-of-process servers are less secure, it is common for IIS to be configured to *not* allow out-of-process executables. In this case, you would receive an error similar to the following:

```
Server object error 'ASP 0196'  
Cannot launch out of process component  
/path/outofprocess_exe.asp, line 11
```

Also, out-of-process components often create individual server processes for each object instance, so they are slower than CGI applications. They do not scale as well as component DLLs that run in-process with IIS or MTS. If performance and scalability are priorities for your site, we recommend that you do not use out-of-process components.

However, intranet sites that receive moderate to low traffic might be able to use an out-of-process component without adversely affecting the site's overall performance.

Registering an application as an Active Server Page object

[Topic groups](#)

You can register the Active Server Page as an in-process or an out-of-process server. However, in-process servers are more common. For more information, see [Creating ASPs for in-process or out-of-process servers](#).

Note: When you want to remove the Active Server Page object from your system, you should first unregister it, removing its entries from the Windows registry.

Registering an in-process server

To register an in-process server (DLL or OCX),

- Choose Run|Register ActiveX Server.

To unregister an in-process server,

- Choose Run|Unregister ActiveX Server.

Registering an out-of-process server

To register an out-of-process server,

- Run the server with the /regserver command-line option. (You can set command-line options with the Run|Parameters dialog box.)

You can also register the server by running it.

To unregister an out-of-process server,

- Run the server with the /unregserver command-line option.

Testing and debugging the Active Server Page application

[Topic groups](#) [See also](#)

Debugging any in-process server such as an Active Server Page is much like debugging a DLL. You choose a host application that loads the DLL, and debug as usual. To test and debug an Active Server Page object,

- 1 Turn on debugging information using the Compiler tab on the Project|Options dialog box, if necessary. Also, turn on Integrated Debugging in the Tools|Debugger Options dialog.
- 2 Choose Run|Parameters, type the name of your Web Server in the Host Application box, and choose OK.
- 3 Choose Run|Run.
- 4 Set breakpoints in the Active Server Page.
- 5 Use the Web browser to interact with the Active Server Page.

The Active Server Page pauses when the breakpoints are reached.

Working with type libraries

[Topic groups](#) [See also](#)

This topic describes how to create and edit type libraries using Delphi's Type Library editor. Type libraries are files that include information about data types, interfaces, member functions, and object classes exposed by an ActiveX control or server. Type libraries provide a way for you to identify what types of objects and interfaces are available on your ActiveX server. For a detailed overview on why and when to use type libraries, see [Parts of a COM application](#).

By including a type library with your COM application or ActiveX library, you make information about the objects in your COM application available to other applications and programming tools.

With traditional development tools, you create type libraries by writing scripts in the Interface Definition Language (IDL) or the Object Description Language (ODL), then run that script through a compiler. With the Type Library editor, Delphi automates some of this process, easing the burden of creating your own type libraries.

If you create your COM object, ActiveX control, or Automation object using a wizard, the Type Library editor automatically generates the Pascal syntax for your existing object. Then you can easily refine your type library information using the [Type Library editor](#). As you modify the type library using the Type Library editor, changes can be automatically updated in the associated object, or you can review and veto those changes, if the [Apply Updates dialog](#) is enabled.

You can also use the Delphi Type Library Editor in the development of Common Object Request Broker Architecture (CORBA) applications. With traditional CORBA tools, you must define object interfaces separately from your application, using the CORBA Interface Definition Language (IDL). You then run a utility that generates stub-and-skeleton code from that definition. However, Delphi generates the stub, skeleton, and IDL for you automatically. You can easily edit your interface using the [Type Library editor](#) and Delphi automatically updates the appropriate source files.

A type library can contain any and all of the following:

- Information about data types, including aliases, enumerations, structures, and unions.
 - Descriptions of one or more COM elements, such as an interface, dispinterface, or CoClass.
- Each of these descriptions is commonly referred to as [type information](#).
- Descriptions of constants and methods defined in external units.
 - References to type descriptions from other type libraries.

The following topics describe

- [Using Object Pascal or IDL syntax](#)
- [Creating new type libraries](#)
- [Deploying type libraries](#)

Type Library editor

[Topic groups](#) [See also](#)

The Type Library editor is a tool that enables developers to examine and create type information for ActiveX controls and COM objects.

The main elements of the Type Library editor are

















- [Toolbar](#), which you use to add new interfaces and interface members to your type library.
- [Object list pane](#), which displays all the existing objects in the type library. When you click on an item in the object list pane, it displays pages valid for that object.
- [Status bar](#), which displays syntax errors if you try to add invalid types to your type library.
- [Pages](#), which display information about the selected object. Which pages appear here depends on the type of object selected.
- Error window, which displays errors detected while loading a type library.

Toolbar

[Topic groups](#)

The Type Library editor's toolbar located at the top of the Type Library Editor, contains buttons that you click to add new objects into your type library.

You can add the following object types from the toolbar:

<u>Icon</u>	<u>Meaning</u>
	A <u>type library</u> . This can be expanded to expose the individual type information, including objects and interfaces.
	An <u>interface</u> description.
	A <u>dispinterface</u> description.
	A <u>CoClass</u> .
	An <u>enumeration</u> .
	An <u>alias</u> .
	A <u>record</u> .
	A <u>union</u> .
	A <u>module</u> .
	A method of the <u>interface</u> , <u>dispinterface</u> , or an entry point in a <u>module</u> .
	A put by value property function.
	A put by reference property function.
	A get property function.
	A property.
	A <u>field</u> in a record or union.
	A <u>constant</u> in an enum or a module.

The icons in the left box, Interface, Dispatch, CoClass, Enum, Alias, Record, Union, and Module represent the type info objects that you can edit.

When you click a toolbar button, the icon of that information type appears at the bottom of the object list pane. You can then customize its attributes in the right pane. Depending on the type of icon you selected, different pages of information appear to the right.

When you select an object, the Type Library editor displays which members are valid for that object. These members appear on the toolbar in the second box. For example, when you select Interface, the Type Library editor displays Method and Property icons in the second box because you can add methods and properties to your interface definition. When you select Enum, the Type Library editor displays the Const member, which is the only valid member for Enum type information.

In the third box, you can choose to refresh, register, or export your type library.

Object list pane

[Topic groups](#) [See also](#)

The Object list pane displays all the elements of the current type library, beginning with its interfaces. The interfaces expand to show the properties, methods, and events specified for that interface:



The context menu for the Object list pane provides the following options:

New	Pops up a menu containing a list of objects to add to your type library. These are the same objects available from the toolbar.
Cut	Removes the selected object and puts it on the Windows clipboard.
Copy	Puts the selected object on the Windows clipboard.
Paste	Inserts an object from the Windows clipboard below the selected object.
Delete	Removes the selected object.
View Errors	Toggles the visibility of the error window.
Toolbar	Toggles the visibility of the toolbar.

Status bar

[Topic groups](#)

When editing or saving a type library, syntax, translation errors, and warnings are listed in the Status bar pane.

For example, if you specify a type that the Type Library editor does not support, you will get a syntax error. For a complete list of types supported by the Type Library editor, see [Valid types](#).

Pages of type information

[Topic groups](#) [See also](#)

When you select an object in the object list pane, pages of type information appear in the Type Library editor that are valid for the selected object. Which pages appear depends on the object selected in the object list panel as follows:

<u>Type Info object</u>	<u>Pages of type information</u>
Type library	Attributes, Uses, Flags, Text
Interface	Attributes, Flags, Text
Dispinterface	Attributes, Flags, Text
CoClass	Attributes, Implements, Flags, Text
Enum	Attributes, Text
Alias	Attributes, Text
Record	Attributes, Text
Union	Attributes, Text
Module	Attributes, Text
Method	Attributes, Parameters, Flags, Text
Property	Attributes, Parameters, Flags, Text
Const	Attributes, Flags, Text
Field	Attributes, Flags, Text

Attributes page

All type library elements have attributes pages, which allow you to define a name and other attributes specific to the element. For example, if you have an interface selected, you can specify the GUID and parent interface. If you have a field selected, you can specify its type. Subsequent sections describe the attributes for each type library element in detail.

Attributes common to all elements in the type library are those associated with help. It is strongly recommended that you use the help strings to describe the elements in your type library to make it easier for applications using the type library.

The Type Library editor supports two mechanisms for supplying help. The traditional help mechanism, where a standard windows help file has been created for the library, or where the help information is located in a separate DLL (for localization purposes).

The following help attributes can apply to all elements:

<u>Attribute</u>	<u>Meaning</u>
Help String	A short description of the element.
Help Context	The Help context ID of the element, which identifies the Help topic within the standard windows Help file for this element. Used when a standard windows help file has been specified for the 'Help File' attribute of the type library.
Help String Context	The Help context ID of the element, which identifies the Help topic within the help DLL for this element. Used when a help DLL has been specified for the 'Help String DLL' attribute of the type library.

Note: The help file must be supplied separately by the developer.

Text page

All type library elements have a text page that displays the syntax for the element. This syntax appears in an IDL subset of Microsoft Interface Definition Language, or Object Pascal. See [Using Object Pascal or IDL syntax](#) for details. Any changes you make in other pages of the element are reflected on the text page. If you add code directly in the text page, changes are reflected in the other pages of the Type Library editor.

The Type Library editor will generate syntax errors if you add identifiers that are currently not supported

by the editor; the editor currently supports only those identifiers that relate to type library support (not RPC support or constructs used by the Microsoft IDL compiler for C++ code generation or marshaling support).

Flags page

Some type library elements have flags that allow you to enable or disable certain characteristics or implied capabilities.

Type library information

[Topic groups](#)

When the type library (top-most node) is selected in the Object list pane, you can change type information for the type library itself by modifying the following pages:

- [Attributes page](#)
- [Uses page](#)
- [Flags page](#)

Attributes page for a type library

The Attributes page displays type information about the currently selected type library:

<u>Attribute</u>	<u>Meaning</u>
Name	The descriptive name of the type library without spaces or punctuation.
GUID	The globally unique 128-bit identifier of the interface.
Version	A particular version of the library in cases where multiple versions of the library exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
LCID	The locale identifier that describes the single national language used for all text strings in the type library and elements.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.
Help String DLL	The fully qualified name of the DLL used for Help, if any.
Help File	The name of the help file associated with the type library, if any.

Uses page for a type library

The Uses page lists the names and locations of any other type libraries that this type library references.

Flags page for a type library

The following flags appear on the flags page when a type library is selected. It specifies how other applications must use the server associated with this type library:

<u>Flag</u>	<u>Meaning</u>
Restricted	Prevents the library from being used by a macro programmer.
Control	Indicates that the library represents a control.
Hidden	Indicates that the library exists but should not be displayed in a user-oriented browser.

Interface pages

[Topic groups](#) [See also](#)

The interface describes the methods (and any properties expressed as 'get' and 'set' functions) for an object that must be accessed through a virtual function table (VTable). If an interface is flagged as dual, which is the default, a dispinterface is also implied and can be accessed through OLE automation.

You can modify a type library interface by

- [Changing attributes](#)
- [Changing flags](#)
- Adding, deleting, or modifying [interface members](#)

Attributes page for an interface

[Topic groups](#) [See also](#)

The Attributes page lists the following type information:

Attribute	Meaning
Name	The descriptive name of the interface.
GUID	The globally unique 128-bit identifier of the interface (optional).
Version	A particular version of the interface in cases where multiple versions of the interface exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Parent Interface	The name of the base interface for this interface. All COM interfaces must ultimately derive from <i>IUnknown</i> .
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Interface flags

[Topic groups](#) [See also](#)

The following flags are valid when an interface is selected in the main object pane.

Flag	Meaning
Hidden	Indicates that the interface exists but should not be displayed in a user-oriented browser.
Dual	Identifies an interface that exposes properties and methods through both <i>IDispatch</i> and directly through a v-table. Default setting.
Ole Automation	Indicates that an interface can only use Automation-compatible types. This flag is not allowed on dispinterfaces since they are Automation compatible by definition.
Non-extensible	Indicates that the interface is not intended to be used as a base interface for another interface.

