ObjectComponents Framework Help Contents

Click the icon above to open all folders. Click an icon below to open a folder or click the underlined text to see a specific topic.

Welcome to Help for Borland C++ ObjectComponents Framework (OCF).

Shortcut: Use the **Search** button at the top of the Help window if you are ready to look for something by name.

ObjectComponents Framework Help Contents

Click the icon above to open all folders. Click an icon below to open or close a folder. Click the underlined text to see a specific topic.

Welcome to Help for Borland C++ ObjectComponents Framework (OCF).

Shortcut: Use the **Search** and **Search All** buttons at the top of the Help window if you are ready to look for an item by name.

ObjectComponents Framework Help Contents

Click the icon above to open all folders. Click an icon below to open or close a folder. Click the underlined text to see a specific topic.

Welcome to Help for Borland C++ ObjectComponents Framework (OCF).

Shortcut: Use the **Search** button at the top of the Help window if you are ready to look for something by name.

ObjectComponents Framework Help Contents

Click the icon above to open all folders. Click an icon below to open or close a folder. Click the underlined text to see a specific topic.

Welcome to Help for Borland C++ ObjectComponents Framework (OCF).

Essentials and concepts you will need to know to work with OCF.

Tasks with step-by-step directions for using Borland C++ tasks and its utilities.

Language Reference for OCF language elements and error messages.

Shortcut: Use the **Search** button at the top of the Help window if you are ready to look for something by name.

ObjectComponents Framework Help Contents

Click any icon to close all folders or click the underlined text to see a specific topic.

Welcome to Help for Borland C++ ObjectComponents Framework (OCF).

Essentials

Tasks

ObjectComponents Framework Language Reference

Automation Macros

Automating an Application

See Also

Automating a program means exposing its functions to other programs. Once a program is automated, other programs can control it by issuing commands through OLE. ObjectComponents is your interface to OLE automation. Through ObjectComponents you can expose any C++ classes to OLE, and you won't have to restructure your existing classes to do it.

Automating a Progam

The following steps for automating an application are the same whether the application uses ObjectWindows or not. They will work for any C++ application.

1. Register the automation server
2. Automate a class
3. Build the server

To enhance your server by localizing symbol names, combining C++ objects, exposing collections, invalidating deleted objects, or creating a type library, see Enhancing Automation Server Functions.

ObjectComponents lets OLE reach members of your classes through standard C++ mechanisms. At run time, other programs can send commands for your program to OLE.

A program that exposes itself to receive automation commands is called an *automation server*. A program that sends commands for others to execute is called an *automation controller*.

Automation servers can be built as DLLs using the same methods for making an in-process linking and embedding server.

Note: The AutoCalc example program installed in the EXAMPLES/OCF/AUTOCALC directory illustrates many of the above steps. AutoCalc draws a calculator on the screen and lets the user click buttons to perform calculations. AutoCalc also automates its classes so that a controller application can send commands to do the same things a user does.

See Also

Creating an Automation Controller

Making a DLL server

Registering an Automation Server

See Also

Registering an application means giving OLE information about what the application can do.

To register the application.

- Create a registration table and record the information.
- Create a registrar object and pass the table to the constructor of the registrar object.

See Also

Automating a Class

Building the Server

Enhancing Automation Server Functions

Creating a Registration Table

See Also Registering an Automation Server

An automation server must set four pieces of information in the application's registration table: its program ID, its class ID, a description of itself, and command-line arguments for invoking the automation server.

```
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,      "{877B6200-7627-101B-B87C-0000C057CE4E}")
  REGDATA(progid,     "Calculator.Application")
  REGDATA(appname,    "AutoCalc"
  REGDATA(description,"Automated Calculator 1.2 Application")
  REGDATA(cmdline,    "/Automation")
  REGDATA(version,    "1.2")
END_REGISTRATION
```

The registration macros create a structure that this example names *AppReg*. The clsid must be specially generated to ensure uniqueness. To learn how, refer to the entry for the *clsid* registration key in the *ObjectWindows Reference Guide*. The *progid* for an automation server conventionally has two parts, one naming the program ("Calculator") and one naming the type of object ("Application"). A period (".") is the only permissible delimiter character in a *progid*.

A server that creates several different kinds of automatable objects must give each a different *progid*, such as *AppName*.*MySecondObject* and *AppName*.*MyThirdObject*. You need not, however, supply a different *clsid* for each kind of object, only for the first one. ObjectComponents increments the first *clsid* for each subsequent object.

The *progid* is visible to users when they create your object in their automation scripts. They also see the *description string* when OLE browses the system for available automation objects.

For the final registration key, *cmdline*, an automation server should normally include the -Automation switch. When an automation controller asks to create your object, OLE invokes your application and places the *cmdline* value on the command line.

Although OLE conventions call for this switch, OLE itself pays no attention to it. ObjectComponents uses it, though, to determine whether to invoke new instances of your program for each new OLE client or whether a single instance should service all clients. Normally you don't want several different clients sending commands to the exact same object. The commands they send might interfere with each other. ObjectComponents responds to the -Automation switch by making the application support only one client per instance. (The -Automation switch overrides *ocrMultipleUse* if you register that in your *usage* key.) If it makes sense for your automation server to support simultaneous clients with a single instance, then register it as *ocrMultipleUse* and omit -Automation from the *cmdline* string.

The -Automation switch directs the creation of an OLE factory, the facility that ObjectComponents registers at run time for OLE to call when it wants to create whatever the server produces. When the switch is present, ObjectComponents registers the application class so that OLE can create an instance of the application. ObjectComponents does not register the document factory. To make the application single-use, ObjectComponents removes the factory after it creates the first instance of the application.

The registration table in the example holds information about the application, so it is called an *application registration table*. Using the same macros, an application can also create *document registration structures* to describe the kinds of documents (or objects) that the server produces. An automation server creates automated documents if it wants controllers to embed the automated object before issuing commands.

The following table briefly describes all the registration keys that might be used by an automation server. It shows which are optional and which required, as well as which belong in the application registration table (usually named *AppReg*) and which in the document registration table (usually named *DocReg*). The table assumes that the server's documents support automation. For non-automated document types, the server needs to register only docflags and docfilter.

Key	in AppReg?	in DocReg?	Description
appname	Yes	Optional	Short name for the application
clsid	Yes	Optional	Globally unique identifier (GUID); generated automatically for the DocReg structure
description	Yes	Yes	Descriptive string (up to 40 characters)
progid	Yes	Yes	Name of program or object type (unique string)
extension	No	Optional	Document file extension associated with server
docfilter	No	Yes	Wildcard file filter for File Open dialog box
			Note: *dtHidden* is not in DocReg.
docflags	No	Yes	Options for running the File Open dialog box
typehelp	No	Optional	Name of an .HLP file documenting supported commands
helpdir	No	Optional	Path to an .HLP file documenting supported commands (defaults to current module path)
debugger	Optional	Optional	Command line for running debugger.
debugprogid	Optional	Optional	Name of debugging version (unique string)
debugdesc	Optional	Optional	Description of debugging version
cmdline	Yes	No	Arguments to place on server's command line
path	Optional	No	Path to server file (defaults to current module path)
permid	Optional	Optional	Name string without version information
permname	Optional	Optional	Descriptive string without version information
usage	Optional	Optional	Support for concurrent clients
language	Optional	No	Language for registered strings (defaults to system's user language setting)
version	Optional	No	Major and minor version numbers (defaults to "1.0")

The table assumes that the server's documents support automation. For non-automated document types, the server needs to register only *docflags* and *docfilter*

Register all the keys that are required in any of the tables that apply to your application.

For more information about individual registration keys and the values they hold, see the *ObjectWindows Reference Guide*.

An automation server that supports system language settings should localize the *description* string it registers. (The *progid* must never be localized)

The complete *AppReg* structure is later passed to the program's *TRegistrar* object and written to the registry.

See Also

Building Registration Tables

Localizing Symbol Names

Registration Keys

Creating a Registrar Object

See Also Registering an Automation Server

An automation server needs a registrar object, just as linking and embedding applications do. Applications that support only automation, however, without linking and embedding, should create *TRegistrar* instead of *TOcRegistrar*. *TRegistrar* is the base class for *TOcRegistrar*. *TOcRegistrar* extends *TRegistrar* by connecting the application to the BOCOLE support library interfaces that support linking and embedding.

First, declare a static pointer to hold the *TRegistrar**. Use the *TPointer*<> template to ensure that the registrar object is properly deleted when the program ends.

```
TPointer<TRegistrar> Registrar;     // initialized at WinMain or LibMain
```

In the main procedure (for AutoCalc, this is *WinMain*), you should create the registrar object and call its *Run* method.

```
Try {
  ::Registrar=new
  TRegistrar(AppReg,TOcAutoFactory<TCalc>,string(cmdLine),hInst);
  if (!::Registrar->IsOptionSet(amAnyRegOption))
    ::Registrar->Run();
  ::Registrar = 0;                          // deletes registrar by
  replacing pointer
  return 0;
}
catch (TXBase& x) {
  ::MessageBox(0, x.why().c_str(), "OLE Exception", MB_OK);
}
```

The first parameter of the *TRegistrar* constructor is the application registration structure, conventionally named *AppReg*. The second parameter is a factory callback function. The example uses a factory template to create the callback. For an automation server that doesn't use ObjectWindows, the appropriate template is *TOcAutoFactory*.The call to *IsOptionSet* determines whether the application was passed a command-line switch asking the application to register itself in the system registration database and then quit. If not, the application calls *Run*. The registrar then calls the factory callback, where the message loop resides. When *Run* returns, the application has ended.

See Also

Factory Template Classes

TOcRegistrar

TRegistrar

TRegistrar::IsOptionSet

TRegistrar::Run

Automating a Class

See Also

Automating a class requires building two tables, one in the class declaration and one in the class implementation.

The first table is called the automation declaration, where you declare automatable methods and properties or which members of the class a controller can reach.

The second table is called the automation definition, where you define external methods and properties and give them public names that a controller uses to reach each exposed class member.

See Also

Building the Server

Enhancing Automation Server Functions

Registering an Automation Server

TOcRegistrar

TRegistrar::Run

Declaring Automatable Methods and Properties

See Also   Automating a Class

Automating a class requires building two tables, one in the class declaration and one in the class implementation. The first table is called the *automation declaration*, and it declares which members of the class a controller can reach. The second table is called the *automation definition*, and it defines public names that a controller uses to reach each exposed class member. This section tells how to build an automation declaration.

To create an automation declaration, you will need to:

- Write Declaration Macros

You might also want to set hooks to monitor automation commands.

- Provide Optional Hooks for Validation and Filtering

The automation declaration belongs inside the declaration of an automated class. It begins with the macro DECLARE_AUTOCLASS and includes one entry for each class member that you choose to expose. The macros add nested classes that ObjectComponents instantiates to process commands received from OLE. They do not alter the structure or size of the original class.