Interface members

[Topic groups](#) [See also](#)

The Type Library editor allows you to define new or modify existing

- [Properties](#)
- [Methods](#)

You can also change property and method parameters with the [Parameters page](#).

The interface is more commonly used than the dispinterface to describe the properties and methods of an object.

Members of interfaces that need to raise exceptions should return an HRESULT and specify a return value parameter (PARAM_RETVAL) for the actual return value. Declare these methods using the **safecall** calling convention.

Interface methods

[Topic groups](#) [See also](#)

When you select the Method icon to add a new method, the Type Library editor displays the valid pages for a method: attributes, parameters, flags, and text.

Method attributes

The attributes for an interface method are as follows:

<u>Attribute</u>	<u>Meaning</u>
Name	Name of the member.
ID	Dispatch ID.
Invoke kind	Indicates whether this method or property is a function. For methods, specify function.

See [Pages of type information](#) for a description of help attributes.

Method parameters

You supply parameters for methods as described in the [Parameters page](#).

Method flags

The flags for an interface method are as follows:

<u>Flag</u>	<u>IDL Identifier</u>	<u>Meaning</u>
Replaceable	replaceable	The object supports <i>IConnectionPointWithDefault</i> .
Restricted	restricted	Prevents the property or method from being used by a programmer.
Source	source	Indicates that the member returns an object or VARIANT that is a source of events.
Bindable	bindable	Indicates that the property supports data binding.
Hidden	hidden	Indicates that the property exists but should not be displayed in a user-oriented browser.
UI Default	uidefault	Indicates that the type information member is the default member for display in the user interface.

Interface properties

[Topic groups](#) [See also](#)

Properties for interfaces are represented by the 'get' and 'set' methods used to read and write the property's underlying data. They are represented in the tree view using special icons that indicate their purpose.



A write (set, put) by value property function.



A read (get) |write (set, put)|write by reference property function.



A read (get) property function.

Note: ActiveX properties specified as Write By Reference means the property is passed as a pointer rather by value. Some applications, such a Visual Basic, use the Write By Reference, if it is present, to optimize performance. To pass the property only by reference rather than by value, use the property type, By Reference Only. To pass the property by reference as well as by value, select Read|Write|Write By Ref. To invoke this menu, go to the toolbar and select the arrow next to the property icon.

Property attributes

The attributes for an interface property are as follows:

Attribute	Meaning
Name	Name of the member.
ID	Dispatch ID.
Type	Type of property; this can be any valid type
Invoke kind	Indicates whether this property is a getter function, setter function, or setter by reference.

See [Pages of type information](#) for a description of help attributes.

Property flags

The flags that you can set for an interface property are as follows:

Flag	IDL Identifier	Meaning
Replaceable	replaceable	The object supports <i>IConnectionPointWithDefault</i> .
Restricted	restricted	Prevents the property or method from being used by a programmer.
Source	source	Indicates that the member returns an object or VARIANT that is a source of events.
Bindable	bindable	Indicates that the property supports data binding.
Request Edit	requestedit	Indicates that the property supports the OnRequestEdit notification.
Display Bindable	displaybind	Indicates a property that should be displayed to the user as bindable.
Default Bindable	defaultbind	Indicates the single, bindable property that best represents the object. Properties that have the defaultbind attribute must also have the bindable attribute. Cannot be specified on more than one property in a dispinterface.
Hidden	hidden	Indicates that the property exists but should not be displayed in a user-oriented browser.
Default Collection Element	defaultcollelem	Allows for optimization of code in Visual Basic.

UI Default	uidefault	Indicates that the type information member is the default member for display in the user interface.
Non Browsable	nonbrowsable	Indicates that the property appears in an object browser that does not show property values, but does not appear in a properties browser that does show property values.
Immediate Bindable	immediatebind	Allows individual bindable properties on a form to specify this behavior. When this bit is set, all changes will be notified. The bindable and request edit attribute bits need to be set for this new bit to have an effect.

Property and method and parameters page

[Topic groups](#) [See also](#)

The parameters page allows you to specify the parameters and return values for your functions.

For property functions, the property type is derived from either the return type or the last parameter. Changing the type on the parameters page affects the property type displayed on the attributes page. Likewise, changing the type displayed on the attributes page, affects the contents of the parameters page.

Note: Changing one property function affects any related property functions. The Type Library editor assumes that property functions are related if they have the same name and Dispatch ID.

The parameters page differs, depending on whether you are working in IDL or Object Pascal.

When you add a parameter, if you are working in Object Pascal, the Type Library editor supplies the following:

- Modifier
- Name
- Type
- Default Value

If you are working in IDL, the Type Library editor supplies the following:

- Name
- Type
- Modifier

You can change the name by typing a new name. Change the type by choosing a new value from the drop-down list. The possible values for type are those supported by the Type Library editor.

If you are working in Object Pascal, change the modifier by choosing a new value from the drop-down list. The possible values are as follows:

<u>Modifier</u>	<u>Meaning</u>
blank (the default)	Input parameter. This can be a pointer, but the value it refers to is not returned. (Same as In in IDL)
none	No information is provided for marshaling parameter values. This modifier should only be used with dispinterfaces, which are not marshaled. (same as having no flags in IDL)
out	Output parameter. This is a reference value that receives the result. (Same as Out in IDL)
optionalout	Output parameter that is optional. This must be a Variant type, and all subsequent parameters must be optional. (Same as [Out, Optional] in IDL)
var	Input/Output parameter. Combination of blank and out. (Same as [In, Out] in IDL)
optionalvar	Input/Output parameter that is optional. Combination of optional and optionalout. (Same as [In, Out, Optional] in IDL)
optional	Input parameter that is optional. This must be a variant type, and all subsequent parameters must be optional. (Same as [In, Optional] in IDL)
retval	Receives the return value. Return values must be the last parameter listed (as enforced by the Type Library editor). Parameters with this value are not displayed in user-oriented browsers. This is only applicable if the function is declared without the safecall directive. (same as [Out, RetVal] in IDL)

Use the default column to specify default parameter values. When you add a default, the Type Library editor automatically adds the appropriate flags to the type library.

To change a parameter flag (when working in IDL), double-click the Modifier field to go to the

Parameters flag dialog box. You can select from the following parameter flags:

Flag	Meaning
In	Input parameter. This can be a pointer, but the value it refers to is not returned.
Out	Output parameter. This must be a pointer to a member that will receive the result.
RetVal	Receives the return value. Return values must be an out attribute, and be the last parameter listed (as enforced by the Type Library editor). Parameters with this value are not displayed in user-oriented browsers.
LCID	Indicates that this parameter is a locale ID. Only one parameter can have this attribute, it must be flagged as [in] and its type must be long. This allows members in the VTable to receive an LCID at the time of invocation. Parameters with this value are not displayed in user-oriented browsers. By convention, LCID is the parameter preceding RetVal. Not allowed in dispinterfaces.
Optional	Specifies an optional parameter. Of course, all subsequent parameters must also be optional.
Has Default Value	This allows you to specify a default value for a typed optional parameter. Value must be a constant that can be stored in a VARIANT.
Default Value	If you choose Has Default Value, supply the default here. The value must be the same type as the optional parameter.

Note: When working in IDL, default values are specified using flags rather than in a separate column.

Also, local IDs are specified using a flag rather than using a parameter type specifier of TLCID.

You can use the Move Up and Move Down buttons to change the order of your parameters. However, the editor will not change the order if the move breaks a rule of the IDL language. For example, the Type Library editor enforces the rule that return values must always be the last parameter in the parameter list.

Dispatch type information

[Topic groups](#) [See also](#)

Interfaces are more commonly used than dispinterfaces to describe the properties and methods of an object. Dispinterfaces are only accessible through dynamic binding, while interfaces can have static binding through a vtable.

You can modify a dispatch interface (dispinterface) by

- [Changing attributes](#)
- [Changing flags](#)
- Adding, deleting, or modifying [dispinterface members](#)

Attributes page for dispatch

[Topic groups](#) [See also](#)

The following attributes apply to the dispinterface:

Attribute	Meaning
Name	The name by which the dispinterface is known in the type library. It must be a unique name within the type library.
GUID	The globally unique 128-bit identifier of the dispatch (optional).
Version	A particular version of the dispinterface in cases where multiple versions of the dispinterface exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Dispatch flags page

[Topic groups](#) [See also](#)

The flags page for Dispatch is the same as for [Interface](#).

Dispatch members

[Topic groups](#) [See also](#)

For your dispinterface, you can define

- [Methods](#)
- [Properties](#)

How you add methods and properties for dispinterfaces is the same as how you add them for interfaces.

Notice that when you create a property for a dispinterface, you can no longer specify a function kind or parameter types.

CoClass Pages

[Topic groups](#)

The CoClass describes a unique COM object which implements one or more interfaces and specifies which implemented interface is the default for the object, and optionally, which dispinterface is the default source for events.

You can modify a CoClass definition in the type library by

- [Changing attributes](#)
- [Changing the implements page](#)
- [Changing flags](#)

Attributes page for a CoClass

[Topic groups](#) [See also](#)

The attributes page that apply for the CoClass are as follows:

Attribute	Meaning
Name	The descriptive name of the CoClass.
GUID	The globally unique 128-bit identifier of the CoClass (optional).
Version	A particular version of the CoClass in cases where multiple versions of the CoClass exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

CoClass Implements page

[Topic groups](#) [See also](#)

The Implements page is used to specify what interfaces and dispinterfaces are implemented for the CoClass. For each interface, the Implements page specifies whether the following items are supported:

<u>Interface</u>	<u>Description</u>
Interface	Name of an interface that the CoClass implements.
GUID	Identifies the GUID of the member interface of the object.
Source	Indicates that the member is a source of events.
Default	Indicates that the interface or dispinterface represents the default interface. This is the interface that is returned by default when an instance of the class is created. A CoClass can have two default members at most. One represents the primary interface or dispinterface, and the other represents an optional dispinterface that serves as an event source.
Restricted	Prevents the item from being used by a programmer. A member of an interface cannot have both restricted and default attributes.
VTable	Enables an object to have two different source interfaces.

The implements page's context menu contains menu items to toggle the above options. The 'Insert interface' menu option brings up a dialog where you can choose an interface to add to the CoClass. The list includes interfaces that are defined in the current type library and interfaces defined in any type libraries that the current type library references.

CoClass flags

[Topic groups](#) [See also](#)

The following flags are valid when a CoClass is selected in the main object pane.

Flag	Meaning
Hidden	Indicates that the interface exists but should not be displayed in a user-oriented browser.
Can Create	Instance can be created with <code>CoCreateInstance</code> .
Application Object	Identifies the CoClass as an application object, which is associated with a full EXE application, and indicates that the functions and properties of the CoClass are globally available in this type library.
Licensed	Indicates that the CoClass to which it applies is licensed, and must be instantiated using <code>IClassFactory2</code> .
Predefined	The client application should automatically create a single instance of the object that has this attribute.
Control	Identifies a CoClass as an ActiveX control, from which a container site will derive additional type libraries or CoClasses.
Aggregatable	Indicates that the members of the class can be aggregated.
Replaceable	The object supports <code>ICornerPointWithDefault</code> .

Enumeration type information

[Topic groups](#)

You can add or modify an Enumeration definition in the type library by

- [Changing attributes](#)
- Adding, deleting, or modifying [enum members](#)

Attributes page for an enum

[Topic groups](#) [See also](#)

The attributes page that apply to an enum are as follows:

Attribute	Meaning
Name	The descriptive name of the enum.
GUID	The globally unique 128-bit identifier of the enum (optional).
Version	A particular version of the enum in cases where multiple versions of the enum exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

It is strongly recommended that you provide a help string for your enumerations to make their meaning clearer. The following is a sample entry of an enumeration type for a mouse button and includes a help string for each enumeration element.

```
mbLeft = 0 [helpstring 'mbLeft'];  
mbRight = 1 [helpstring 'mbRight'];  
mbMiddle = 3 [helpstring 'mbMiddle'];
```

Enumeration members

[Topic groups](#) [See also](#)

Enums are comprised of a list of constants, which must be numeric. Numeric input is usually an integer in decimal or hexadecimal format. The base value is zero by default.

To define constants for your enum, click the New Const button.

Alias type information

[Topic groups](#)

An alias creates an alias (type definition) for a type. You can use the alias to define types that you want to use in other type info such as records or unions.

You can create or modify an alias definition in the type library by

- Changing attributes

Attributes page for an alias

The attributes page for an alias contains the following

Attribute	Meaning
Name	The descriptive name by which the alias is known in the type library.
GUID	The globally unique 128-bit identifier of the interface (optional). If omitted, the alias is not uniquely specified in the system.
Version	A particular version of the alias in cases where multiple versions of the alias exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Type	Type that you want to alias.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Record type information

[Topic groups](#)

You can add or modify a record definition in the type library by

- [Changing attributes](#)
- Adding, deleting, or modifying [record members](#)

Attributes page for a record

The attributes page for a record contain the following

<u>Attribute</u>	<u>Meaning</u>
Name	The descriptive name by which the record is known in the type library.
GUID	The globally unique 128-bit identifier of the interface (optional). If omitted, the record is not uniquely specified in the system.
Version	A particular version of the record in cases where multiple versions of the record exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Record members

A record is comprised of a list of structure members or fields. A field has a

- Name
- Type

Members can be of any built-in type, or you can specify a type using alias before you define the record.

Records can be defined with an optional tag.

Union type information

[Topic groups](#)

A union is a record with only a variant part.

You can add or modify a union definition in the type library by

- [Changing attributes](#)
- Adding, deleting, or modifying [union members](#)

Attributes page for a union

The attributes page for a union contain the following

<u>Attribute</u>	<u>Meaning</u>
Name	The descriptive name by which the union is known in the type library.
GUID	The globally unique 128-bit identifier of the interface (optional). If omitted, the union is not uniquely specified in the system.
Version	A particular version of the union in cases where multiple versions of the union exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Union members

A union, like a record, is comprised of a list of structure members or fields. A field has the following parts:

- Name
- Type

Members can be of any built-in type, or you can specify a type using alias before you define the record.

Unions can be defined with an optional tag.

Module type information

[Topic groups](#)

The module defines a group of functions, typically a set of DLL entry points.

You can create or modify a module definition in the type library by

- [Changing attributes](#)
- Adding, deleting, or modifying [module members](#)

Note: Delphi does not automatically generate any declarations or implementation related to the module. The specified DLL is created by the user as a separate project.

Attributes page for a module

The attributes page for a module contains the following

<u>Attribute</u>	<u>Meaning</u>
Name	The descriptive name of the module.
GUID	The globally unique 128-bit identifier of the interface (optional). If omitted, the module is not uniquely specified in the system.
Version	A particular version of the module in cases where multiple versions of the module exist. The version is either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are short unsigned integers in the range between 0 and 65535, inclusive.
DLL name	Name of associated DLL for which these entry points apply.
Help String	A short description of the element. Used with Help Context to provide Help as a Help file.
Help Context	The Help context ID of the element, which identifies the Help topic within the Help file for this element.
Help String Context	For help DLLs, the Help context ID of the element, which identifies the Help topic within the help file for this element. Used with Help String DLL to provide Help as a separate DLL.

Module members

A module can be comprised of

- Methods
- Constants

Module methods

Module methods have the following attributes:

<u>Attribute</u>	<u>Meaning</u>
Name	The descriptive name of the module
DLL entry	Entry point into the associated DLL

You supply parameters for module methods in the same way you supply interface parameters, with the [Parameters page](#).

Module constants

To define constants for your module, you supply the following:

- Name
- Value

- Type

A module constant can be either a numeric or a string, depending on the attribute. Numeric input is usually an integer in decimal or hexadecimal format; it can also be a single char constant (such as `\0`). String input is delimited by double quotation marks (`""`), and cannot span multiple lines. The backslash is an escape character. The backslash character followed by another character, including another backslash, prevents the second character from being interpreted with any special meaning. For example, to put a backslash into text, use:

```
"Pathname: c:\\bin\\"
```

Creating new type libraries

[Topic groups](#) [See also](#)

The Type Library editor enables you to create type libraries for ActiveX controls, ActiveX servers and other COM objects.

The editor supports a subset of valid types and safe arrays in a type library

This topic describes how to:

- [Create a new type library](#)
- [Open an existing type library](#)
- [Add an interface to the type library](#)
- [Add properties and methods to the type library](#)
- [Add a CoClass to the type library](#)
- [Add an enumeration to the type library](#)
- [Saving and registering type library information](#)

Valid types

[Topic groups](#) [See also](#)

In the Type Library editor, you use different type identifiers, depending on whether you are working in IDL or Object Pascal. Specify the language you want to use in the Environment options dialog.

The following types are valid in a type library for COM development. The Automation compatible column specifies whether the type can be used by an interface that has its Automation or DisplInterface flag checked. These are the types that COM can marshal via the type library automatically.

<u>Pascal type</u>	<u>IDL type</u>	<u>variant type</u>	<u>Automation compatible</u>	<u>Description</u>
Smallint	short	VT_I2	Yes	2-byte signed integer
Integer	long	VT_I4	Yes	4-byte signed integer
Single	single	VT_R4	Yes	4-byte real
Double	double	VT_R8	Yes	8-byte real
Currency	CURRENCY	VT_CY	Yes	currency
TDateTime	DATE	VT_DATE	Yes	date
WideString	BSTR	VT_BSTR	Yes	binary string
IDispatch	IDispatch	VT_DISPATCH	Yes	pointer to IDispatch interface
SCODE	SCODE	VT_ERROR	Yes	Ole Error Code
WordBool	VARIANT_BOOL	VT_BOOL	Yes	True = -1, False = 0
OleVariant	VARIANT	VT_VARIANT	Yes	Ole Variant
IUnknown	IUnknown	VT_UNKNOWN	Yes	pointer to IUnknown interface
Shortint	byte	VT_I1	No	1 byte signed integer
Byte	unsigned char	VT_UI1	Yes	1 byte unsigned integer
Word	unsigned short	VT_UI2	No*	2 byte unsigned integer
LongWord	unsigned long	VT_UI4	No*	4 byte unsigned integer
Int64	__int64	VT_I8	No	8 byte signed real
Largeuint	uint64	VT_UI8	No	8 byte unsigned real
SYSINT	int	VT_INT	No*	system dependent integer (Win32=Integer)
SYSUINT	unsigned int	VT_UINT	No*	system dependent unsigned integer
HResult	HRESULT	VT_HRESULT	No	32 bit error code
Pointer		VT_PTR -> VT_VOID	No	untyped pointer
SafeArray	SAFEARRAY	VT_SAFEARRAY	No	OLE Safe Array
PChar	LPSTR	VT_LPSTR	No	pointer to Char
PWideChar	LPWSTR	VT_LPWSTR	No	pointer to WideChar

Note: For valid types for CORBA development, see [Defining Object Interfaces](#).

* Word, LongWord, SYSINT, and SYSUINT may be Automation-compatible with some applications.

Note: Byte (VT_UI1) is Automation-compatible, but is not allowed in a Variant or OleVariant since many Automation servers do not handle this value correctly.

Besides these types, any interfaces and types defined in the library or defined in referenced libraries can be used in a type library definition.

The Type Library editor stores type information expressed in the Interface Definition Language (IDL) syntax in the generated type library (.TLB) file in binary form.

If the parameter type is preceded by a Pointer type, the Type Library editor usually translates that type into a variable parameter. When the type library is saved, the variable parameter's associated ElemDesc's IDL flags are marked IDL_FIN or IDL_FOUT.

Often, ElemDesc IDL flags are not marked by IDL_FIN or IDL_FOUT when the type is preceded with a

Pointer. Or, in the case of dispinterfaces, IDL flags are not typically used. In these cases, you may see a comment next to the variable identifier such as `{IDL_None}` or `{IDL_In}`. These comments are used when saving a type library to correctly mark the IDL flags.

SafeArrays

[Topic groups](#) [See also](#)

COM requires that arrays be passed via a special data type known as a SafeArray. You can create and destroy SafeArrays by calling special COM functions to do so, and all elements within a SafeArray must be valid automation-compatible types. The Delphi compiler has built-in knowledge of COM SafeArrays and will automatically call the COM API to create, copy, and destroy SafeArrays.

In the Type Library editor, a *SafeArray* must specify its component type. For example, in the following code, the *SafeArray* specifies a component type of Integer:

```
procedure HighLightLines(Lines: SafeArray of Integer);
```

The component type for a *SafeArray* must be an Automation-compatible type. In the Object Pascal type library translation unit, the component type is not necessary or allowed.

Using Object Pascal or IDL syntax

[Topic groups](#) [See also](#)

By default, the Text page of the Type Library editor displays your type information using an extension of Object Pascal syntax. You can work directly in IDL instead by changing the setting in the Environment Options dialog. Choose Tools|Environment Options, and specify IDL as the Editor language on the Type Library page of the dialog.

Note: The choice of Object Pascal or IDL syntax also affects the choices available on the parameters attributes page.

Like Object Pascal applications in general, identifiers in type libraries are case insensitive. They can be up to 255 characters long, and must begin with a letter or an underscore (_).

Attribute specifications

Object Pascal has been extended to allow type libraries to include attribute specifications. Attribute specifications appear enclosed in square brackets and separated by commas. Each attribute specification consists of an attribute name followed (if appropriate) by a value.

The following table lists the attribute names and their corresponding values.

Attribute name	Example	Applies to
aggregatable	[aggregatable]	typeinfo
appobject	[appobject]	CoClass typeinfo
bindable	[bindable]	members except CoClass m
control	[control]	type library, typeinfo
custom	[custom '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}' 0]	anything
default	[default]	CoClass members
defaultbind	[defaultbind]	members except CoClass m
defaultcollection	[defaultcollection]	members except CoClass m
defaultvtbl	[defaultvtbl]	CoClass members
dispid	[dispid]	members except CoClass m
displaybind	[displaybind]	members except CoClass m
dllname	[dllname 'Helper.dll']	module typeinfo
dual	[dual]	interface typeinfo
helpfile	[helpfile 'c:\help\myhelp.hlp']	type library
helpstringdll	[helpstringdll 'c:\help\myhelp.dll']	type library
helpcontext	[helpcontext 2005]	anything except CoClass me and parameters
helpstring	[helpstring 'payroll interface']	anything except CoClass me and parameters
helpstringcontext	[helpstringcontext \$17]	anything except CoClass me and parameters
hidden	[hidden]	anything except parameters
immediatebind	[immediatebind]	members except CoClass m
lcid	[lcid \$324]	type library
licensed	[licensed]	type library, CoClass typeinfo
nonbrowsable	[nonbrowsable]	members except CoClass m
nonextensible	[nonextensible]	interface typeinfo
oleautomation	[oleautomation]	interface typeinfo
predeclid	[predeclid]	typeinfo
propget	[propget]	members except CoClass m
propput	[propput]	members except CoClass m
propputref	[propputref]	members except CoClass m

public	[public]	alias typeinfo
readonly	[readonly]	members except CoClass m
replaceable	[replaceable]	anything except CoClass me and parameters
requestedit	[requestedit]	members except CoClass m
restricted	[restricted]	anything except parameters
source	[source]	all members
uidefault	[uidefault]	members except CoClass m
usesgetlasterror	[usesgetlasterror]	members except CoClass m
uuid	[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}']	type library, typeinfo (require
vararg	[vararg]	members except CoClass m
version	[version 1.1]	type library, typeinfo

Interface syntax

The Object Pascal syntax for declaring interface type information has the form

```
interfacename = interface [(baseinterface)] [attributes]
functionlist
[property methodlist]
end;
```