This sample automation declaration exposes functions and data members of a C++ class that mimics a calculator:

```
DECLARE_AUTOCLASS(TCalc)
  AUTODATA  (Accum,   Accum,     long,  )
  AUTODATA  (Opnd,    Opnd,      long,  )
  AUTODATA  (Op,      Op, short, AUTOVALIDATE(Val>=OP_NONE &&
  Val<=OP_CLEAR) )
  AUTOFUNC0 (Eval,    Eval,      TBool, )
  AUTOFUNC0V(Clear,   Clear,            )
  AUTOFUNC0V(Display, Display,          )
  AUTOFUNC0V(Quit,    Quit,             )
  AUTOFUNC1 (Button,  Button,    TBool, TAutoString,)
  AUTOFUNC0 (Window,  GetWindow, TAutoObject<TCalcWindow>,  )
  AUTOFUNC1 (LookAt,  LookAtWindow, long, TAutoObject<const TCalcWindow>,)
  AUTODATARO(MyArray, Elem,      TAutoObjectByVal<TMyArray>,)
```

The automated class is called *TCalc*. Each AUTOFUNC or AUTODATA macro exposes one member of *TCalc*. Some of the *TCalc* member functions are *Eval, Clear, Display,* and *Quit*. Its data members include *Accum, Opnd, Op,* and *Elem*. *TCalc* also has other members that it chooses not to automate and so excludes from the declaration table.

No termination macro is needed for an automation declaration. The END_AUTOCLASS macro that closes an automation definition is not used here. Also, each line of the declaration ends with a closing parenthesis, not with punctuation.

Note: The automation declaration should appear at the end of a class declaration because the macros can modify the access specifier. If you put the declaration anywhere other than the end, be sure to follow it immediately with an access specifier (**public**, **protected**, or **private**).

See Also

AUTODATA Macros

AUTOFUNC Macros

Automation Declaration Macros

DECLARE_AUTOCLASS Macro

END_AUTOCLASS Macro

Writing Declaration Macros

See Also Automating a Class

Each of the macros within an automation declaration describes a single method or property that other programs can manipulate. The different macros expose different kinds of class members. AUTOFUNC1, for example, exposes a member function that takes one parameter. AUTOFUNC2V exposes a function that takes two parameters and returns nothing (**void**). AUTOPROP exposes a property through Set and Get functions that insert or retrieve a single value. AUTODATA exposes a data member that the controller can read and modify directly.

The general form of the automation macros is this:

```
MACRONAME( InternalName, FunctionName, ReturnType, ArgumentType, Options )
```

Some of the macros don't use all five parameters. AUTOFUNC1V, for example, doesn't have a *ReturnType* because the function has a **void** return. AUTOFUNC0 doesn't have any arguments, while AUTOFUNC2 has two different arguments. But whatever parameters are relevant appear in the order shown.

InternalName is an identifier you assign to each automatable property or function. It is used internally by ObjectComponents for keeping track of the members. The only other place you ever use the internal name is in the corresponding entry of the class's automation definition table. The internal name is a unique identifier for the member. (the names used in source code are not necessarily unique. They can be overloaded, for example.)

FunctionName is the name you use in your source code to refer to the same property or function. *FunctionName* can be any expression that evaluates to a function call. The expression must, however, be defined within the scope of the automated object. ObjectComponents attempts to reach the function through the **this** pointer.

The internal and function names should be the same unless the function name is overloaded or uses indirection. For example, suppose a class contains a data member that points to another object:

```
TObject* MyObject;
```

To expose a function call like *MyObject->MyFunction,* you should supply an internal name that does not use indirection. In this case, a good choice would be *MyFunction*.

```
AUTOFUNC0V( MyFunction, MyObject->MyFunction, )
```

If a function is overloaded, use the same function name for all versions but give each a different internal name. ObjectComponents can distinguish the overloaded functions by the return types and argument types in the parameters that follow.

The *ReturnType* and *ArgumentType* parameters can be any fundamental C type, such as **int** or **char**, or a pointer to any fundamental type. Some pointers, however, require special handling. If the data type is a string (type **char***), declare it to be a *TAutoString* instead. If the data type is a pointer or a reference to a C++ object, then declare it using the *TAutoObject*<> wrapper. The type substitutions help ObjectComponents convert between C++ data types and the VARIANT union type that OLE uses. Pointers and object references are hardest to convert because they refer to data that is not in the variable itself. The *TAutoString* and *TAutoObject* classes provide type information for the conversion so that ObjectComponents can pass the right information between server and controller applications.

The *TCalc* example shows how to use *TAutoObject*. One of the functions *TCalc* exposes is *GetFunction*, which returns a reference to a *TCalcWindow* object.

```
AUTOFUNC0 (Window, GetWindow, TAutoObject<TCalcWindow>, )
```

When it declares *TCalcWindow* as the return type, it makes use of the *TAutoObject* template to create a smart, self-describing pointer to a *TCalcWindow* object.

See Also
AUTODATA Macros
AUTOFUNC Macros
Automation Declaration Macros
DECLARE_AUTOCLASS Macro
TAutoObject
TAutoString

Providing Optional Hooks for Validation and Filtering

See Also Automating a Class

The final parameter of every automation macro names a hook function to be called whenever OLE calls the exposed class member. A *hook* is code that executes every time anyone uses a particular class member. ObjectComponents supports hooks to record commands, undo commands, validate command arguments, and override a command's implementation. Hooks are always optional.

To install a hook, use one of these macros as the last parameter to any automation declaration:

- AUTOINVOKE
- AUTORECORD
- AUTOUNDO
- AUTONOHOOK
- AUTOREPORT
- AUTOVALIDATE

Each macro receives a single parameter containing code to execute. The form of the required macro varies with its function.

To validate arguments, for example, the code should be a Boolean expression. The *Op* data member of *TCalc* holds an integer that identifies an operation to perform, such as addition or subtraction. The automation declaration installs a hook to be sure that *Op* is not assigned a value outside the legal range of operator identifiers.

```
AUTODATA(Op, Op, short, AUTOVALIDATE(Val>=OP_NONE && Val<=OP_CLEAR))
```

AUTOVALIDATE introduces the expression to execute for validation. Within the validation expression, use the name *Val* to represent the value received from the controller. When used to validate function arguments, AUTOVALIDATE uses the names *Arg1, Arg2, Arg3*, and so on.

Whenever any automation controller attempts to set a value in the *Op* data member, ObjectComponents verifies that the new value falls within the range OP_NONE to OP_CLEAR. If passed an illegal value, ObjectComponents cancels the command and sends OLE an error result.

The expression passed to AUTOVALIDATE can include function calls.

```
AUTODATA(Op, Op, short, AUTOVALIDATE(Val>=OP_NONE && NotTooBig(Val))
```

Now ObjectComponents calls NotTooBig whenever a controller attempts to modify *Op*.

```
bool NotTooBig(int Val) {
  return (Val <= OP_CLEAR)
}
```

See Also

AUTOINVOKE Macro

Automation Declaration Macros

Automation Hook Macros

AUTONOHOOK Macro

AUTORECORD Macro

AUTOREPORT Macro

AUTOUNDO Macro

AUTOVALIDATE Macro

Defining External Methods and Properties

See Also

To create an automation definition, you will need to:

- Write an Automation Definition Table
- Use Data Type Specifiers in an Automation Definition

In addition, if you want to expose enumerated values through automation, you also need this step:

- Expose Data for Enumeration

Besides declaring which of its members are automatable, an automated class must also create a second table of macros to assign public symbols for referring to the exposed methods and properties. The public symbols are what other applications see. They become the controller's interface to an automated OLE object.

Behind the scenes, ObjectComponents links the public names to the C++ object or objects that you create to implement the OLE object. The automation declaration table identifies which class members to expose, and the automation definition table assigns them names.

The automation definition belongs with the class implementation. It begins with the DEFINE_AUTOCLASS macro and ends with END_AUTOCLASS. Here's the automation definition for *TCalc*:

```
DEFINE_AUTOCLASS(TCalc)
  EXPOSE_PROPRW(Opnd,      TAutoLong,  "Operand",      "@Operand_",
  HC_TCALC_OPERAND)
  EXPOSE_PROPRW_ID(0,Accum,TAutoLong,  "!Accumulator", "@Accumulator_",
        HC_TCALC_ACCUMULATOR)
  EXPOSE_PROPRW(Op,        CalcOps,    "Op",           "@Op_",
  HC_TCALC_OPERATOR)
  EXPOSE_METHOD(Eval,      TAutoBool,  "!Evaluate",    "@Evaluate_",
  HC_TCALC_EVALU ATE)
  EXPOSE_METHOD(Clear,     TAutoVoid,  "!Clear",       "@Clear_",
  HC_TCALC_CLEAR)
  EXPOSE_METHOD(Display,   TAutoVoid,  "!Display",     "@Display_",
  HC_TCALC_DISPLA Y)
  EXPOSE_METHOD(Quit,      TAutoVoid,  "!Quit",        "@Quit_",
  HC_TCALC_QUIT)
  EXPOSE_METHOD(Button,    TAutoBool,  "!Button",      "@Button_",
  HC_TCALC_BUTTON )
    REQUIRED_ARG(           TAutoString,"!Key")
  EXPOSE_PROPRO(Window,    TCalcWindow,"!Window",      "@Window_",
  HC_TCALC_WI NDOW)
  EXPOSE_METHOD(LookAt,    TAutoLong,  "!LookAtWindow","@LookAtWindow_",
        HC_TCALC_LOOKATWINDOW)
    REQUIRED_ARG(           TCalcWindow,"!Window")
  EXPOSE_PROPRO(MyArray,   TMyArray,   "!Array",       "@Array_",
  HC_TCALC_ARRAY)
  EXPOSE_APPLICATION(      TCalc,      "!Application", "@Application_",
        HC_TCALC_APPLICATION)
END_AUTOCLASS(TCalc, tfNormal, "TCalc", "@TCalc", HC_TCALC)
```

The EXPOSE_xxxx macros assign names to methods and properties. EXPOSE_PROPRW defines a property that controllers can both read and write. EXPOSE_PROPRO limits a controller's access so it can only read the property value. REQUIRED_ARG assigns a name to a function argument.

for example, a controller invokes the *LookAt* function by using the name *LookAtWindow*, and it calls the function's one parameter *Window*. The DEFINE_AUTOCLASS and END_AUTOCLASS macros assign "TCalc" as the public name for objects of type *TCalc*.

Most of the strings in this automation definition begin with a symbol, either ! or @. These symbols indicate that the AutoCalc application has in its resources translations for each public symbol. Each command from an automation controller comes with a locale ID indicating the language the controller is using. If the controller was written in German, for example, it can pass the string "Auswerten" instead of "Evaluate," and ObjectComponents correctly invokes the *Eval* function.

Every item listed in the automation definition must already appear in the automation declaration. For example, every function name you define with EXPOSE_METHOD must have a corresponding AUTOFUNC declaration. Every EXPOSE_PROP must have a corresponding AUTOPROP, AUTOFUNC, AUTOFLAG, or AUTODATA, depending on how you implement the property.