For example, the following text declares an interface with two methods and one property:

```
Interface1 = interface (IDispatch)
[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}', version 1.0]
function Calculate(optional seed:Integer=0): Integer;
procedure Reset;
procedure PutRange(Range: Integer) [propput, dispid $00000005]; stdcall;
function GetRange: Integer; [propget, dispid $00000005]; stdcall;
end;
```

The corresponding syntax in IDL is

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}', version 1.0]
interface Interface1 :IDispatch
{
    long Calculate([in, optional, defaultvalue(0)] long seed);
    void Reset(void);
    [propput, id(0x00000005)] void _stdcallPutRange([in] long Value);
    [propget, id(0x00000005)] void _stdcall getRange([out, retval] long *Value);
};
```

Dispatch interface syntax

The Object Pascal syntax for declaring dispinterface type information has the form

```
dispinterfacename = dispinterface [attributes]
functionlist
[property list]
end;
```

For example, the following text declares a dispinterface with the same methods and property as the previous interface:

```
MyDispObj = dispinterface
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
version 1.0,
helpstring 'dispatch interface for MyObj']
function Calculate(seed:Integer): Integer [dispid 1];
procedure Reset [dispid 2];
property Range: Integer [dispid 3];
end;
```

The corresponding syntax in IDL is

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
version 1.0,
```

```

    helpstring "dispatch interface for MyObj"]
dispinterface Interface1
{
    methods:
    [id(1)] int Calculate([in] int seed);
    [id(2)] void Reset(void);
    properties:
    [id(3)] int Value;
};

```

CoClass syntax

The Object Pascal syntax for declaring CoClass type information has the form

```
classname = coclass (interfacename[interfaceattributes], ...); [attributes];
```

For example, the following text declares a coclass for the interface *IMyInt* and dispinterface *DMyInt*:

```

myapp = coclass (IMyInt [source], DMyInt);
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 version 1.0,
 helpstring 'A class',
 appobject]

```

The corresponding syntax in IDL is

```

[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 version 1.0,
 helpstring 'A class',
 appobject]
coclass myapp
{
    methods:
    [source] interfaceIMyInt);
    dispinterface DMyInt;
};

```

Enum syntax

The Object Pascal syntax for declaring Enum type information has the form

```
enumname = ([attributes] enumlist);
```

For example, the following text declares an enumerated type with three values:

```

location = ([uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 helpstring 'location of booth'
 Inside = 1 [helpstring 'Inside the pavillion'];
 Outside = 2 [helpstring 'Outside the pavillion'];
 Offsite = 3 [helpstring 'Not near the pavillion'];);

```

The corresponding syntax in IDL is

```

[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 helpstring 'location of booth']
typedef enum
{
    [helpstring 'Inside the pavillion'] Inside = 1,
    [helpstring 'Outside the pavillion'] Outside = 2,
    [helpstring 'Not near the pavillion'] Offsite = 3
} location;

```

Alias syntax

The Object Pascal syntax for declaring Alias type information has the form

```
aliasname = basetype[attributes];
```

For example, the following text declares DWORD as an alias for integer:

```
DWORD = Integer [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'];
```

The corresponding syntax in IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'] typedef long DWORD;
```

Record syntax

The Object Pascal syntax for declaring Record type information has the form

```
recordname = record [attributes] fieldlist end;
```

For example, the following text declares a record:

```
Tasks = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',  
               helpstring 'Task description']  
  ID: Integer;  
  StartDate: TDate;  
  EndDate: TDate;  
  Ownername: WideString;  
  Subtasks: safearray of Integer;  
end;
```

The corresponding syntax in IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',  
  helpstring 'Task description']  
typedef struct  
{  
  long ID;  
  DATE StartDate;  
  DATE EndDate;  
  BSTR Ownername;  
  SAFEARRAY (int) Subtasks;  
} Tasks;
```

Union syntax

The Object Pascal syntax for declaring Union type information has the form

```
unionname = record [attributes]  
case Integer of  
  0: field1;  
  1: field2;  
  ...  
end;
```

For example, the following text declares a union:

```
MyUnion = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',  
                helpstring 'item description']  
case Integer of  
  0: (Name: WideString);  
  1: (ID: Integer);  
  3: (Value: Double);  
end;
```

The corresponding syntax in IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',  
  helpstring 'item description']  
typedef union  
{  
  BSTR Name;  
  long ID;  
  double Value;  
} MyUnion;
```

Module syntax

The Object Pascal syntax for declaring Module type information has the form

```
modulename = module constants entrypoints end;
```

For example, the following text declares the type information for a module:

```
MyModule = module [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',  
                 dllname 'circle.dll']  
  PI: Double = 3.14159;  
  function area(radius: Double): Double [ entry 1 ]; stdcall;
```

```
function circumference(radius: Double): Double [ entry 2 ]; stdcall;  
end;
```

The corresponding syntax in IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',  
  dllname("circle.dll")]  
module MyModule  
{  
  double PI = 3.14159;  
  [entry(1)] double _stdcall area([in] double radius);  
  [entry(2)] double _stdcall circumference([in] double radius);  
};
```

Creating a new type library

[Topic groups](#) [See also](#)

You may want to create a type library that is independent of an ActiveX control, for example, if you want to define a type library for an ActiveX control that is not yet implemented.

To create a new type library,

- 1 Choose File|New to open the New Items dialog box.
- 2 Choose the ActiveX page which opens the New page.
- 3 Select the Type Library icon.
- 4 Choose OK.
The Type Library editor opens with a prompt to enter a name for the type library.
- 5 Enter a name for the type library.

Opening an existing type library

[Topic groups](#) [See also](#)

When you use the wizards to create an ActiveX control, Automation object, ActiveForm, COM object, MTS object, Remote Data Module, or MTS Data Module, a type library is automatically created with an implementation unit.

To open an existing type library independent of a project,

- 1 Choose File|Open to open the Open dialog box.
- 2 In File Type, choose type library extension for a list of available type libraries.
- 3 Select the desired type library.
- 4 Choose Open.

Or, to open a type library associated with the current project,

- 1 Choose View|Type Library.

Now, you can add interfaces, CoClasses, and other elements of the type library such as enumerations, properties, and methods.

Note: Changes you make to any type library information with the Type Library editor can be automatically reflected in the associated ActiveX control. If you would prefer to review the changes beforehand, be sure that the [Apply Updates](#) dialog is on. It is on by default and can be changed in the setting, "Display updates before refreshing," on the Tools|Environment Options|Type Library page.

Adding an interface to the type library

[Topic groups](#) [See also](#)

To add an interface,

- 1 On the toolbar, click on the interface icon.
An interface is added to the object list pane prompting you to add a name.
- 2 Type a name for the interface in the interface.

The new interface contains default attributes that you can modify as needed. You can add properties (represented by getter/setter functions and methods to suit the purpose of the interface.

Adding properties and methods to the type library

[Topic groups](#) [See also](#)

To add members to an interface or dispinterface,

- 1 Select the interface, and choose either a property or method icon from the toolbar.
An interface member is added to the object list pane prompting you to add a name.
- 2 Type a name for the member.

The new member contains default attributes in its attributes page that you can modify to suit the member.

You can add properties and methods by typing directly into the text page using the Pascal syntax. For example, you can type the following property declarations into the text page of an interface:

```
property AutoSelect: WordBool; dispid 1;  
property AutoSize: WordBool; dispid 2;  
property BorderStyle: BorderStyle; dispid 3;
```

After you have added members to an interface member page, the members appear as separate items in the object list pane each with its own attributes page where you can modify each new member's attributes.

If you have the [Apply Updates](#) dialog enabled, the Type Library editor will notify you before updating the sources when you save the type library. and will warn you of potential problems. For example, if you rename an event by mistake, you will get a warning in your source file that looks like this:

```
Because of the presence of instance variables in your implementation file,  
Delphi was not able to update the file to reflect the change in your event  
interface name. As Delphi has updated the type library for you, however, you  
must update the implementation file by hand.
```

You will also get a TODO comment in your source file immediately above it.

Note: If you ignore this warning and TODO comment, the code will not compile.

Adding a CoClass to the type library

[Topic groups](#) [See also](#)

To add a CoClass to a type library,

- 1 On the toolbar, click on the CoClass icon.
A CoClass is added to the object list pane prompting you to add a name.
- 2 Type a name for the class.
The new class contains default attributes in its attributes page that you can modify to suit the class.
To add members,
- 3 Right-click in the text page for the class to display a list of interfaces from which you can choose.
The list includes interfaces that are defined in the current type library and defined in any type libraries that the current type library references.
- 4 Double-click on an interface you want the class to implement.
The interface is added to the page with with its GUID and other attributes.

Adding an enumeration to the type library

[Topic groups](#) [See also](#)

To add enumerations to a type library,

- 1 On the toolbar, click on the enum icon.

An enum type is added to the object list pane prompting you to add a name.

- 2 Type a name for the member.

The new enum is empty and contains default attributes in its attributes page for you to modify.

Add values to the enum by clicking on the New Const button. Then, select each enumerated value and assign its attributes using the attributes page.

Saving and registering type library information

[Topic groups](#) [See also](#)

After modifying your type library, you'll want to save and register the type library information. If the type library was created with one of the ActiveX server project types or objects, saving the type library automatically updates the binary type library, the Object Pascal code representing its contents, and the implementation code maintained for it.

The Type Library editor stores type library information in two formats:

- As an OLE compound document file, called *project.TLB*.
- As a Delphi unit.

This unit is the compilation of the declarations of the elements that are defined in the type library in Object Pascal terms. Delphi uses this unit to bind the type library as a resource in your .OCX or .EXE file. Make changes to the type library within the Type Library editor, as the editor generates these files each time you save the Type Library.

Note: The type library is stored as a separate binary (.TLB) file, but is also linked into the server (.EXE, DLL, or .OCX).

Note: When using the Type Library editor for CORBA interfaces, this unit defines the stub and skeleton objects required by the CORBA application.

The Type Library editor gives you options for storing your type library information, which way you choose depends on what stage you are at in implementing the type library:

- Save, to save both the .TLB and Delphi unit to disk. See [Saving a type library](#).
- Refresh, to update the Delphi type library units in memory only. See [Refreshing the type library](#).
- Register, to add an entry for the type library in your system's Windows registry. This is done automatically when the server with which the .TLB is associated is itself registered. See [Registering the type library](#).
- Export, to save a .IDL file that contains the type and interface definitions in IDL syntax. See [Exporting an IDL file](#).

All the above methods perform syntax checking. When you refresh, register, or save the type library, Delphi automatically updates the source file of the associated object. Optionally, you can review the changes before they are committed, if you have the Type Library editor option, [Apply Updates](#) on.

Apply Updates dialog

[Topic groups](#) [See also](#)

The Apply Updates dialog appears when you refresh, register, or save the type library if you have selected “Display updates before refreshing” in the Tools|Environment Options|Type Library page (which is on by default).

Without this option, the Type Library editor automatically updates the sources of the associated object when you make changes in the editor. With this option, you have a chance to veto the proposed changes when you attempt to refresh, save, or register the type library.

The Apply Updates dialog will warn you about potential errors, and will insert TODO comments in your source file. For example, if you rename an event by mistake, you will get a warning in your source file that looks like this:

```
Because of the presence of instance variables in your implementation file,  
Delphi was not able to update the file to reflect the change in your event  
interface name. As Delphi has updated the type library for you, however, you  
must update the implementation file by hand.
```

You will also get a TODO comment in your source file immediately above it.

Note: If you ignore this warning and TODO comment, the code will not compile.

Saving a type library

[Topic groups](#) [See also](#)

Saving a type library

- Performs a syntax and validity check.
- Saves information out to a .TLB file.
- Saves information out to a Delphi unit.
- Notifies the IDE's module manager to update the implementation, if the type library is associated with an ActiveForm, ActiveX control, or Automation object.

To save the type library, choose File|Save from the Delphi main menu.

Refreshing the type library

[Topic groups](#) [See also](#)

Refreshing the type library

- Performs a syntax check.
- Regenerates the Delphi type library units in memory only. It does not save the unit to disk.
- Notifies the module manager to update the implementation if the type library is associated with an ActiveForm, ActiveX control, or Automation object.

To refresh the type library choose the Refresh icon on the Type Library editor toolbar.

Note: If you have renamed or deleted items from the type library, refreshing the implementation may create duplicate entries. In this case, you must move your code to the correct entry and delete any duplicates.

Registering the type library

[Topic groups](#) [See also](#)

Registering the type library,

- Performs a syntax check
- Adds an entry to the Windows Registry for the type library

To register the type library, choose the Register icon on the Type Library editor toolbar.

Exporting an IDL file

[Topic groups](#) [See also](#)

Exporting the type library,

- Performs a syntax check.
- Creates an IDL file that contains the type information declarations. This file can describe the type information in either CORBA IDL or Microsoft IDL.

To export the type library, choose the Export icon on the Type Library editor toolbar.

Deploying type libraries

[Topic groups](#) [See also](#)

By default, when you have a type library that was created as part of an ActiveX server project, the type library is automatically linked into the .DLL, .OCX, or EXE as a resource.

You can, however, deploy your application with the type library as a separate .TLB, as Delphi maintains the type library, if you prefer.

Historically, type libraries for Automation applications were stored as a separate file with the .TLB extension. Now, typical Automation applications compile the type libraries into the .OCX or .EXE file directly. The operating system expects the type library to be the first resource in the executable (.DLL, .OCX, or .EXE) file.

When you make type libraries other than the primary project type library available to application developers, the type libraries can be in any of the following forms:

- A resource. This resource should have the type TYPELIB and an integer ID. If you choose to build type libraries with a resource compiler, it must be declared in the resource (.RC) file as follows:

```
1 typelib mylib1.tlb  
2 typelib mylib2.tlb
```

There can be multiple type library resources in an ActiveX library. Application developers use the resource compiler to add the .TLB file to their own ActiveX library.

- Stand-alone binary files. The .TLB file output by the Type Library editor is a binary file.

Creating MTS objects

[Topic groups](#) [See also](#)

MTS is a robust runtime environment that provides transaction services, security, and resource pooling for distributed COM applications.

Delphi provides an MTS Object wizard that creates an MTS object so that you can create server components that can take advantage of the benefits of the MTS environment. MTS provides many underlying services to make creating COM clients and servers, particularly remote servers, easier to implement.

MTS components provide a number of low-level services, such as

- [Managing system resources](#), including processes, threads, and database connections so that your server application can handle many simultaneous users
- Automatically [initiating and controlling transactions](#) so that your application is reliable
- [Creating, executing, and deleting server components](#) when needed
- Providing [role-based security](#) so that only authorized users can access your application

By providing these underlying services, MTS allows you to concentrate on developing the specifics for your particular distributed application. With MTS, you implement your business logic into [MTS objects](#), or in [MTS remote data modules](#). Upon building the components into libraries (DLLs), the DLLs are installed in the MTS runtime environment.

With Delphi, MTS clients can be stand-alone applications or ActiveForms. Any COM server can run within the MTS runtime environment.

This is an overview of the Microsoft Transaction Server (MTS) technology and how you can use it to write applications based on MTS objects. Delphi also provides support for an MTS remote data module.

Microsoft Transaction Server (MTS) components

[Topic groups](#) [See also](#)

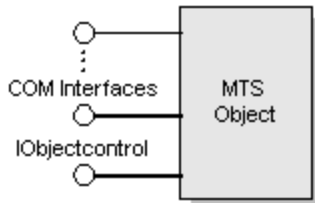
MTS components are COM in-process server components contained in dynamic-link libraries (DLLs). They are distinguished from other COM components in that they execute in the MTS runtime environment. You can create and implement these components with Delphi or any ActiveX-compatible development tool.

Note: In MTS terms, a component represents the code that implements a COM object. For example, MTS components are implemented as classes in Delphi. MTS use of the term component interferes with Delphi traditional use of this term. We use component to refer to a class or object descending from the specific class, *TComponent*. However, to remain consistent with MTS terminology, we will use MTS component when talking specifically about MTS classes. In both MTS and Delphi, we use the term object to refer to an instance of an MTS component.

Typically, MTS server objects are small, and are used for discrete business functions. For example, MTS components can implement an application's business rules, providing views and transformations of the application state. Consider, for example, the case of a physician's medical application. Medical records stored in various databases represent the persistent state of the medical application, such as a patient's health history. MTS components update that state to reflect such changes as new patients, blood test results, and X-ray files.

As shown in the following figure, an MTS object can be viewed as any other COM object. In addition to supporting any number of COM interfaces, it also supports MTS interfaces. Just as *IUnknown* is the interface common to all COM objects, *IObjectControl* is common to all MTS objects. *IObjectControl* contains methods to activate and deactivate the MTS object and to handle resources such as database connections.

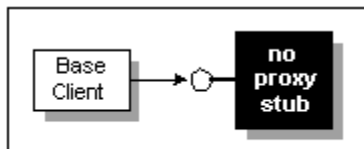
MTS object interface



A client of a server within the MTS environment is called a **base client**. From a base client's perspective, a COM object within the MTS environment looks like any other COM object. The MTS object is installed as a DLL into the MTS executive. By running with the MTS EXE, MTS objects benefit from the MTS runtime environment features such as resource pooling and transaction support.

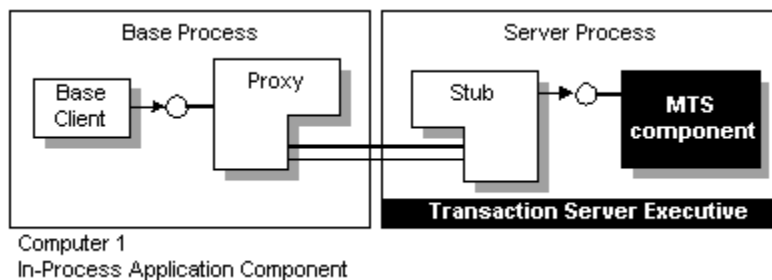
The MTS executive (.EXE) can be running in the same process as the base client as shown in the following figure.

MTS In-process component



In-Process Application Component

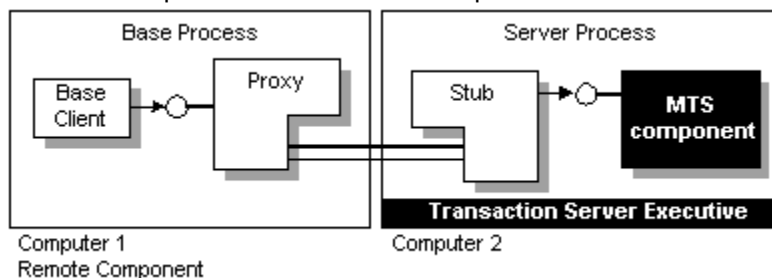
The MTS component can be installed in a remote server process within the same machine as shown in the following figure. The base client talks to a proxy which marshals the client's request to the MTS component's stub, which, in turn, accesses the MTS component through its interface.



An MTS component in an out-of-process server

The MTS component can be installed in a remote server process on a separate computer as shown in the following figure. Just as in any other remote server process, the client and remote server communicate across machines using DCOM.

An MTS component in a remote server process



Connection information is maintained in the MTS proxy. The connection between the MTS client and proxy remains open as long as the client requires a connection to the server, so it appears to the client that it has continued access to the server. In reality though, the MTS stub may deactivate and reactivate the object, conserving resources so that other clients may use the connection. For details on activating and deactivating, see [Managing resources with just-in-time activation and resource pooling](#).

Requirements for an MTS component

In addition to the COM requirements, MTS requires that the component be a dynamic-link library (DLL). Components that are implemented as executable files (.EXE files) cannot execute in the MTS runtime environment.

In addition, an MTS component must meet the following requirements:

- When using the MTS Object wizard, the component must have a standard class factory, which is automatically supplied by Delphi.
- The component must expose its class object by exporting the standard *DllGetClassObject* method.
- All component interfaces and coclasses must be described by a type library, which is provided by the MTS Object wizard. You can add methods and properties to the type library by using the Type Library editor. The information in the type library is used by the MTS Explorer to extract information about the installed components during runtime.
- The component must only export interfaces that use standard COM marshaling, which is automatically supplied by the MTS Object wizard.
- Delphi's support of MTS does not allow manual marshaling for custom interfaces. All interfaces must be implemented as dual interfaces that use COM's automatic marshaling support.
- The component must export the *DllRegisterServer* function and perform self-registration of its CLSID, ProgID, interfaces, and type library in this routine. This is provided by the MTS Object wizard.
- A component running in the MTS process space cannot aggregate with components not running in MTS.

Managing resources with just-in-time activation and resource pooling

[Topic groups](#)

MTS manages resources by providing

- [Just-in-time activation](#)
- [Resource pooling](#)
- [Object pooling](#)

Just-in-time activation

The ability for an object to be deactivated and reactivated while clients hold references to it is called **just-in-time activation**. From the client's perspective, only a single instance of the object exists from the time the client creates it to the time it is finally released. Actually, it is possible that the object has been deactivated and reactivated many times. By having objects deactivated, clients can hold references to the object for an extended time without affecting system resources. When an object becomes deactivated, MTS releases all the object's resources, for example, its database connection.

When a COM object is created as part of the MTS environment, a corresponding context object is also created. This context object exists for the entire lifetime of its MTS object, across one or more reactivation cycles. MTS uses the object context to keep track of the object during deactivation. This context object, accessed by the *IObjectContext* interface, coordinates transactions. A COM object is created in a deactivated state and becomes active upon receiving a client request.

An MTS object is deactivated when any of the following occurs:

- **The object requests deactivation with *SetComplete* or *SetAbort*:** An object calls the *IObjectContext* *SetComplete* method when it has successfully completed its work and it does not need to save the internal object state for the next call from the client. An object calls *SetAbort* to indicate that it cannot successfully complete its work and its object state does not need to be saved. That is, the object's state rolls back to the state prior to the transaction. Often, objects can be designed to be **stateless**, which means that objects deactivate upon return from every method.
- **A transaction is committed or aborted:** When an object's transaction is committed or aborted, the object is deactivated. Of these deactivated objects, the only ones that can continue to exist are the ones that have references from clients outside the transaction. Subsequent calls to these objects reactivate them and cause them to execute in the next transaction.
- **The last client releases the object:** Of course, when a client releases the object, the object is deactivated, and the object context is also released.

Resource pooling

Since MTS frees up idle system resources during a deactivation, the freed resources are available to other server objects. That is, a database connection that is no longer used by a server object can be reused by another client. This is called **resource pooling**.

Opening and closing connections to a database can be time-consuming. MTS uses resource dispensers to provide a way to reuse existing database connections rather than create new ones. A resource dispenser caches resources such as connections to a database, so that components within a package can share resources. For example, if you have a database lookup and a database update component running in a customer maintenance application, you would package those components together so that they can share database connections.

In the Delphi environment, the resource dispenser is the Borland Database Engine (BDE).

When developing MTS applications, you are responsible for [releasing resources](#).

Releasing resources

You are responsible for releasing resources of an object. Typically, you do this by calling *SetComplete* and *SetAbort* methods after servicing each client request. These methods release the resources

allocated by the MTS resource dispenser.

At this same time, you must release references to all other resources, including references to other objects (including MTS objects and context objects) and memory held by any instances of the component, such as using **free** in ObjectPascal.

The only time you would not include these calls is if you want to maintain state between client calls. For details, see [Stateful and stateless objects](#).

Object pooling

Just as MTS is designed to pool resources, it is also designed to pool objects. After MTS calls the deactivate method it calls the *CanBePooled* method, which indicates that the object can be pooled for reuse. If *CanBePooled* is set to TRUE, rather than destroying the object upon deactivation, MTS moves the object to the object pool. Objects within the object pool are available for immediate use to any other client requesting this object. Only when the object pool is empty does MTS create a new object instance.

Objects that return FALSE or that do not support the *IObjectControl* interface are destroyed.