See Also

AUTODATA Macros
AUTOFLAG Macro
AUTOFUNC Macros
Automation Declaration Macros
Automation Definition Macros
AUTOPROP Macros
DEFINE_AUTOCLASS Macro
END_AUTOCLASS Macro
EXPOSE_METHOD Macros
EXPOSE_PROPxxxx Macros
EXPOSE_xxxx Macros
Localizing Symbol Names
REQUIRED_ARG Macro

Writing Definition Macros

See Also		Defining External Methods and Properties

The macros for exposing methods and properties have five parameters: the internal name, the type of value returned, the external name, and a documentation string. The optional fifth parameter allows you to associate a Help context ID with each member.

```
MACRONAME(InternalName, ReturnType, ExternalName, DocString, HelpContext )
```

- *InternalName* is the identifier string you assigned to the member in the automation declaration.
- *ReturnType* tells what automation data type the method returns or the property holds.
- *ExternalName* is what automation controllers see. A user sending commands from a controller refers to all properties and methods by their external names.
- *DocString* should explain to a user what the exposed property or method does. OLE displays this string if the user asks for help with a particular automation command. If you omit the document string, set the parameter to 0.
- *HelpContext*, the fifth parameter, is optional. It is a number that identifies a particular section of a Windows Help file (.HLP). You can create a Help file that describes the syntax and usage of all the members you expose. If you supply the context IDs for each member in the class's automation definition, then an automation controller can ask OLE to display the help screens for the user. A user writing an automation script, for example, can browse at run time for the list of members your application exposes, ask to see their document strings, and even ask to see a Help screen about each one.

If you provide a Help file for automation, you should be sure to register its name with the *typehelp* key.

When exposing a method that takes arguments, you also need to add to the definition a macro describing each argument. Here is the prototype for a function that takes three arguments, along with the macros needed to define the method for automation:

```
// member function declaration
long TCalculator::AddNumbers(short Num1, short Num2 = 0, short Num3 = 0);


// later, this appears after DEFINE_AUTOCLASS(TCalculator)
EXPOSE_METHOD(AddNumbers, TAutoLong, "AddNumbers", "Sum up to 3 numbers",
  HC_ADDNUMBERS)
  REQUIRED_ARG(TAutoShort, "Num1")
  OPTIONAL_ARG(TAutoShort, "Num2", "0")
  OPTIONAL_ARG(TAutoShort, "Num3", "0")
```

The first argument, *Num1*, is required. The others are optional. All three are **short** integers. When describing optional arguments, you need to supply a default value. In the example, 0 is the default value for the two optional arguments.

OLE conventions suggest that each automation object should have a property representing the application it belongs to. You can add this property to any automation definition with the EXPOSE_APPLICATION macro.

```
EXPOSE_APPLICATION(TMyClass, "Application", "My Application",)
```

The class passed to EXPOSE_APPLICATION must be the same class passed to the factory template.

See Also

Automation Data Types

Automation Definition Macros

Creating a Registrar Object

EXPOSE_APPLICATION Macro

Data Type Specifiers in an Automation Definition

See Also Defining External Methods and Properties

Most of the macros in an automation definition ask for a data type - the type of a function's return value, of each function argument, or of a data member. The possible values for data types within an automation definition are not fundamental C types. They can be any of the following:

- An enumeration value previously defined for automation.
- The name of an automated class (such as *TCalc*).
- Any of the predefined classes that ObjectComponents provides to represent intrinsic C types. S

The reason for exposing predefined classes rather than intrinsic C types is to make type information available when browsing from the controller. For exposed classes, ObjectComponents can extract type information using RTTI. The automation data types in the following table are defined as structures that contain no data; they simply retrieve a static value indicating a data type. The identifier values are the same identifiers that OLE uses to distinguish the data types it supports. All the automation data types derive from a base called *TAutoVal*, so they are polymorphic. In effect, ObjectComponents can ask any value passed through automation to describe its own data type. For more information, se the table provided in Declarations and Definitions of Automation Data Types.

See Also

Automation Declaration Macros

Automation Definition Macros

TAutoBool

TAutoCurrency

TAutoCurrencyRef

TAutoDate

TAutoDateRef

TAutoDouble

TAutoDoubleRef

TAutoFloat

TAutoFloatRef

TAutoInt

TAutoLong

TAutoLongRef

TAutoObject

TAutoObjectDelete

TAutoObjectByVal

TAutoShort

TAutoShortRef

TAutoString

TAutoVoid

TAutoVal

Exposing Data for Enumeration

See Also Defining External Methods and Properties

An automation server might also need to expose enumerated values. Use OLE enumerations when you want to expose a set of internal data values and refer to them with localizable strings. For example, AutoCalc defines the enumerated type operators to represent different actions the calculator can perform with numbers.

```
enum operators {
  OP_NONE = 0,
  OP_PLUS,
  OP_MINUS,
  OP_MULT,
  OP_DIV,
  OP_EQUALS,
  OP_CLEAR,
};
```

As the calculator receives input, it stores the pending mathematical operation in a private data member called *Op*.

```
short Op;
```

Operations are identified by different OP_*xxxx* constants. The *Eval* method performs the pending operation using the number just entered and the total in the calculator's accumulator. AutoCalc exposes the *Op* data member to automation so that a controller can enter operators directly. Here's the automation declaration:

```
AUTODATA(Op, Op, short, AUTOVALIDATE(Val>=OP_NONE && Val<=OP_CLEAR))
```

The automation declaration shows that the *Op* data member holds a short value, but the symbols OP_PLUS and OP_MINUS are defined only within the server program. The controller can't use them when it passes commands. Ideally the controller should be able to use more readable strings such as "Add" and "Subtract" in scripts.

The place for declaring public symbols is the automation definition. Use the DEFINE_AUTOENUM macro to begin a table defining symbols for the enumerated values.

```
DEFINE_AUTOENUM(CalcOps, TAutoShort)
  AUTOENUM("Add",      OP_PLUS)
  AUTOENUM("Subtract", OP_MINUS)
  AUTOENUM("Multiply", OP_MULT)
  AUTOENUM("Divide",   OP_DIV)
  AUTOENUM("Equals",   OP_EQUALS)
  AUTOENUM("Clear",    OP_CLEAR)
END_AUTOENUM(CalcOps, TAutoShort)
```

The AUTOENUM macro takes two parameters: an enumeration string and a constant value. The enumeration string (which can be localized) is the external name exposed through OLE for use by controllers.

The macros that begin and end the enumeration table assign the name *CalcOps* to this enumerated type. They also associate the automated data type *TAutoShort* with this enumeration because the enumerated values are all **short ints**.

The following table lists the C++ types that can be enumerated and the corresponding automation types for exposing them.

C++ type	Enumeration type (for automation definitions)
bool	TAutoBool
double	TAutoDoubl
float	TAutoFloat

int	TAutoInt
long	TAutoLong
short	TAutoShort
const char*	TAutoString

Creating a table of enumerated values results in a new data type that you can use to describe arguments and return values in an automation definition. Now that ObjectComponents understands the *CalcOps* enumerated type, you can use the type to define the *Op* property.

```
EXPOSE_PROPRW(Op, CalcOps, "Op", "@Op_", HC_TCALC_OPERATOR)
```

This line says that *Op* is a read-write property holding a value of type *CalcOps*. When the controller tries to place "Multiply" or "Divide" in the *Ops* property, ObjectComponents correctly translates the string into the value defined as OP_MULT or OP_DIV.

See Also
Automation Declaration Macros
Automation Definition Macros
TAutoShort

Building the Server

See Also

To build an automation server, you need to:

- Include header files
- Link to the right libraries and compile

See Also
Automating a Class
Enhancing Automation Server Functions
Registering the Application.

Including Header Files

See Also Building The Server

An automated program needs to include the following headers:

```
#include <ocf/automacr.h>   // definition and declaration macros
#include <ocf/ocreg.h>      // TRegistrar class
```

The list is short because an automation server does not need many of the ObjectComponents classes used for linking and embedding.

See Also
Compiling and Linking

Compiling and Linking

See Also Building The Server

Automation servers and controllers must be compiled with the medium or large memory model. (They run faster in medium model.) They must be linked with the OLE and ObjectComponents libraries.

The IDE chooses the right build options for you when you ask for OLE support. To build any ObjectComponents program from the command line, create a short makefile that includes the OCFMAKE.GEN file found in the EXAMPLES subdirectory.

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj myprogram.obj
!include $(BCEXAMPLEDIR)\ocfmake.gen
```

EXERES and OBJRES hold the name of the file to build and the names of the object files to build it from. The last line includes the OCFMAKE.GEN file. Name your file MAKEFILE and type this at the command line prompt:

```
make MODEL=l
```

MAKE, using instructions in OCFMAKE.GEN, will build a new makefile tailored to your project. The new makefile is called WIN16Lxx.MAK.

Note: The first time the server runs, the registrar object records its information in the registration database. Be sure to run the server once before trying to use it with a controller.

See Also
Including Header Files

Enhancing Automation Server Functions

See Also

There are many ways to enhance a server's capabilities.

- Combine Multiple C++ Objects Into a Single OLE Automation Object
- Tell OLE When The Object Goes Away
- Localize Symbol Names and registration entries
- Expose Collections of Objects
- Create a Type Library

See Also
Automating a Class
Building the Server
Registering An Automation Server

Combining Multiple C++ Objects Into a Single OLE Automation Object

See Also Enhancing Automation Server Functions

The complete set of member functions and properties that belong to a single automatable OLE object can in fact be implemented by a combination of C++ objects. An automatable calendar, for example, might begin with a *TCalendar* class. But the automatable OLE calendar object might need to expose some methods and properties that don't happen to belong to the C++ *TCalendar* object. The background color, for example, might be inherited from *TCalendar*'s base class, and some of the input functions might belong to separate control windows in the calendar's client area. In that case, the automation declaration for *TCalendar* should delegate some tasks to other C++ classes. To combine several C++ objects together into a single OLE object, add macros to the automation definition table.

```
// these lines belong in the definition block that begins
  DEFINE_AUTOCLASS(TCalendar)
    EXPOSE_INHERIT(TCalendarWindow, "CalendarWindow")
    EXPOSE_DELEGATE(TWeekForwardButton, "WeekForward",
      GetWeekForwardButton(this))
```

Any exposed classes must also be automated. In other words, *TCalendarWindow* and *TWeekForwardButton* must also have their own AUTOCLASS tables. By exposing both of these classes in the *TCalendar* automation definition, you combine all the exposed members from all three classes into a single symbol table. When OLE sends an automation command to the calendar, ObjectComponents searches for the matching class member in *TCalendar*, then in *TCalendarWindow*, and finally in *TWeekForwardButton*.

The EXPOSE_DELEGATE macro takes as its third parameter a conversion function. In order to reach members in the delegation class, ObjectComponents needs a pointer to an object of that class. The conversion function has one parameter for receiving a this pointer to the object where the definition table appears. The function must return a pointer to the delegation object. Also, it must be a global function. For example, if *TCalendar* has a data member that points to the Week Forward button, this might be the conversion function.