Note: Object pooling and recycling is not available in this version of MTS. MTS calls *CanBePooled* as described, but no pooling takes place. This is provided for forward-compatibility to allow developers to use *CanBePooled* in their applications now so these applications can handle pooling when it becomes available. Currently, Delphi initializes *CanBePooled* to FALSE since object pooling is not yet available in MTS.

Accessing the object context

As with any COM object, a COM object using MTS must be created before it is used. COM clients create an object by calling the COM library function, *CoCreateInstance*.

Each COM object running in the MTS environment, must have a corresponding context object. This context object is implemented automatically by MTS and is used to manage the MTS component and coordinate transactions. The context object's interface is *IObjectContext*. To access most methods of the object context, you can use the *ObjectContext* property of the *TMtsAutoObject* object. For example, you can use the *ObjectContext* property as follows:

```
if ObjectContext.IsCallerInRole ('Manager') ...
```

Another way to access the Object context is to use methods in the *TMtsAutoObject* object:

```
if IsCallerInRole ('Manager') ...
```

You can use either of the above methods. However, there is a slight advantage of using the *TMtsAutoObject* methods rather than referencing the *ObjectContext* property when you are testing your application. For a discussion of the differences, see [Debugging and testing MTS objects](#).

MTS transaction support

[Topic groups](#) [See also](#)

The transaction support provided by MTS allows you to group actions into transactions. For example, in a medical records application, if you had a Transfer component to transfer records from one physician to another, you could have your Add and Delete methods in the same transaction. That way, either the entire Transfer works or it can be rolled back to its previous state. Transactions simplify error recovery for applications that must access *multiple* databases.

MTS transactions ensure that

- All updates in a single transaction are either committed or get aborted and rolled back to their previous state. This is referred to as **atomicity**.
- A transaction is a correct transformation of the system state, preserving the state invariants. This is referred to as **consistency**.
- Concurrent transactions do not see each other's partial and uncommitted results, which might create inconsistencies in the application state. This is referred to as **isolation**. Resource managers use transaction-based synchronization protocols to isolate the uncommitted work of active transactions.
- Committed updates to managed resources (such as database records) survive failures, including communication failures, process failures, and server system failures. This is referred to as **durability**. Transactional logging allows you to recover the durable state after disk media failures.

When you declare that an MTS component is part of a transaction, MTS associates transactions with the component's objects. When an object's method is executed, the services that resource managers and resource dispensers perform on its behalf execute under a transaction. Work from multiple objects can be composed into a single transaction.

This topic covers:

- [Transaction attributes](#)
- [Object context holds the transaction attribute](#)
- [Stateful and stateless objects](#)
- [Enabling multiple objects to support transactions](#)
- [MTS or client-controlled transactions](#)
- [Advantages of transactions](#)
- [Transaction timeout](#)

Transaction attributes

Every MTS component has a transaction attribute that is recorded in the MTS catalog. The MTS catalog maintains configuration information for components, packages, and roles. You administer the catalog using the MTS Explorer as described in the [MTS Explorer](#) topic.

Each transaction attribute can be set to these settings:

Requires a transaction	MTS objects must execute <i>within the scope of a transaction</i> . When a new object is created, its object context inherits the transaction from the context of the client. If the client does not have a transaction context, MTS automatically creates a new transaction context for the object.
Requires a new transaction	MTS objects must execute <i>within their own transactions</i> . When a new object is created, MTS automatically creates a new transaction for the object, regardless of whether its client has a transaction. An object never runs inside the scope of its client's transaction. Instead, the system always creates independent transactions for the new objects.
Supports transactions	MTS objects can execute <i>within the scope of their client's transactions</i> . When a new object is created, its object

Does not support transactions

context inherits the transaction from the context of the client. This enables multiple objects to be composed in a single transaction. If the client does not have a transaction, the new context is also created without one.

MTS objects *do not run within the scope of transactions*. When a new object is created, its object context is created without a transaction, regardless of whether the client has a transaction. Use this for COM objects designed prior to MTS support.

Object context holds transaction attribute

An object's associated context object indicates whether the object is executing within a transaction and, if so, the identity of the transaction.

Resource dispensers can use the context object to provide transaction-based services to the MTS object. For example, when an object executing within a transaction allocates a database connection by using the BDE resource dispenser, the connection is automatically enlisted on the transaction. All database updates using this connection become part of the transaction, and are either committed or aborted. For more information, see [Enlisting Resources in Transactions](#) in the MTS documentation.

Stateful and stateless objects

Like any COM object, MTS objects can maintain internal state across multiple interactions with a client. Such an object is said to be **stateful**. MTS objects can also be **stateless**, which means the object does not hold any intermediate state while waiting for the next call from a client.

When a transaction is committed or aborted, all objects that are involved in the transaction are deactivated, causing them to lose any state they acquired during the course of the transaction. This helps ensure transaction isolation and database consistency; it also frees server resources for use in other transactions. Completing a transaction enables MTS to deactivate an object and reclaim its resources. See [Enabling multiple objects to support transactions](#) for information on how to control when MTS releases your object state.

Maintaining state on an object requires the object to remain activated, holding potentially valuable resources such as database connections.

Note: Stateless objects are more efficient and, therefore, they are recommended.

Enabling multiple objects to support transactions

You use *IObjectContext* methods as shown in the following table to enable an MTS object to participate in determining how a transaction completes. These methods, together with the component's transaction attribute, allow you to enlist one or more objects into a single transaction.

<u>Method</u>	<u>Description</u>
SetComplete	Indicates that the object has successfully completed its work for the transaction. The object is deactivated upon return from the method that first entered the context. MTS reactivates the object on the next call that requires object execution.
SetAbort	Indicates that the object's work can never be committed. The object is deactivated upon return from the method that first entered the context. MTS reactivates the object on the next call that requires object execution.
EnableCommit	Indicates that the object's work is not necessarily done, but that its transactional updates can be committed in their current form. Use this to retain state across multiple calls

from a client. When an object calls `EnableCommit`, it allows the transaction in which it is participating to be committed, but it maintains its internal state across calls from its clients until it calls `SetComplete` or `SetAbort` or until the transaction completes.

`EnableCommit` is the default state when an object is activated. This is why an object should *always call `SetComplete` or `SetAbort` before returning from a method*, unless you want the object to maintain its internal state for the next call from a client.

DisableCommit

Indicates that the object's work is inconsistent and that it cannot complete its work until it receives further method invocations from the client. Call this before returning control to the client to maintain state across multiple client calls.

This prevents the MTS runtime environment from deactivating the object and reclaiming its resources on return from a method call. Once an object has called `DisableCommit`, if a client attempts to commit the transaction before the object has called `EnableCommit` or `SetComplete`, the transaction will abort.

You may use this, for example, to change the default state when an object is activated.

MTS or client-controlled transactions

Transactions can either be controlled directly by the client, or automatically by the MTS runtime environment.

Clients can have direct control over transactions by using a transaction context object (using the *`ITransactionContext`* interface). However, MTS is designed to simplify client development by taking care of transaction management automatically.

MTS components can be declared so that their objects always execute within a transaction, regardless of how the objects are created. This way, objects do not need to include any logic to handle the special case where an object is created by a client not using transactions. This feature also reduces the burden on client applications. Clients do not need to initiate a transaction simply because the component that they are using requires it.

With MTS transactions, you can implement the business logic of your application in your server objects. The server objects can enforce the rules so the client does not need to know about the rules. For example, in a physicians' medical application, an X-ray technician client can add and view X-rays in any medical record. It does not need to know that the application does not allow the X-ray technician to add or view any other type of medical record. That logic is in other server objects within the application.

Advantage of transactions

Allowing a component to either live within its own transaction or be part of a larger group of components that belong to a single transaction is a major advantage of the MTS runtime environment. It allows a component to be used in various ways, so that application developers can reuse application code in different applications without rewriting the application logic. In fact, developers can determine how components are used in transactions when packaging the component. They can change the transaction behavior simply by adding a component to a different package. For details about packaging components, see [Installing MTS objects](#) into an MTS package.

Transaction timeout

The transaction timeout sets how long (in seconds) a transaction can remain active. Transactions that

are still alive after the timeout are automatically aborted by the system. By default, the timeout value is 60 seconds. You can disable transaction timeouts by specifying a value of 0, which is useful when debugging MTS objects.

To set the timeout value on your computer,

- 1 In the MTS Explorer, select Computer, My Computer.
By default, My Computer corresponds to the local computer on which MTS is installed.
- 2 Right-click and choose Properties and then choose the Options tab.
The Options tab is used to set the computer's transaction timeout property.
- 3 Change the timeout value to 0 to disable transaction timeouts.
- 4 Click OK to save the setting and return to the MTS Explorer.

Role-based security

[Topic groups](#)

MTS currently provides role-based security where you assign a role to a logical group of users. For example, a medical information application might define roles for Physician, X-ray technician, and Patient.

You define authorization for each component and component interface by assigning roles. For example, in the physicians' medical application, only the Physician may be authorized to view all medical records; the X-ray Technician may view only X-rays; and Patients may view only their own medical record.

Typically, you define roles during application development and assign roles for each package of components. These roles are then assigned to specific users when the application is deployed. Administrators can configure the roles using the MTS Explorer.

You can also set roles programmatically using the *ObjectContext* property *TMtsAutoObject*. For example,

```
if ObjectContext.IsCallerInRole ('Manager') ...
```

Another way to access the object context is to use methods of the MTS *TMtsAutoObject* object:

```
if IsCallerInRole ('Manager') ...
```

Note: For applications that require stronger security, context objects implement the *ISecurityProperty* interface, whose methods allow retrieval of the Window's security identifier (SID) for the direct caller and creator of the object, as well as the SID for the clients which are using the object.

Resource dispensers

[Topic groups](#)

A resource dispenser manages the nondurable shared state on behalf of the application components within a process. Resource dispensers are similar to resource managers such as the SQL Server, but without the guarantee of durability. Delphi provides two resource dispensers:

- [BDE resource dispenser](#)
- [Shared Property Manager](#)

BDE resource dispenser

The BDE resource dispenser manages pools of database connections for MTS components that use the standard database interfaces, allocating connections to objects quickly and efficiently. For remote MTS data modules, connections are automatically enlisted on an object's transactions, and the resource dispenser can automatically reclaim and reuse connections.

Shared property manager

The Shared Property Manager is a resource dispenser that you can use to share state among multiple objects within a server process. For example, you can use it to maintain the shared state for a multiuser game.

By using the Shared Property Manager, you avoid having to add a lot of code to your application; MTS provides the support for you. That is, the Shared Property Manager protects object state by implementing locks and semaphores to protect shared properties from simultaneous access. The Shared Property Manager eliminates name collisions by providing **shared property groups**, which establish unique name spaces for the shared properties they contain.

To use the Shared Property Manager resource, you first use the *CreateSharedPropertyGroup* helper function to create a shared property group. Then you can write all the properties to that group and read all the properties from that group. By using a shared property group, the state information is saved across all deactivations of an MTS object. In addition, state information can be shared among all MTS objects installed in the same package. You can install MTS components into a package as described in [Installing MTS objects into an MTS package](#).

The following [example](#) shows how to add code to support the Shared Property Manager in an MTS object. After the example are [tips](#) for you to consider when designing your MTS application for sharing properties.

Example: Sharing properties among MTS object instances

The following example creates a property group called MyGroup to contain the properties to be shared among objects and object instances. In this example, we have a Counter property that is shared. It uses the *CreateSharedPropertyGroup* helper function to create the property group manager and property group, and then uses the *CreateProperty* method of the Group object to create a property called Counter.

To get the value of a property, you use the *PropertyByName* method of the Group object as shown below. You can also use the *PropertyByPosition* method.

```
unit Unit1;
interface
uses
  MtsObj, Mtx, ComObj, Project2_TLB;
type
  Tfoobar = class(TMtsAutoObject, Ifoobar)
  private
    Group: ISharedPropertyGroup;
  protected
    procedure OnActivate; override;
```

```

        procedure OnDeactivate; override;
        procedure IncCounter;
    end;
implementation
uses ComServ;
{ Tfoobar }
procedure Tfoobar.OnActivate;
var
    Exists: WordBool;
    Counter: ISharedProperty;
begin
    Group := CreateSharedPropertyGroup('MyGroup');
    Counter := Group.CreateProperty('Counter', Exists);
end;
procedure Tfoobar.IncCounter;
var
    Counter: ISharedProperty;
begin
    Counter := Group.PropertyByName['Counter'];
    Counter.Value := Counter.Value + 1;
end;
procedure Tfoobar.OnDeactivate;
begin
    Group := nil;
end;
initialization
    TAutoObjectFactory.Create(ComServer, Tfoobar, Class_foobar, ciMultiInstance,
tmApartment);
end.

```

Tips for using the Shared Property Manager

For objects to share state, they all must be running in the same server process.

You can only use shared properties to share between objects running in the same process. If you want instances of different components to share properties, you must install the components in the same MTS package. Because there is a risk that administrators will move components from one package to another, it's safest to limit the use of a shared property group to instances of components that are defined in the same DLL.

Components sharing properties must have the same activation attribute. If two components in the same package have different activation attributes, they generally won't be able to share properties. For example, if one component is configured to run in a client's process and the other is configured to run in a server process, their objects will usually run in different processes, even though they're in the same package.

Base clients and MTS components

Topic groups

It's important to understand the difference between clients and objects in the MTS runtime environment. Clients, or **base clients** in MTS terms, are not running under MTS. Base clients are the primary consumers of MTS objects. Typically, they provide the application's user interface or map the end-user's requests to the business functions defined in the MTS server objects. Alternatively, clients do not have the benefit of underlying MTS features. Clients do not get transaction support nor can they rely on resource dispensers.

The following table contrasts MTS components with base client applications.

MTS components

MTS components are contained in COM dynamic-link libraries (DLLs); MTS loads DLLs into processes on demand.

MTS manages server processes that host MTS components.

MTS creates and manages the threads used by components.

Every MTS object has an associated context object. MTS automatically creates, manages, and releases context objects.

MTS objects can use resource dispensers. Resource dispensers have access to the context object, allowing acquired resources to be automatically associated with the context.

Base clients

Base clients can be written as executable files (EXE) or dynamic-link libraries (DLL). MTS is not involved in their initiation or loading.

MTS does not manage base client processes.

MTS does not create or manage the threads used by base client applications.

Base clients do not have implicit context objects. They can use transaction context objects, but they must explicitly create, manage, and release them.

Base clients cannot use resource dispensers.

MTS and underlying technologies, COM and DCOM

[Topic groups](#) [See also](#)

MTS used the Component Object Model (COM) as its foundation to support the COM objects in client/server applications. COM defines a set of structured interfaces that enable components to communicate.

MTS uses DCOM for remote communication. To access a COM object on another machine, the client uses DCOM, which transparently transfers a local object request to the remote object running on a different machine. For remote procedure calls, DCOM uses the RPC protocol provided by Open Group's Distributed Computing Environment (DCE).

For distributed security, DCOM uses the NT LAN Manager (NTLM) security protocol. For directory services, DCOM uses the Domain Name System (DNS).

Resource pooling in the MTS environment is generally provided by an underlying database engine. In Delphi, resource pooling is provided by the Borland Database Engine. All database connections allocated by MTS objects come from this pool. These connections can participate in two-phase commit with MTS as the controller.

Overview of creating MTS objects

[Topic groups](#) [See also](#)

The process of creating an MTS component is as follows:

- 1 Use the [MTS Object wizard](#) to create an MTS component.
- 2 Add methods and properties to the application using the [Type Library editor](#).
- 3 [Debug and test](#) the MTS component.
- 4 [Install](#) the MTS component into a new or existing MTS package.
- 5 [Administer the MTS environment](#) using the MTS Explorer.

Using the MTS Object wizard

[Topic groups](#) [See also](#)

Use the MTS Object wizard to create an MTS object that allows client applications to access your server within the MTS runtime environment. MTS provides extensive runtime support such as resource pooling, transaction processing, and role-based security.

To bring up the MTS Object wizard,

- 1 Choose File|New.
- 2 Select the tab labeled Multitier.
- 3 Double-click the MTS Object icon.

In the wizard, specify the following:

ClassName	Specify the name the for the MTS class.
Threading Model	Choose the <u>threading model</u> to indicate how client applications can call your object's interface. This is the threading model that you commit to implementing in the MTS object. . Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.
Transaction Model	Specify whether and how this MTS object supports <u>transactions</u> .
Generate event support code	Check this box to tell the wizard to implement a separate interface for <u>managing events</u> of your MTS object.

When you complete this procedure, a new unit is added to the current project that contains the definition for the MTS object. In addition, the wizard adds a type library project and opens the type library. Now you can expose the properties and methods of the interface through the type library. You expose the interface as you would expose any Automation object as described in [Exposing an application's properties and methods](#).

The MTS object implements a **dual interface**, which supports both early (compile-time) binding through the vtable and late (runtime) binding through the *IDispatch* interface.

The MTS Object wizard implements the *IObjectControl* interface methods, Activate, Deactivate, and CanBePooled.

Choosing a threading model for an MTS object

[Topic groups](#)

The MTS runtime environment manages threads for you. MTS components should not create threads. Components must never terminate a thread that calls into a DLL.

When you specify the threading model when using the MTS wizard, you are specifying how the objects are assigned to threads for method execution.

<u>Threading model</u>	<u>Description</u>	<u>Implementation pros and cons</u>
Single	<p>No thread support. Client requests are serialized by the calling mechanism. All objects of a single-threaded component execute on the main thread.</p> <p>This is compatible with the default COM threading model, which is used for components that do not have a Threading Model Registry attribute or for COM components that are not reentrant. Method execution is serialized across all objects in the component and across all components in a process.</p>	<p>Allows components to use libraries that are not reentrant.</p> <p>Very limited scalability.</p> <p>Single-threaded, stateful components are prone to deadlocks. You can eliminate this problem by using stateless objects and calling SetComplete before returning from any method.</p>
Apartment (or Single-threaded apartment)	<p>Each object is assigned to a thread an apartment, which lasts for the life of the object; however, multiple threads can be used for multiple objects. This is a standard COM concurrency model. Each apartment is tied to a specific thread and has a Windows message pump.</p>	<p>Provides significant concurrency improvements over the single threading model.</p> <p>Two objects can execute concurrently as long as they are in different <u>activities</u>. These objects may be in the same component or in different components.</p> <p>Similar to a COM apartment, except that the objects can be distributed across multiple processes.</p>

Note: These threading models are similar to those defined by COM objects. However, because the MTS environment provides more underlying support for threads, the meaning of each threading model differs here. Also, the free threading model does not apply to objects running in the MTS environment due to the MTS support for activities.

MTS activities

MTS supports concurrency through **activities**. Every MTS object belongs to one activity, which is recorded in the object's context. The association between an object and an activity cannot be changed. An activity includes the MTS object created by the base client, as well as any MTS objects created by that object and its descendants. These objects can be distributed across one or more processes, executing on one or more computers.

For example, a physician's medical application may have an MTS object to add updates and remove records to various medical databases, each represented by a different object. This add object may use other objects as well, such as a receipt object to record the transaction. This results in several MTS objects that are either directly or indirectly under the control of the base client. These objects all belong to the same activity.

MTS tracks the flow of execution through each activity, preventing inadvertent parallelism from corrupting the application state. This feature results in a single logical thread of execution throughout a

potentially distributed collection of objects. By having one logical thread, applications are significantly easier to write.

When an MTS object is created from an existing context, using either a transaction context object or an MTS object context, the new object becomes a member of the same activity. In other words, the new context inherits the activity identifier of the context used to create it.

MTS allows only a single logical thread of execution within an activity. This is similar in behavior to a COM apartment, except that the objects can be distributed across multiple processes. When a base client calls into an activity, all other requests for work in the activity (such as from another client thread) are blocked until after the initial thread of execution returns back to the client.

For more information on threading in the MTS environment, search the MTS documentation for the topic, Components and Threading.

Setting the transaction attribute

[Topic groups](#) [See also](#)

You set a transaction attribute either at design time or at runtime.

At design time, the MTS Object wizard prompts you to choose the transaction attribute.

You can change the transaction attribute at runtime by using the Type Library editor.

To change a transaction attribute at runtime,

- 1 Choose View|Type Library to open the Type Library editor.
- 2 Select the class corresponding to the MTS object.
- 3 Click the Transaction tab and choose the desired transaction attribute.

Note: If the MTS object is already installed into the runtime environment, you must first uninstall the object and reinstall it. Use Run|Install MTS objects to do so.

In addition, you can change the transaction attribute of an object installed in the MTS runtime environment by using the MTS Explorer.

Passing object references

[Topic groups](#)

You can pass object references, for example, for use as a [callback](#), only in the following ways:

- Through return from an object creation interface, such as *CoCreateInstance* (or its equivalent), *ITransactionContext.CreateInstance*, or *IObjectContext.CreateInstance*.
- Through a call to *QueryInterface*.
- Through a method that has called [SafeRef](#) to obtain the object reference.

An object reference that is obtained in the above ways is called a **safe reference**. MTS ensures that methods invoked using safe references execute within the correct context.

Calls that use safe references always pass through the MTS runtime environment. This allows MTS to manage context switches and allows MTS objects to have lifetimes that are independent of client references.

Using the SafeRef method

An object can use the *SafeRef* function to obtain a reference to itself that is safe to pass outside its context.

The unit that defines the *SafeRef* function is *Mtx*.

SafeRef takes as input

- A reference to the interface ID (RIID) of the interface that the current object wants to pass to another object or client.
- A reference to the current object's *IUnknown* interface.

SafeRef returns a pointer to the interface specified in the RIID parameter that is safe to pass outside the current object's context. It returns **nil** if the object is requesting a safe reference on an object other than itself, or the interface requested in the RIID parameter is not implemented.

When an MTS object wants to pass a self-reference to a client or another object (for example, for use as a [callback](#)), it should always call *SafeRef* first and then pass the reference returned by this call. An object should never pass a **self** pointer, or a self-reference obtained through an internal call to *QueryInterface*, to a client or to any other object. Once such a reference is passed outside the object's context, it is no longer a valid reference.

Calling *SafeRef* on a reference that is already safe returns the safe reference unchanged, except that the reference count on the interface is incremented.

When a client calls *QueryInterface* on a reference that is safe, MTS automatically ensures that the reference returned to the client is also a safe reference.

An object that obtains a safe reference must release the safe reference when it is finished with it.

For details on *SafeRef* see the *SafeRef* topic in the MTS documentation.

Callbacks

Objects can make callbacks to clients and to other MTS components. For example, you can have an object that creates another object. The creating object can pass a reference of itself to the created object; the created object can then use this reference to call the creating object.

If you choose to use callbacks, note the following restrictions:

- Calling back to the base client or another package requires access-level security on the client. Additionally, the client must be a DCOM server.
- Intervening firewalls may block calls back to the client.
- Work done on the callback executes in the environment of the object being called. It may be part of the same transaction, a different transaction, or no transaction.
- The creating object must call *SafeRef* and pass the returned reference to the created object in order to call back to itself.

Setting up a transaction object on the client side

[Topic groups](#)

A client base application can control transaction context through the *ITransactionContextEx* interface. The following code example shows how a client application uses *CreateTransactionContextEx* to create the transaction context. This method returns an interface to this object.