```
TWeekForwardButton *GetWeekForwardButton ( TCalendar* this ) {
  return( this->m_ForwardButton );
}
```

You don't need to provide a conversion function when exposing an inherited function or property because in that case ObjectComponents can create its own templatized conversion function to reach the base class.

Note: Another way to coordinate the actions of several automated objects within a single application is to give one object access functions that return the other objects. For example, the sample program AutoCalc automates five different classes, but no class delegates to any other. When a controller asks for an object from the AutoCalc server, it receives only the automated *TCalc* object. *TCalc*, however, has a property called Window that holds a *TCalcWindow* object. *TCalcWindow*, in turn, has a property that holds the collection of buttons. The collection object returns individual button objects. Without properties or functions that return the other objects, the controller would never be able to reach them. Be sure to add access functions if necessary.

See Also

Automation Definition Macros

EXPOSE_DELEGATE Macro

EXPOSE_INHERIT Macro

Telling OLE When the Object Goes Away

See Also Enhancing Automation Server Functions

If there is a chance that your program might delete its automated object while still connected to a controller, then you need to tell OLE when the object is destroyed. This precaution matters only if the logic of your program might cause the object to be destroyed through nonautomated means while an OLE session is still in progress. If OLE attempts to use an automation object whose underlying C++ object has been destroyed, it attempts to use an invalid pointer. A single function call prevents the error by sending OLE an obituary to announce that the object no longer exists.

```
// place this line in the destructor of your automated class
::GetAppDescriptor()->InvalidateObject(this);
```

GetAppDescriptor is a global function returning a pointer the application's *TAppDescriptor* object. *InvalidateObject* is a *TAppDescriptor* method. It tells OLE the object that was passed to the descriptor's constructor is now invalid.

Although the object's destructor is a good place to call *InvalidateObject*, you can call it anywhere. If you do not own the class you are automating, it might not be possible to modify the destructor. This works, too:

```
TMyAutoClass MyAutomatedObject;
.
.
.
::GetAppDescriptor()->InvalidateObject(MyAutomatedObject);
delete MyAutomatedObject;
```

The object pointer you pass to *InvalidateObject* must always represent the most derived form of the object. In other words, if the pointer is polymorphic, it must point to the class as it was created and not to any of its base classes. Calling *InvalidateObject* from the object's own destructor is safe because in that case **this** always points to the most derived class. If you call *InvalidateObject* from somewhere else, you might need the global function *MostDerived* to ensure that you are invalidating the correct object.

```
appDesc->InvalidateObject(::MostDerived(MyPolymorphObject,
  typeid(MyPolymorphObject)));
```

In the example, *MyPolymorphObject* is a pointer to a polymorphic object, so it might point to a base class or to an object of any type derived from the base. *MostDerived* converts the pointer, making it point to an object of the type furthest down the hierarchy, the one furthest descended from the base.

Besides calling *InvalidateObject*, there are two other ways to be sure OLE knows when the object is destroyed. One way is to derive the object's class from *TAutoBase*. The only code in *TAutoBase* is a virtual destructor that calls *InvalidateObject* for you. This example declares a class called *TMyAutoClass*. OLE always knows when any object of type *TMyAutoClass* is destroyed.

```
class TMyAutoClass: public TAutoBase { /* declarations */ };
```

The other way is to put the AUTODETACH macro in the class's automation declaration table. This works without having to change the class derivation, but it does add one byte to the size of the class.

See Also

AUTODETACH macro

MostDerived function

TAutoBase

TOcRegistrar::GetAppDescriptor

Localizing Symbol Names

See Also Enhancing Automation Server Functions

The symbols that appear in an automation definition become visible to other OLE programs. Users writing scripts can see and use the symbols. The symbols become part of the program's user interface. Programs intended to reach international audiences need to translate the strings for different markets. For example, a property named "Color" in English should be called "Couleur" in a French script, "Farbe" in a German script, and "Colour" in a British one.

To localize symbols:

1. Put Translations in the Resource Script
2. Mark Translatable Strings in the Source Code

These two topics give more information about how ObjectComponents implements localization.

- Understand How ObjectComponents Uses XLAT Resources
- Localize Registration Strings

OLE does its best to help you out by passing a number that indicates the user's language setting. This number is called a locale ID, or LCID. LCIDs are defined by OLE and the Win32 API. They consist of two numbers, one identifying a language and one identifying a subdialect within the language. When OLE passes an automation call into an automated application, it also passes an LCID. The automation controller might determine the LCID from the system settings at run time, or the person using the controller might choose a locale.

An automated program is expected to examine the LCID and respond with appropriately translated strings. ObjectComponents eases the burden by letting you build a resource table to supply localized versions of any strings you use. When handling automation calls, ObjectComponents automatically searches the table to find strings that match whatever language the controller requests.

ObjectComponents searches first for a string with the correct language and dialect IDs. Failing that, ObjectComponents searches for a match on primary language only, ignoring dialect. If still no match is found, ObjectComponents simply uses the original, untranslated string.

See Also
Langxxxx ID constants (OWL.HLP)

Putting Translations in the Resource Script

See Also Localizing Symbol Names

To build a table of translations in your resource (.RC) file, use the XLAT resource type.

```
#include "owl/locale.rh"
Left    XLAT FRENCH "Gauche" GERMAN "Links" SPANISH "Izquierda" XEND
Right   XLAT FRENCH "Droit"  GERMAN DUTCH "Rechts"              XEND
Center  XLAT ENGLISH_UK FRENCH GERMAN "Centre" SPANISH "Centro" XEND
Help    XLAT FRENCH "Aide" GERMAN "Hilfe" SPANISH "Ayuda"       XEND
```

The locale.rh header file defines XLAT as a type of resource. XLAT and XEND are delimiters for all the translations of a single string. The same header also defines macros to represent various locale IDs. FRENCH, DUTCH, and ENGLISH_UK, for example, each represent a different LCID. UK is a subdialect of ENGLISH.

Each line in the localization table begins with a resource identifier. These examples use the original string itself to identify the resource that holds its translations.

A localization table is not obliged to provide the same set of translations for each string. For example, it is legal to provide FRENCH_FRANCE, FRENCH_BELGIUM, and SWEDISH for one string, but only FRENCH and ITALIAN for the next string. Also, if several languages happen to use the same string, it is legal to write the string only once, as in this example:

```
Center XLAT ENGLISH_UK FRENCH GERMAN "Centre" SPANISH "Centro" XEND
```

In British English, French, and German, "Center" is translated as "Centre." in Spanish, it becomes "Centro." Writing "Centre" only once makes the .EXE file smaller.

See Also
Marking Translatable Strings in the Source Code

Marking Translatable Strings in the Source Code

See Also Localizing Symbol Names

Composing a resource table is the first step, but ObjectComponents still needs to be told when to use the table you have provided. In the automation definition, mark each translatable string by prefixing it with an exclamation point.

```
EXPOSE_METHOD(Clear, TAutoVoid, "!Clear", "Clear accumulator",
  HC_TCALC_CLEAR)
```

This line from *AutoCalc* exposes a class method named *Clear*. *Clear* returns **void**. The third parameter, *!Clear*, gives the external name that controllers see. The initial exclamation point tells ObjectComponents to look in the program's executable file for a localization resource whose identifier is the string *Clear*.

```
Clear XLAT GERMAN "AllesLöschen" XEND
```

The exclamation point prefix also marks *Clear* as the language-neutral form of the string. If an automation controller decides to use the locale ID GERMAN, then ObjectComponents tells it that the exposed property is called *AllesLöschen*. If the controller sets any other locale ID, it receives the neutral form, *Clear*.

Argument names as well as properties and methods can be localized.

```
EXPOSE_METHOD(Button, TAutoBool, "!Button", "Button push sequence",
  HC_TCALC_BUTTON)
   REQUIRED_ARG( TAutoString, "!Key")
```

in determining what to call both the *Button* method and its one argument, ObjectComponents will search the program's localization resources for *Button* and Key.

```
Button        XLAT GERMAN "Schaltfläche" XEND
Key           XLAT GERMAN "Taste" XEND
```

The algorithm that searches for resources is not sensitive to case, and the current implementation of 16-bit Windows does not allow the use of extended characters (such as characters with diacritical marks) in resource names. The strings stored in a resource, however, can use any characters and do preserve their case.

A problem arises in naming your resource if the string contains spaces. Resource identifier strings cannot have spaces. Consider what happens if you try to localize the description string for this property:

```
// illegal: no spaces allowed in resource identifiers
EXPOSE_PROPRW(Caption, TAutoString, "!Caption", "!Window Title",
  HC_TCALCWINDOW_TITLE)
```

It's a good idea to localize descriptions as well as property names, but "Window title" is not a legal resource identifier. In cases like this, use @ instead of ! as the localization prefix, and follow it with any legal identifier.

```
EXPOSE_PROPRW(Caption, TAutoString, "!Caption",
  "@Caption_",HC_TCALCWINDOW_TITLE)
```

The @ prefix tells ObjectComponents that the string is *only* a resource identifier and should never be displayed no matter what locale the controller requests. To make the distinction even clearer for programmers reading the code, strings used only as identifiers conventionally end with an underscore, as in *Caption_* .

To make "Window Title" the language-neutral string, do not assign it a locale ID in the localization resource.

```
Caption_ XLAT "Window Title" GERMAN "Fenster-Aufschrift" XEND
```

Now a controller that requests any locale setting other than GERMAN is given the string *Window Title*.

Besides ! and @, there is a third localization prefix: #. The # prefix must be followed by digits that identify a localization resource by number.

```
EXPOSE_PROPRW(Caption, TAutoString, "!Caption",
  "#10047",HC_TCALCWINDOW_TITLE)
```

This example tells ObjectComponents to look for a resource numbered 10047. This is how the resource should appear in the .RC file:

```
10047 XLAT "Window Title" GERMAN "Fenster-Aufschrift" XEND
```

See Also

Automation Definition Macros

Putting Translations in the Resource Script

Understanding How Object Components Uses XLAT Resources

Understanding How ObjectComponents Uses XLAT Resources

See Also Localizing Symbol Names

The external names in macros like EXPOSE_METHOD and EXPOSE_PROPRW are wrapped in objects of type *TLocaleString*, a localizable substitute for **char*** strings. A *TLocaleString* object contains code that searches a program's executable file for XLAT resources. All access to the XLAT resources is performed by *TLocaleString*.

The *TLocaleString* class is defined in osl/locale.h. You don't need to refer to *TLocaleString* directly. The macros and headers bring it in for you.

TLocaleString is very efficient. If the controller is working in the server's native language, then *TLocaleString* realizes the strings in the source code already match the locale and it doesn't waste any time reading resources.

Usually ObjectComponents determines the application's default language by reading the system's locale ID at compile time and storing it in the compiled program. You can override the default by including a line like this in your source code.

```
#include "olenls.h"
TLangId
  TLocaleString::NativeLangId=MAKELANGID(LANG_ENGLISH,SUBLANG_ENGLISH_US);
```

The olenls.h header holds national language support constants, including the MAKELANGID macro and the language and dialect symbols.

When it must resort to resources, *TLocaleString* does everything it can to minimize the time spent searching for translations. When it finds a string to match the current locale, it caches the string in memory and never has to load it again. That means only the first attempt to use each translated string incurs a performance hit. Subsequent requests are satisfied quickly. Once in memory, the strings are stored in a hash table so no space is wasted on duplicates. If *TLocaleString* fails to find a requested string, it remembers the failure as well and won't try to find the same string a second time.

See Also

Localizing Registration Strings

Putting Translations in the Resource Script

TLangId (OWL.HLP)

Localizing Registration Strings

See Also Localizing Symbol Names

The same localization mechanism works with strings your application registers. Some strings, such as the *progid*, should not be localized, but the following list names registration keys that can be localized.

- appname
- debugdesc
- description
- format*n*
- menuname
- permname
- typehelp
- verb*n*

The following excerpt from the AutoCalc registration tables shows where to put the localization prefixes. The *appname*, *description* and *typehelp* keys are localized.