This example wraps the call to the transaction context in a call to OleCheck which is necessary because the CreateInstance method is not declared as **safecall**.

```
procedure TForm1.MoveMoneyClick(Sender: TObject);
begin
    Transfer(CLASS_AccountA, CLASS_AccountB, 100);
end;
procedure TForm1.Transfer(DebitAccountId, CreditAccountId: TGuid; Amount: Currency);
var
    TransactionContextEx: ITransactionContextEx;
    CreditAccountIntf, DebitAccountIntf: IAccount;
begin
    TransactionContextEx := CreateTransactionContextEx;
    try
        OleCheck(TransactionContextEx.CreateInstance(DebitAccountId,
            IAccount, DebitAccountIntf));
        OleCheck(TransactionContextEx.CreateInstance(CreditAccountId,
            IAccount, CreditAccountIntf));
        DebitAccountIntf.Debit(Amount);
        CreditAccountIntf.Credit(Amount);
    except
        TransactionContextEx.Abort;
        raise;
    end;
    TransactionContextEx.Commit;
end;
```

Setting up a transaction object on the server side

[Topic groups](#)

To control transaction context from the MTS server side, you create an instance of *ObjectContext*. In the following example, the Transfer method is in the MTS object. In using ObjectContext this way, the instance of the object we are creating will inherit all the transaction attributes of the object who is creating it. We wrap the call in a call to OleCheck because the CreateInstance method is not declared as **safecall**.

```
procedure TAccountTransfer.Transfer(DebitAccountId, CreditAccountId: TGuid;
  Amount: Currency);
var
  CreditAccountIntf, DebitAccountIntf: IAccount;
begin
  try
    OleCheck(ObjectContext.CreateInstance(DebitAccountId,
      IAccount, DebitAccountIntf));
    OleCheck(ObjectContext.CreateInstance(CreditAccountId,
      IAccount, CreditAccountIntf));
    DebitAccountIntf.Debit(Amount);
    CreditAccountIntf.Credit(Amount);
  except
    DisableCommit;
    raise;
  end;
  EnableCommit;
end;
```


Debugging and testing MTS objects

[Topic groups](#) [See also](#)

You can debug local and remote MTS objects. When debugging MTS objects, you may want to turn off transaction timeouts.

The transaction timeout sets how long (in seconds) a transaction can remain active. Transactions that are still alive after the timeout are automatically aborted by the system. By default, the timeout value is 60 seconds. You can disable transaction timeouts by specifying a value of 0, which is useful when debugging MTS objects.

When testing the MTS object, you may first want to test your object outside the MTS environment to simplify your test environment.

While developing an MTS server, you cannot rebuild a server when it is still in memory. You may get a compiler error like, "Cannot write to DLL while executable is loaded." To avoid this, you can set the package properties in the MTS Explorer to shut down the server when it is idle.

To shut down the MTS server when idle,

- 1 In the MTS Explorer, right-click the package in which your MTS component is installed and choose Properties.
- 2 Select the Advanced tab.
The Advanced tab determines whether the server process associated with a package always runs, or whether it shuts down after a certain period of time.
- 3 Change the timeout value to 0, which shuts down the server as soon as no longer has a client to service.
- 4 Click OK to save the setting and return to the MTS Explorer.

Note: When testing outside the MTS environment, you do not reference the *ObjectProperty* of *TMtsObject* directly. The *TMtsObject* implements methods such as *SetComplete* and *SetAbort* that are safe to call when the object context is **nil**.

Installing MTS objects into an MTS package

[Topic groups](#) [See also](#)

MTS applications consist of a group of in-process MTS objects (or MTS remote data modules) running in a single instance of the MTS executive (EXE). A group of COM objects that all run in the same process is called a **package**. A single machine can be running several different packages, where each package is running within a separate MTS EXE.

You can group your application components into a single package to run within a single process. You might want to distribute your components into different packages to partition your application across multiple processes or machines.

To install MTS objects into a package,

- 1 Choose Run|Install MTS Objects to install MTS objects into a package.
- 2 Check the MTS objects to be installed.
- 3 In the Install Object dialog box, choose the Into New Package to create a new package in which to install the MTS object or choose Into Existing Package to install the object into one of the existing MTS packages listed.
- 4 Choose OK to refresh the MTS catalog, which makes the objects available at runtime.

Packages can contain components from multiple DLLs, and components from a single DLL can be installed into different packages. However, a single component cannot be distributed among multiple packages.

Administering MTS objects with the MTS Explorer

[Topic groups](#) [See also](#)

Once you have installed MTS objects into an MTS runtime environment, you can administer these runtime objects using the MTS Explorer. The MTS Explorer is a graphical user interface for managing and deploying MTS components. With the MTS Explorer, you can

- Configure MTS objects, packages, and roles
- View properties of components in a package and view packages installed on a computer
- Monitor and manage transactions for MTS components that comprise transactions
- Move packages between computers
- Make a remote MTS object available to a local client

For details on the MTS Explorer, see the *MTS Administrator's Guide*.

Using MTS documentation

[Topic groups](#)

The documentation accompanying Microsoft's MTS provides thorough details of MTS concepts, programming scenarios, and administration tools. This documentation is likely to help those who are new to developing MTS applications.

Here is the roadmap of MTS documentation that accompanies the Microsoft product.

Source	Description
<i>Setting Up MTS</i>	Describes how to set up MTS and MTS components, including instructions for accessing Oracle databases from MTS application and installing MTS sample applications.
<i>Getting Started with MTS</i>	Provides an overview of the new features in MTS, gives a brief tour of the documentation, and contains a glossary of terms.
<i>Quick Tour of MTS</i>	Provides an overview of MTS.
<i>MTS Administrator's Guide</i>	
Roadmap to the MTS Administrator's Guide	Describes the different ways to use the MTS Explorer to deploy and administer applications, and gives an overview of the MTS Explorer graphical interface.
Creating MTS Packages	Provides task-oriented documentation for creating and assembling MTS packages.
Distributing MTS Packages	Provides task-oriented documentation for distributing MTS packages.
Installing MTS Packages	Provides task-oriented documentation for installing and configuring MTS packages.
Maintaining MTS Packages	Provides task-oriented information for maintaining and monitoring MTS packages.
Managing MTS Transactions	Describes distributed transactions and the management of transactions using the MTS Explorer.
Automating MTS Administration	Provides a conceptual overview, procedures, and sample code explaining how to use the MTS scriptable objects to automate procedures in the MTS Explorer.
<i>MTS Programmer's Guide</i>	
Overview and Concepts	Provides an overview of the product and how the product components work together, explains how MTS addresses the needs of client/server developers and system administrators, and provides in-depth coverage of programming concepts for MTS components.
Building Applications for MTS	Provides task-oriented information for developing ActiveX™ components for MTS.
MTS Administrative Reference	Provides a reference for using the MTS scriptable objects to automate procedures in the MTS Explorer.
MTS Reference	Provides a reference for the MTS application programming interface (API).

Related topic groups

Developing COM-based applications

- [Overview of component creation](#)
- [Object-oriented programming for component writers](#)
- [Creating properties](#)
- [Creating events](#)
- [Creating methods](#)
- [Creating an Active Server Page](#)
- [Working with type libraries](#)
- [Creating MTS automation objects](#)

Overview of component creation

[Related topic groups](#)

- [Overview of COM Technologies](#)
- [Parts of a COM application](#)
- [COM interfaces](#)
- [The fundamental COM interface, IUnknown](#)
- [COM interface pointers](#)
- [COM servers](#)
- [COM clients](#)
- [COM extensions](#)
- [Automation](#)
- [ActiveX controls](#)
- [Type libraries](#)
- [Active Server Pages](#)
- [Active Documents](#)
- [Visual cross-process objects](#)
- [Implementing COM objects with Wizards](#)

Object-oriented programming for component writers

[Related topic groups](#)

- [Creating a simple COM object: Overview](#)
- [Designing a COM object](#)
- [Creating a COM object with the COM object wizard](#)
- [COM object instancing types](#)
- [Choosing a threading model](#)
- [Registering a COM object](#)
- [Testing a COM object](#)

Creating properties

[Related topic groups](#)

- [Creating an Automation controller: Overview](#)
- [Creating an Automation controller by importing a type library](#)
- [Example: Printing a document with Microsoft Word](#)

Creating events

[Related topic groups](#)

- [Creating an Automation server: Overview](#)
- [Creating an Automation object for an application](#)
- [Managing events in your Automation object](#)
- [Exposing an application's properties, methods, and events](#)
- [Registering an application as an Automation server](#)
- [Testing and debugging the application](#)
- [Automation interfaces](#)
- [Dual interfaces](#)
- [Dispatch interfaces](#)
- [Custom interfaces](#)
- [Marshaling data](#)

Creating methods

[Related topic groups](#)

- [Creating an ActiveX control: Overview](#)
- [Elements of an ActiveX control](#)
- [Designing an ActiveX control](#)
- [Generating an ActiveX control from a VCL control](#)
- [Licensing ActiveX controls](#)
- [Generating an ActiveX control based on a VCL form](#)
- [Working with properties, methods, and events in an ActiveX control](#)
- [Enabling simple data binding with the type library](#)
- [Enabling simple data binding of ActiveX controls in the Delphi container](#)
- [Creating a property page for an ActiveX control](#)
- [Creating a new property page](#)
- [Adding controls to a property page](#)
- [Associating property page controls with ActiveX control properties](#)
- [Updating the property page](#)
- [Updating the object](#)
- [Connecting a property page to an ActiveX control](#)
- [Exposing properties of an ActiveX control](#)
- [Registering an ActiveX control](#)
- [Testing an ActiveX control](#)
- [Deploying an ActiveX control on the Web](#)
- [Setting Web deployment options](#)
- [Option combinations](#)
- [Project tab](#)
- [Packages tab](#)
- [Additional Files tab](#)

Creating an Active Server Page

[Related topic groups](#)

- [Creating Active Server Pages: Overview](#)
- [Creating an Active Server Page object](#)
- [Creating ASPs for in-process or out-of-process servers](#)
- [Registering an application as an Active Server Page object](#)
- [Testing and debugging the Active Server Page application](#)

Working with type libraries

[Related topic groups](#)

- [Working with type libraries: Overview](#)
- [Type Library editor](#)
- [Toolbar](#)
- [Object list pane](#)
- [Status bar](#)
- [Pages of type information](#)
- [Type library information](#)
- [Interface pages](#)
- [Attributes page for an interface](#)
- [Interface flags](#)
- [Interface members](#)
- [Type Library editor, Interface methods;Interface methods](#)
- [Interface properties](#)
- [Property and method parameters page](#)
- [Dispatch type information](#)
- [Attributes page for dispatch](#)
- [Dispatch flags page](#)
- [Dispatch members](#)
- [CoClass Pages](#)
- [Attributes for a CoClass](#)
- [CoClass Implements page](#)
- [CoClass flags](#)
- [Enumeration type information](#)
- [Attributes page for an enum](#)
- [Enumeration members](#)
- [Alias type information](#)
- [Record type information](#)
- [Union type information](#)
- [Module type information](#)
- [Creating new type libraries](#)
- [Valid types](#)
- [SafeArrays](#)
- [Using Object Pascal or IDL syntax](#)
- [Creating a new type library](#)
- [Opening an existing type library](#)
- [Adding an interface to the type library](#)
- [Adding properties and methods to the type library](#)
- [Adding a CoClass to the type library](#)
- [Adding an enumeration to the type library](#)
- [Saving and registering type library information](#)
- [Apply Updates dialog](#)

- Saving a type library
- Refreshing the type library
- Registering the type library
- Exporting an IDL file
- Deploying type libraries

Creating MTS automation objects

[Related topic groups](#)

- [Creating MTS objects: Overview](#)
- [Microsoft Transaction Server \(MTS\) components](#)
- [Managing resources with just-in-time activation and resource pooling](#)
- [MTS transaction support](#)
- [Role-based security](#)
- [Resource dispensers](#)
- [Base clients and MTS components](#)
- [MTS and underlying technologies, COM and DCOM](#)
- [Overview of creating MTS objects](#)
- [Using the MTS Object wizard](#)
- [Choosing a threading model for an MTS object](#)
- [Setting the transaction attribute](#)
- [Passing object references](#)
- [Setting up a transaction object on the client side](#)
- [Setting up a transaction object on the server side](#)
- [Debugging and testing MTS objects](#)
- [Installing MTS objects into an MTS package](#)
- [Administering MTS objects with the MTS Explorer](#)
- [Using MTS documentation](#)

Link not found

The topic you requested is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.



The topic you requested is now loading. If it does not appear within a few seconds, the topic is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.