```
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,       "{877B6200-7627-101B-B87C-0000C057CE4E}")
  REGDATA(progid,      APP_NAME ".Application")
  REGDATA(appname,     "@AppName")
  REGDATA(description,"@Desc")
  REGDATA(typehelp,    "@typehelp")
  REGDATA(version,     "1.2")
END_REGISTRATION
```

AutoCalc supplies translations for the *appname*, *description*, and *typehelp* strings in its resource script. Here are two of them.

```
Desc          XLAT "Automated Calculator 1.2 Application"
            GERMAN "Automatisierte Taschenrechner-Anwendung 1.2"
              XEND
Typehelp      XLAT "autocalc.hlp"
            GERMAN "acalcger.hlp"
              XEND
```

ObjectComponents determines the proper language for registration by examining the system settings at run time, but it is possible to override the system setting with the Language command-line switch.

See Also
Putting Translations in the Resource Script
Registration Keys
Registration Macros (OWL.HLP)

Exposing Collections of Objects

ObjectComponents lets an automated object expose collections of various types as object properties. The items in a collection can belong to an array, a linked list, or any other structure that organizes sets of similar items.

Exposing collections for automation

- Constructand Expose a Collection Class - How you do this depends on the constructor of the class that manages the collection.
- Implement an Iterator for the Collection
- Add Other Members to the Collection Class

To expose a collection, you need to expose methods for manipulating it. These methods typically include a counter to show the size of the collection, an iterator to walk through the collection, and a random-access function to retrieve specific items in the collection.

A collection object is an object that returns on request individual items from a set of related items. It implements the methods that manipulate the items. In the AutoCalc sample program, the buttons on the face of the calculator are a set of related, similar objects. AutoCalc defines a new class, *TCalcButtons*, whose methods let a controller ask for individual button objects. The buttons themselves are automated objects, so once a controller receives a button it can send a push command or change the text the button displays.

Constructing and Exposing a Collection Class

See Also Exposing Collections of Objects

If you are converting an application to support automation, you are likely to find that it does not already have a C++ class to act as a collection object. The items might be simple values, structures, or even system objects represented by handles. You have to create a new C++ class, and you have to expose the class in the parent's automation tables as a property of the parent class. How you expose the collection in the parent's automation declaration table depends on what information the parent passes the collection object to construct it. This section considers several different possible constructors and shows the macros for adding the collection as a property of its parent.

Instances of the collection class are constructed only when a controller requests it. The collection object appears to the controller as a property of the parent class. In AutoCalc, for example, when a controller asks for what is in the *Buttons* property, ObjectComponents creates a *TCalcButtons* object on the fly. The constructor of a collection object must accept a single argument passed from the parent to initialize itself.

Because *TCalcButtons* manages a collection of child windows, its parent passes the handle of the parent window. The constructor looks like this:

```
TCalcButtons(HWND window) : HWnd(window) {}
```

For the handle to be passed to the constructor, the parent must add a line to its automation declaration:

```
// from the automation declaration of the parent class
DECLARE_AUTOCLASS(TCalcWindow)
  AUTODATARO(Buttons, hWnd, TAutoObjectByVal<TCalcButtons>,)
```

Buttons is assigned as the internal name of a read-only property whose value is *TCalcWindow::hWnd*. For the data type of this property, the table specifies a new class based on the collection class. *TAutoObjectByVal*<*T*> causes an instance of *T* to be constructed that persists until all external references to that instance are released (when the exposed object goes out of scope in the automation controller).

TCalcWindow must also expose the collection property in its automation definition:

```
// from the automation definition of the parent class
DEFINE_AUTOCLASS(TCalcWindow)
  EXPOSE_PROPRO(Buttons, TCalcButtons, "!Buttons", "@Buttons_",
  HC_TCALCWINDOW_BUTTONS)
```

When a controller asks for what is stored in the read-only property called *Buttons*, ObjectComponents creates a *TCalcButtons* object and passes *hWnd* to its constructor.

Other Ways To Expose a Collection Object

Here are three examples showing other ways a parent class might expose a collection object as one of its properties:

- **Case 1**: *TParent::DocList* points to the head of a linked list of *TDocument* objects. A new class, *TDocumentList*, is created as the collection object. The constructor of *TDocumentList* receives from its parent the head of the linked list:

  ```
  TDocumentList(TDocument*);
  ```

  The automation declaration of *TParent* exposes *DocList* as a read-only property, using the collection class to assign it a type.

  ```
  DECLARE_AUTOCLASS(TParent)
    AUTODATARO(Documents, DocList, TAutoObjectByVal<TDocumentList>,)
  ```

  The automation definition of *TParent* calls the collection *Documents* and says its type is *TDocumentList*.

```
DEFINE_AUTOCLASS(TParent)
  EXPOSE_PROPRO(Documents, TDocumentList, "Documents", "Doc Collection",
 270)
```

- **Case 2**: *TParent* contains a list. It passes **this** to the collection object, *TList*, which extracts list items by indirection through the parent's pointer. The constructor receives the pointer.

```
TList(TParent* owner)
```

The automation declaration of *TParent* exposes **this** as a read-only property, using the collection class to assign it a type.

```
DECLARE_AUTOCLASS(TParent)
  AUTOTHIS(List, TAutoObjectByVal<TList>,)
```

The automation definition of *TParent* calls the collection *List* and says its type is *TList*.

```
DEFINE_AUTOCLASS(TParent)
  EXPOSE_PROPRO(List, TList, "List", "List of items", 240)
```

- **Case 3**: *Elem* is an array of integers, defined as **short** *Elem[COUNT]*. The collection object is *TMyArray*, and the constructor receives from the parent a pointer to *Elem*.

```
TMyArray(short* array)
```

The automation declaration of *TParent* exposes *Elem* as a read-only property, using the collection class to assign it a type.

```
DECLARE_AUTOCLASS(TParent)
  AUTODATARO(MyArray, Elem, TAutoObjectByVal<TMyArray>,)
```

The automation definition of *TParent* calls the collection *Array* and says its type is *TMyArray*.

```
DEFINE_AUTOCLASS(TParent)
  EXPOSE_PROPRO(MyArray, TMyArray, "Array", "Array as collection", 110)
```

See Also
Automation Declaration Macros
Automation Definition Macros
Implementing an Iterator for the Collection
TAutoObjectByVal

Implementing an Iterator for the Collection

See Also Exposing Collections of Objects

The collection class performs whatever actions you want a controller to be able to perform with the collection. Common collection methods include *Count* and *GetObject*, which return the number of items in the collection or individual items specified by number. The only methods you need to implement, however, are the constructor and an iterator. You have already seen the constructor. An iterator function walks through the collection and returns successive items on each new call.

The easy way to define an iterator is with the AUTOITERATOR macro, which you add to the declaration table of the collection object.

```
DECLARE_AUTOCLASS(TCalcButtons)
  AUTOITERATOR(int Id, Id = IDC_FIRSTID+1, Id <= IDC_LASTID, Id++,
                TAutoObjectByVal<TCalcButton>(::GetDlgItem(This->HWnd,Id)))
```

The parameters to AUTOITERATOR define the algorithm for enumerating objects in the collection. Each of the five macro arguments represents a code fragment, ordered as in a **for** loop.

1. Declare state variables for keeping track of loop iterations. For example,

```
int Index;
```

2. Assign initial values to the state variables. For example,

```
Index = 0;
```

3. Test a Boolean expression to decide whether to enter the loop. For example,

```
Index < This->Total
```

4. Modify state variables to prepare for the next iteration. For example,

```
Index++;
```

5. Retrieve one item from the collection. For example,

```
(This->Array)[Index];
```

Note that the server can return any data type for itemsvalues or objects.

In the AUTOITERATOR parameters, do not use commas except inside parentheses. Semicolons can separate multiple statements, but cannot be used to end a macro argument. As in automated methods, *This* is defined to be the **this** pointer of the enclosing C++ class (here, the collection itself).

AUTOITERATOR puts an iterator in the automation declaration table, but the iterator member must still be exposed in the definition table. Use the EXPOSE_ITERATOR macro.

```
EXPOSE_ITERATOR(TAutoShort, "Array Iterator", HC_ARRAY_ITERATOR)
```

EXPOSE_ITERATOR takes fewer parameters than other EXPOSE_xxxx macros do. No internal or external names are supplied. A class can have only one iterator, and the external name is always *_NewEnum*. The first parameter describes the type of the items returned from the iterator.

The automation type describes the type of the items returned from the iterator, in the same manner as a function return. The previous example iterates an array of **short int** values, so its automation data type is *TAutoShort*. The second parameter is the documentation string describing the iterator property, and the third parameter, which is optional, identifies a context in an .HLP file for more information about the iterator.

Note: From the external side, a script controller sees the enumerator as a property with the reserved name *_NewEnum* that returns an object supporting the standard OLE interface *IEnumVARIANT*. This interface contains methods to perform iteration. A controller makes use of an iterator in a loop like this one, which is written in Visual Basic for Applications:

```
For Each Thing in Owner.Bunch   ("Thing" is an arbitrary iterator name)
  Thing.Member......            (can access methods and properties)
  Next Thing                    (loops through all items in collection)
```

Note: The AUTOITERATOR macro generates a nested class definition within the collection class. For complex iterators, you can choose to code the iterator explicitly in C++. Here is an example:

```
class TIterator : public TAutoIterator {
  public:
    ThisClass* This;
    /* declare state variables here as members */
    void Init() {/* loop initialization function body */}
    bool Test() {/* loop entry test function body */}
    void Step() {/* loop iteration function body;}
    void Return(TAutoVal& v) {/* current element return: v = expr */}
    TIterator* Copy() {return new TIterator(*this);}
    TIterator(ThisClass* obj, TServedObject& owner)
                  : This(obj), TAutoIterator(owner) {}
    static TAutoIterator* Build(ObjectPtr obj, TServedObject& owner)
    { return new TIterator((ThisClass*)obj, owner); }
};
friend class TIterator;    // make iterator a friend of the surrounding
  collection class
```

See Also
Adding Other Members to the Collection Class
AUTOITERATOR Macros
Constructing and Exposing a Collection Class
EXPOSE_ITERATOR Macro
TAutoShort

Adding Other Members to the Collection Class

See Also Exposing Collections of Objects

In addition to exposing an iterator, a collection class by convention exposes a *Count* method to return the number of items in the collection, an *Index* method for random access to members of the collection, and optionally, methods such as *Add* and *Delete* to manage the collection externally. Here, for example, is the complete code for the *TCalcButtons* collection class in AutoCalc:

```
class TCalcButtons {    // class used only temporarily to expose collection
  public:
    TCalcButtons(HWND window) : HWnd(window) {}
    short GetCount() { return IDC_LASTID - IDC_FIRSTID; }
    HWND GetButton(short i) {return ::GetDlgItem(HWnd, i + IDC_FIRSTID+1);}
    HWND HWnd;
  DECLARE_AUTOCLASS(TCalcButtons)
    AUTOFUNC0 (Count, GetCount, short,)
    AUTOFUNC1 (Item,  GetButton, TAutoObjectByVal<TCalcButton>, short,
                           AUTOVALIDATE(Arg1 >= 0 && Arg1 < This->GetCount())
)
    AUTOITERATOR(int Id, Id = IDC_FIRSTID+1, Id <= IDC_LASTID, Id++,
                 TAutoObjectByVal<TCalcButton>(::GetDlgItem(This->HWnd,Id)))
};

DEFINE_AUTOCLASS(TCalcButtons)
  EXPOSE_PROPRO(Count, TAutoLong, "!Count", "@CountBu_",
HC_TCALCBUTTONS_COUNT)
  EXPOSE_ITERATOR(TCalcButton, "Button Iterator", HC_TCALCBUTTONS_ITERATOR)
  EXPOSE_METHOD_ID(0, Item, TCalcButton,"!Item",  "@ItemBu_", 0)
    REQUIRED_ARG(TAutoShort, "!Index")
END_AUTOCLASS(TCalcButtons, tfNormal, "TButtonList", "@TCalcButtons",
HC_TCALCBUTTONS)
```

See Also
Automation Declaration Macros
Automation Definition Macros
Constructing and Exposing a Collection Class
Implementing an Iterator for the Collection

Creating a Type Library

See Also Enhancing Automation Server Functions

A type library is a binary file containing information about an automation server. The information describes the objects, properties, and methods the server supports. It is used by programming tools, such as automation controllers, that call the server. Controllers can query the type library for documentation and help with specific objects. The location of its type library is one of the pieces of information an automation server records in the system's registration database.

ObjectComponents can create a type library for you from information in the server's automation definitions. To make a type library, call the server and set the TypeLib switch on the command line.

```
myapp -TypeLib
```

This command causes ObjectComponents to create a new file, MYAPP.OLB, in the same directory as MYAPP.EXE. ObjectComponents also records the library's location in the registration database.

The TypeLib flag also accepts an optional path and file name.

```
myapp -TypeLib = data\mytyplib
```

ObjectComponents places MYTYPLIB.OLB in a subdirectory called DATA under the directory where MYAPP.EXE resides.

You can also make ObjectComponents generate mutliple type libraries in different languages with the Language switch. This command produces two type libraries, one in German and one in Italian.

```
myapp -Language=10 -TypeLib=italiano -Language=7 -TypeLib=deutsch
```

The number passed to Language must be hexadecimal digits. The Win32 API defines *80C* as the locale ID for the Belgian dialect of the French language. For this command line to have the effect you want, of course, *myapp* must supply Belgian French strings in its XLAT resources.

You can also create an .HLP file of online Help to accompany your type library. The Help file documents all the commands the server exposes, explaining what arguments they expect and how to use them. If you have a Help file, be sure to register it using the *typehelp* and *helpdir* registration keys Use the final parameter of the EXPOSE_xxxx macros in the automation definition table to associate Help context IDs with each command. If the automation controller asks for help on a command, OLE launches the Help file automatically.

See Also

Automating a Class

Buildingthe Server

Enhancing Automation Server Functions

EXPOSE_xxxx Macros

Exposing Collections of Objects

Localizing Symbol Names

Processing the Command Line

Registering An Automation Server

Creating an OLE Container

See Also

OLE container is an application that can store in its own documents data objects taken from other applications. A container can link objects or embed them in its OLE container is an application that can store in its own documents data objects taken documents. A program that creates objects to be linked or embedded is called a *server*.

The following topics explain how to turn existing programs into OLE containers.

- Turning a Doc/View Application Into an OLE Container
- Turning an Objectwindows Application Into an OLE Container
- Turning a C++ Application Into an OLE Container

See Also
Automating an Application

Turning a Doc/View Application Into an OLE Container

See Also

Turning a Doc/View application into an OLE container requires only a few modifications.

1. Connect Objects to OLE
2. Register the Container
3. Support OLE Commands
4. Build the Container

That's all you need to do. By following these steps, you can create an OLE container that supports all the following features:

- Linking
- Embedding
- OLE clipboard operations
- Drag and drop operations
- In-place editing
- Tool bar and menu merging
- Compound document storage

You also get standard OLE 2 user interface features, such as object verbs on the Edit menu, the Insert Object dialog box, and a pop-up menu that appears when the user right-clicks an embedded object.

ObjectComponents provides default behavior for all these common OLE features. Should you want to modify the default behavior, you can additionally choose to override the default event handlers for messages that ObjectComponents sends. The code examples in this section are based on the STEP14.CPP and STEP14DV.CPP sample programs in EXAMPLES/OWL/TUTORIAL. Look there for a complete working program that incorporates all the prescribed steps.

See Also

Turning a C++ Application Into an OLE Container

Turning an Objectwindows Application Into an OLE Container

Connecting Objects to OLE

See Also Turning a Doc/View Application Into an OLE Container

Your application, window, document, and view objects need to make use of new OLE-enabled classes. The constructor for the application object expects to receive an application dictionary object, so create that first.

The following tasks are necessary when connecting objects to OLE.

- Deriving the Application Object from TOcModule
- Inheriting from OLE Classes
- Creating an Application Dictionary

See Also
Turning a Doc/View Application Into an OLE Server

Deriving the Application Object from TOcModule

See Also   Turning a Doc/View Application Into an OLE Container

The application object of an ObjectComponents program needs to derive from *TOcModule* as well as *TApplication*. *TOcModule* coordinates some basic housekeeping chores related to registration and memory management. It also connects your application to OLE. More specifically, *TOcModule* manages the connector object that implements COM interfaces on behalf of an application.

If the declaration of your application object looks like this:

```
class TMyApp : public TApplication {
  public:
    TMyApp() : TApplication(){};
   .
   .
   .
};
```

Then change it to look like this:

```
class TMyApp : public TApplication, public TOcModule {
  public:
    TMyApp(): TApplication(::AppReg["appname"], ::Module, &::AppDictionary)
{};
   .
   .
   .
};
```

The constructor for the revised *TMyApp* class takes three parameters.

- A string naming the application
  AppReg is the application's registration table. The expression *::AppReg["appname"]* extracts a string that was registered to describe the application.
- A pointer to the application module.
  Module is a global variable of type *TModule** defined by ObjectWindows.
- The address of the application dictionary.
  AppDictionary is the application dictionary object.

See Also
TApplication (OWL.HLP)
TOcModule

Inheriting from OLE Classes

See Also Turning a Doc/View Application Into an OLE Container

ObjectWindows includes classes that let windows, documents, and views interact with the ObjectComponents classes. The ObjectWindows OLE classes include default implementations for most normal OLE operations. To adapt an existing ObjectWindows program to OLE, change its derived classes so they inherit from the OLE classes. The following table shows which OLE class replaces each of the non-OLE classes.

Non-OLE class	OLE class
TFrameWindow	TOleFrame
TMDIFrame	TOleMDIFrame
TDecoratedFrame	TOleFrame
TDecoratedMDIFrame	TOleMDIFrame
TWindow	TOleWindow
TDocument	TOleDocument
TView	TOleView
TFileDocument	TOleDocument

The *TOleFrame* and *TOleMDIFrame* classes both derive from decorated window classes. The OLE 2 user interface requires containers to handle tool bars and status bars. Even if the container has no decorations, servers might need to display their own in the container's window. The OLE window classes handle those negotiations for you.

Wherever your existing OWL program uses a non-OLE class, replace it with an OLE class, as shown here. Boldface type highlights the change.

Before

```
// pre-OLE declaration of a window class
class TMyFrame: public TFrameWindow    { /* declarations */ );
```

After

```
// new declaration of the same window class
class TMyFrame: public TOleFrame     { /* declarations */ );
```

Note: If the implementation of your class makes direct calls to its base class, be sure to change the base class calls, as well. Response tables also refer to the base class and need to be updated.

See Also
TDocument (OWL.HLP)
TFrameWindow (OWL.HLP)
TOleDocument (OWL.HLP)
TOleFrame (OWL.HLP)
TOleMDIFrame (OWL.HLP)
TMDIFrame (OWL.HLP)
Turning a Doc/View Application Into an OLE Server
TWindow (OWL.HLP)

Creating an Application Dictionary

See Also Turning a Doc/View Application Into an OLE Container

An *application dictionary* tracks information for the currently active process. It is particularly useful for DLLs. When several processes use a DLL concurrently, the DLL must maintain multiple copies of the global, static, and dynamic variables that represent its current state in each process. for example, the DLL version of ObjectWindows maintains a dictionary that allows it to retrieve the *TApplication* corresponding to the currently active client process. If you convert an executable server to a DLL server, your application too must maintain a dictionary of the *TApplication* objects representing each of its container clients. If your DLL uses the DLL version of ObjectWindows, then your DLL needs its own dictionary and cannot use the one in ObjectWindows.

The DEFINE_APP_DICTIONARY macro provides a simple and unified way to create the application object for any application, whether it is a container or a server, a DLL or an EXE. Insert this statement with your other static variables:

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

For any application linked to the static version of the DLL, the macro simply creates a reference to the application dictionary in ObjectWindows. for DLL servers using the DLL version of ObjectWindows, however, it creates an instance of the *TAppDictionary* class.

Note: Name your dictionary object *AppDictionary* to take advantage of the factory templates such as *TOleDocViewFactory*.

See Also

DEFINE_APP_DICTIONARY macro (OWL.HLP)

Factory Template Classes (OWL.HLP)

Turning a Doc/View Application Into an OLE Server

Registering a Container

Turning a Doc/View Application Into an OLE Container

Registering your application with OLE involves the following steps:

- Building Registration Tables
- Understanding Registration Macros
- Creating a Registrar Object

Building Registration Tables

See Also		Registering a Container

OLE requires programs to identify themselves by registering unique identifiers and names. OLE also needs to know what Clipboard formats a program supports. Doc/ View applications also register their document file extensions and document flags. To accommodate the many new items an application might need to register, in ObjectWindows 2.5 you use macros to build structures to hold the items. Then you can pass the structure to the object that needs the information. The advantage of this method lies in the structure's flexibility. It can hold as many or as few items as you need.

Note: Previous versions of ObjectWindows passed some of the same information in parameters. Old code still works unchanged, but passing information in registration structures is the recommended method for all new applications.

A Doc/View OLE container fills one registration structure with information about the application and then creates another to describe each of its Doc/View pairs. The structure with application information is passed to the *TOcRegistrar* constructor. Document registration structures are passed to the document template constructor.

Here are the commands to register a typical container:

```
REGISTRATION_FORMAT_BUFFER(100)     // allow extra space for expanding
macros

BEGIN_REGISTRATION(AppReg) // information for the TOcRegistrar constructor
  REGDATA(clsid,      "{383882A1-8ABC-101B-A23B-CE4E85D07ED2}")
  REGDATA(appname,    "DrawPad Container")
END_REGISTRATION

BEGIN_REGISTRATION(DocReg)          // information for the document template
  REGDATA(progid,     "DrawPad.Document.14")
  REGDATA(description,"Drawing Pad (Step14--Container)")
  REGDATA(extension,  "p14")
  REGDATA(docfilter,  "*.p14")
  REGDOCFLAGS(dtAutoOpen | dtAutoDelete | dtUpdateDir | dtCreatePrompt |
dtRegisterExt)
  REGFORMAT(0, ocrEmbedSource,  ocrContent,  ocrIStorage, ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent,  ocrMfPict|ocrStaticMed, ocrGet)
  REGFORMAT(2, ocrBitmap, ocrContent,  ocrGDI|ocrStaticMed, ocrGet)
  REGFORMAT(3, ocrDib, ocrContent,  ocrHGlobal|ocrStaticMed, ocrGet)
  REGFORMAT(4, ocrLinkSource, ocrContent,  ocrIStream, ocrGet)
END_REGISTRATION
```

The registration macros build structures of type *TRegList*. Each entry in a registration structure contains a key, such as *clsid* or *progid*, and a value assigned to the key. Internally ObjectComponents finds the values by searching for the keys. The order in which the keys appear does not matter.

Insert the registration macros after your declaration of the application dictionary. Since the value of the *clsid* key must be a unique number identifying your application, it is recommended that you generated a new value using the GUIDGEN.EXE utility. (the *ObjectWindows Reference Guide* entry for *clsid* explains other ways to generate an identifer.) of course, modify the value of the *description* key to describe your container.

The example builds two structures, one named *AppReg* and one named *DocReg*. *AppReg* is an application registration structure and *DocReg* is a document registration structure. Both structures are built alike, but each contains a different set of keys and values. The keys in an application registration structure describe attributes of the application. A document registration structure describes the type of document an application can create. A document's attributes include the data formats that it can exchange with the clipboard, its file extensions, and its document type name.

The set of keys you place in a structure depends on what OLE capabilities you intend to support. The macros in the example show the minimum amount of information a container should provide.

The following table describes all the registration keys that a container can use. It shows which are optional and which required as well as which belong in the application registration table and which in the document registration table.

Key	in AppReg?	in DocReg?	Description
appname	Optional	No	Short name for the application
clsid	Yes	Optional	Globally unique identifier (GUID); generated automatically for the DocReg structure.
description	No	Yes	Descriptive string (up to 40 characters)
progid	No	Yes	Identifier for program or document type (unique string) **Note:** (Yes for a link source)
extension	No	Optional	Document file extension associated with server
docfilter	No	Yes	Wildcard file filter for File Open dialog box
docflags	No	Yes	Options for running the File Open dialog box
formatn	No	Yes	A clipboard format the container supports
directory	No	Optional	Default directory for storing document files
permid	No	Optional	Name string without version information
permname	No	Optional	Descriptive string without version information
version	Optional	No	Major and minor version numbers (defaults to "1.0")

The table shows what is required for container documents that support linking or embedding. For documents that support neither, the container needs to register only docflags and docfilter.

If your container is also a linking and embedding server or an automation server, then you should also consult the server table or the automation table. Register all the keys that are required in any of the tables that apply to your application.

The values assigned to keys can be translated to accommodate system language settings.

See Also
Localizing Symbol Names
Registering Localized Entries
TOcRegistrar
Understanding Registration Macros

Understanding Registration Macros

See Also Registering a Container

The first macro in the example, REGISTRATION_FORMAT_BUFFER, sets the size of a buffer needed temporarily as the macros that follow are expanded.

The REGDATA, REGFORMAT, and REGDOCFLAGS macros place items in the registration structure. All the registration macros are documented in the ObjectWindows Help (OWL.HLP).

REGDATA's first parameter is a key and the second is a value to associate with the key. in the example, the *AppReg* structure begins by assigning a value to the key *clsid*. The *clsid* is a globally unique identifier (GUID) specifying the application. The application's *progid* is a text string that serves the same purpose. The *description* key briefly describes the application *(Drawing Pad (Step14Container))*. Of these three keys, only the *description* value is visible to users. The document structure registers its own *progid* and *description*. Although each document type also needs its own unique *clsid*, if you omit it ObjectComponents supplies it for you by incrementing the application's *clsid*.

REGFORMAT entries list the data formats that the container can place on the Clipboard. The first parameter sets a priority order for the formats you use. 0 marks the format that renders data with the best fidelity, and higher numbers indicate lower fidelity. The second parameter represents a data format. The other parameters tell what presentation aspect of the format you use, what medium you use to transfer the data, and whether you can supply and receive Clipboard data in that format. All the data formats you specify with REGFORMAT are registered with the Windows Clipboard for you.

Even a simple container is usually capable of placing OLE objects on the Clipboard. If the user selects a linked or embedded object from the container's document and wants to transfer it through the Clipboard to another container, then the first container needs to act as a server by supporting at least the *ocrEmbedSource* or *ocrLinkSource* formats. Any application that registers either of these formats must also register *ocrMetafilePict*. The usual case is to register the five formats shown in the example. ObjectComponents automatically handles OLE objects in any of the standard formats for you. All you have to do is register the ones you want to support.

To register user-defined formats, replace the data format parameter with a string naming your format.

```
REGFORMAT(3, "MyOwnFormat", ocrContent, ocrIStorage, ocrGet)
```

If you register any custom Clipboard formats, you must also provide OLE with strings to describe your format in dialog boxes. Call *AddUserFormatName*, a method on classes derived from *TOleFrame*, to supply the descriptions.

REGDOCFLAGS adds to the registration structure an entry containing flags for a document template. The flags set options for running the File Open common dialog box.

After creating registration tables, you must pass them to the appropriate object constructors. The *AppReg* structure is passed to the *TOcRegistrar* constructor, as described in "Creating a registrar object." In a Doc/View application, document registration tables are passed to the document template constructor.

```
DEFINE_DOC_TEMPLATE_CLASS(TMyOleDocument, TMyOleView, MyTemplate);
MyTemplate myTpl(DocReg);
```

A program that uses several document templates should create a different registration table for each template. Each registration table must start with the BEGIN_REGISTRATION macro and have a different name, for example *DocReg1* and *DocReg2*.

All the information that normally gets passed to a document template constructor can be placed in a registration structure using REGFORMAT, REGDOCFLAGS, and REGDATA. Previous versions of OWL passed the same information to the document template as a series of separate parameters. The old method is still supported for backward compatibility, but new programs, whether they use OLE or not, should use the registration macros to supply document template parameters.

See Also

ocrxxxx Clipboard Constants

Registration Macros (OWL.HLP)

TOcRegistrar

Understanding Registration Macros

Creating a Registrar Object

See Also Registering a Container

Every ObjectComponents application needs a registrar object to manage its registration tasks. In a linking and embedding application, the registrar is an object of type *TOcRegistrar*. At the top of your source code file, declare a global variable holding a pointer to the registrar.

```
static TPointer<TOcRegistrar> Registrar;
```

The *TPointer* template ensures that the *TOcRegistrar* instance is deleted when the program ends.

Note: Name the variable *Registrar* to take advantage of the factory callback template used in the registrar's constructor.

The next step is to modify your *OwlMain* function to allocate a new *TOcRegistrar* object and initialize the global pointer *Registrar*. The *TOcRegistrar* constructor expects three parameters: the application's registration structure, the component's factory callback and the command line string that invoked that application.

- The registration structure you create with the registration macros.
- The factory callback you create with a template class.
  For a linking and embedding ObjectWindows application that uses Doc/View, the template class is called *TOleDocViewFactory*. The code in the factory template assumes you have defined an application dictionary called *AppDictionary* and a *TOcRegistrar** called *Registrar*.
- The command line string can come from the *GetCmdLine* method of *TApplication*.

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
  try {
    // Create Registrar object
    Registrar = new TOcRegistrar(::AppReg, TOleDocViewFactory<TMyApp>(),
                                 TApplication::GetCmdLine());
    return Registrar->Run();
  }
  catch (xmsg& x) {
    ::MessageBox(0, x.why().c_str(), "Exception", MB_OK);
  }
  return -1;
}
```

After initializing the *Registrar* pointer, your OLE container application must invoke the *Run* method of the registrar instead of *TApplication::Run*. For OLE containers, the registrar's *Run* simply invokes the application object's *Run* to create the application's windows and process messages. However, using the registrar method makes your application OLE server-ready. The following code shows a sample *OwlMain* before and after the addition of a registrar object. Boldface type highlights the changes.

Before:

```
// Non-OLE OwlMain
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  return TMyApp().Run();
}
```

After adding the registrar object:

```
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  ::Registrar = new TOcRegistrar(::AppReg,
         TOleDocViewFactory<TMyApp>(),
         TApplication::GetCmdLine());
  return ::Registrar->Run();
}
```

The last parameter of the *TOcRegistrar* constructor is the command line string that invokes the application. The registrar object processes the command line by searching for switches, such as Embedding or Automation, that OLE may have placed there. ObjectComponents takes whatever action the switches call for and then removes them. If for some reason you need to test the OLE switches, be sure to do it before constructing the registrar. If you have no use for the OLE switches, wait until after constructing the registrar before parsing the command line.

See Also

Factory Template Classes (OWL.HLP)

Processing the Command Line

TOcRegistrar

Turning a Doc/View Application Into an OLE Server

Supporting OLE Commands

See Also Turning a Doc/View Application Into an OLE Container

A container needs to place some standard OLE commands on its Edit menu. ObjectWindows implements the commands for you. A container also needs to let ObjectComponents read and write any linked or embedded objects when loading or saving documents.

The following topics discuss what you need to know in order to support OLE commands.

- Setting Up the Edit Menu and the Tool Bar
- Loading and Saving Compound Documents

See Also
Turning a Doc/View Application Into an OLE Server

Setting Up the Edit Menu and the Tool Bar

See Also Supporting OLE Commands

An OLE container places OLE commands on its Edit menu. The following table describes the standard OLE commands. It's not necessary to use all of them, but every container should support at least Insert Object, to let the user add new objects to the current document, and Edit Object, to let the user activate the currently selected object. The *TOleView* class has default implementations for all the commands. It invokes standard dialog boxes where necessary and processes the user's response. All you have to do is add the commands to the Edit menu for each view you derive from *TOleView*.

Menu command	**Predefined identifier**	**Command description**
Paste Special	CM_EDITPASTESPECIAL	Lets the user choose from available formats for pasting an object from the Clipboard.
Paste Link	CM_EDITPASTELINK	Creates a link in the current document to the object on the Clipboard.
Insert Object	CM_EDITINSERTOBJECT	Lets the user create a new object by choosing from a list of available types.
Edit Links	CM_EDITLINKS	Lets the user manually update the list of linked items in the current document.
Convert	CM_EDITCONVERT	Lets the user convert objects from one type to another.
Object	CM_EDITOBJECT	Reserves a space on the menu for the server's verbs (actions the server can take with the container's object).

If your OLE container has a tool bar, assign it the predefined identifier IDW_TOOLBAR. ObjectComponents must be able to find the container's tool bar if a server asks to display its own tool bar in the container's window. If ObjectComponents can identify the old tool bar, it temporarily replaces it with a new one taken from the server. For ObjectComponents to identify the container's tool bar, the container must use the IDW_TOOLBAR as its window ID, as shown here.

```
TControlBar *cb = new TControlBar(parent);
cb->Attr.Id = IDW_TOOLBAR;          // use this identifier
```

The *TOleFrame::EvAppBorderSpaceSet* method uses the IDW_TOOLBAR for its default implementation. A container can provide its own implementation to handle more complex situations, such as merging with multiple tool bars.

See Also
TOleFrame (OWL.HLP)
TOleView (OWL.HLP)

Loading and Saving Compound Documents

See Also Supporting OLE Commands

When the user pastes or drops an OLE object into a container, the object becomes data in the container's document. The container must store and load the object along with the rest of the document whenever the user chooses Save or Open from the File menu. The new *Commit* and *Open* methods of *TOleDocument* perform this chore for you. All you have to do is add calls to the base class in your own implementation of *Open* and *Commit*. The code that reads and writes your document's native data remains unchanged.

Because *TOleDocument* is derived from *TStorageDocument* rather than *TFileDocument*, it always creates compound files. *Compound files* are a feature of OLE 2 used to organize the contents of a disk file into separate compartments . You can ask to read or write from any compartment in the file without worrying about where on the disk the compartment begins or ends. OLE calls the compartments *storages*. The storages in a file can be ordered hierarchically, just like directories and subdirectories. Any storage compartment can contain other sub-storages.

Compound files are good for storing compound documents. When you call *Open* or *Commit*, ObjectComponents automatically creates storages in your file to hold whatever objects the document contains. All the document's native data is saved in the file's root storage. Your existing file data structure remains intact, isolated in a separate compartment. The following code shows how load compound documents.

```
// document class declaration derived from TOleDocument
class _DOCVIEWCLASS TMyDocument : public TOleDocument {
  // declarations
}


// document class implementation
bool
TDrawDocument::Open(int mode, const char far* path) {
  TOleDocument::Open(mode, path);    // load any embedded objects
                                  // code to load other document data
}
```

The *TOleDocument::Open* command does not actually copy the data for all the objects into memory. ObjectComponents is smart enough to load the data for particular objects only when the user activates them.

The next code shows how to save compound documents.

```
bool
TMyDocument::Commit(bool force) {
  TOleDocument::Commit(force);      // save the embedded objects
                                 // code to save other document data
  TOleDocument::CommitTransactedStorage();   // commit if in transacted mode
}
```

By default, *TOleDocument* opens compound files in transacted mode. Transacted mode saves changes in a temporary buffer and merges them with the file only after an explicit command. A revert command discards any uncommitted changes. *Commit* buffers a new transaction. *CommitTransactedStorage* merges all pending transactions.

The opposite of transacted mode is direct mode. Direct mode eliminates buffers and makes each change take effect immediately. To alter the default mode, override *TOleDocument::PreOpen*. Omit the *ofTransacted* flag to specify direct mode.

Note: **I**n order for compound file I/O to work correctly, you need to include the *dtAutoOpen* flag when you register docflags in the document registration table.

See Also

Turning a Doc/View Application Into an OLE Container

TOleDocument (OWL.HLP)

Building the Container

Turning a Doc/View Application Into an OLE Container

To build the container, include the right headers, compile with a supported memory model, and link to the ObjectComponents and OLE libraries.

The following topics discuss what you need to know when building containers:

- Including OLE Headers
- Compiling and Linking

Including OLE Headers

See Also   Building the Container

An ObjectComponents program needs the classes, structures, macros, and symbols defined in the header files for the ObjectWindows OLE classes. The following list shows the headers needed for an OLE container that uses the Doc/View model and an MDI frame window.

```
#include <owl/oledoc.h>       // replaces DOCVIEW.H
#include <owl/oleview.h>      // replaces DOCVIEW.H
#include <owl/olemdifr.h>     // replaces MDI.H
```

An SDI application includes oleframe.h instead of olemdifr.h.

See Also
Turning a Doc/View Application Into an OLE Container

Compiling and Linking

See Also Building the Container

Containers that use ObjectComponents and ObjectWindows require the large memory model. Link them with the OLE and ObjectComponents libraries.

The integrated development environment (IDE) chooses the right build options when you ask for OLE support. To build any ObjectComponents program from the command line, create a short makefile that includes the OWLOCFMK.GEN file found in the EXAMPLES subdirectory. Here, for example, is the makefile that builds the AutoCalc sample program:

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj autocalc.obj
HLP = MYPROGRAM
!include $(BCEXAMPLEDIR)\owlocfmk.gen
```

EXERES and OBJEXE hold the name of the file to build and the names of the object files to build it from. HLP is an optional online Help file. Finally, your makefile should include the OWLOCFMK.GEN file.

Name your file MAKEFILE and type this at the command line prompt:

```
make MODEL=l
```

Make, using instructions in OWLOCFMK.GEN, builds a new makefile tailored to your project. The new makefile is called WIN16L*xx*.MAK. The final two digits of the name tell whether the makefile builds diagnostic or debugging versions of the libraries. *01* indicates a debugging version, *10* a diagnostic version, and *11* means both kinds of information are included. The same command then runs the new makefile and builds the program. If you change the command to define MODEL as *d*, the new makefile is WIN16D*xx*.MAK and it builds the program as a DLL.

For more information about how to use OWLOCFMK.GEN, read the instructions at the beginning of MAKEFILE.GEN, found in the EXAMPLES directory.

The following table shows the libraries an ObjectComponents program links with.

Large model libraries	DLL import libraries	Description
OCFWL.LIB	OCFWI.LIB	ObjectComponents
OWLWL.LIB	OWLWI.LIB	ObjectWindows
BIDSL.LIB	BIDSI.LIB	Class libraries
OLE2W16.LIB	OLE2W16.LIB	OLE system DLLs
IMPORT.LIB	IMPORT.LIB	Windows system DLLs
MATHWL.LIB		Math support
CWL.LIB	CRTLDLL.LIB	C run-time libraries

The ObjectComponents library must be linked first, before the ObjectWindows library. Also, ObjectComponents requires RTTI and exception handling. Do not use compiler command line options that disable these features.

See Also
Turning a Doc/View Application Into an OLE Container

Turning an ObjectWindows Application Into an OLE Container

See Also

Turning an ObjectWindows application into an OLE container requires a few modifications.

The following topics discuss converting ObjectWindows applications into OLE containers:

1. Set Up the Application
2. Register a Container
3. Set Up the Client Window
4. Program the User Interface
5. Build a Container

By following these steps, you give your ObjectWindows application the following features:

- Linking
- Embedding
- OLE clipboard operations
- Drag and drop operations
- In-place editing
- Tool bar and menu merging
- Compound document storage
- OLE 2 user interface

Code excerpts are from the OWLOCF0.CPP sample in the EXAMPLES/OWL/TUTORIAL/OLE directory. The OWLOCF0.CPP sample is based on the STEP10.CPP sample used in the *ObjectWindows Tutorial*. It does not support OLE. OWLOCF1.CPP modifies the first program to create an OLE container.

See Also

Turning a Doc/View Application Into an OLE Container

Turning a C++ Application Into an OLE Container

Setting Up the Application

Turning an ObjectWindows Application Into an OLE Container

An ObjectComponents application needs an application dictionary, and the object you derive from *TApplication* must also derive from *TOcModule*.

The following topics discuss what you need to know to set up the application for ObjectComponents:

- Defining an Application Dictionary Object
- Modifying Your Application Class

Defining an Application Dictionary Object

Setting Up the Application

When a DLL is used by more than one application or process, it must maintain multiple copies of the global, static, and dynamic variables that represent its current state in each process. For example, the DLL version of ObjectWindows maintains a dictionary that allows it to retrieve the *TApplication* object which corresponds to the current active process. If you turn your application into a DLL server, the application must also maintain a dictionary of the *TApplication* objects created as each new client attaches to the DLL. The DEFINE_APP_DICTIONARY macro provides a simple and unified method for creating an application dictionary object. Insert the following statement with your other static variable declarations.

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

The DEFINE_APP_DICTIONARY macro correctly defines the *AppDictionary* variable regardless of how the application is built. in applications using the static version of ObjectWindows, it simply creates a reference to the existing ObjectWindows application dictionary. for DLL-servers using the DLL version of ObjectWindows, however, the macro declares a instance of the *TAppDictionary* class. It is important to use the name *AppDictionary* when creating your application dictionary object. This allows you to take advantage of the factory template classes for implementing a factory callback function.

Modifying Your Application Class

See Also Setting Up the Application

ObjectWindows provides the mix-in class *TOcModule* for applications that support linking and embedding. Change your application object so it derives from both *TApplication* and *TOcModule* as shown in the following example:

```
// Non-OLE application
class TScribbleApp : public TApplication { /* declarations */ };

// New declaration of same class
class TScribbleApp : public TApplication, public TOcModule { /* declarations
*/ };
```

The *TOcModule* object coordinates basic housekeeping chores related to registration and memory management. It also connects your application object to OLE.

Your *TApplication*-derived class must provide a *CreateOleObject* method with the following signature:

```
TUnknown*  CreateOleObject(uint32 options, TDocTemplate* tpl);
```

The method is used by the factory template class. Because containers don't create OLE objects, a container can implement *CreateOleObject* by simply returning 0. Servers have more work to do to implement *CreateOleObject*.

```
//
// non-OLE application class
//
class TScribbleApp : public TApplication {
  public:
    TScribbleApp() : TApplication("Scribble Pad") {}

  protected:
    InitMainWindow();

};

//
// New declaration of same class
//
class TScribbleApp : public TApplication, public TOcModule {
  public:
    TScribbleApp() : TApplication(::AppReg["description"]){}
    TUnknown*  CreateOleObject(uint32, TDocTemplate*){ return 0; }

  protected:
    InitMainWindow();
```

See Also
TOcModule

TApplication (OWL.HLP)

Registering a Container

Turning an ObjectWindows Application Into an OLE Container

To register an application, you build registration tables with macros. Then you pass the tables to a registrar object to process the information they contain.

The following topics discuss these tasks:

- Creating Registration Tables
- Creating a Registrar Object

Creating Registration Tables

Registering a Container

OLE requires programs to identify themselves by registering unique identifiers and names. ObjectWindows offers macros that let you build a structure to hold registration information. The structure can then be used when creating the application's instance of *TOcRegistrar*.

Here are the commands to create a simple container registration structure:

```
REGISTRATION_FORMAT_BUFFER(100)    // create buffer for expanding macros

BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid, "{9B0BBE60-B6BD-101B-B3FF-86C8A0834EDE}")
  REGDATA(description, "Scribble Pad Container")
END_REGISTRATION
```

The first macro, REGISTRATION_FORMAT_BUFFER, sets the size of a buffer needed temporarily as the macros that are expanded. The REGDATA macro places items in the registration structure, *AppReg*. Each item in *AppReg* is a smaller structure that contains a key, such as *clsid* or *progid*, and a value assigned to the key. The values you assign are case-sensitive strings. The order of keys within the registration table does not matter.

Insert the registration macros after your declaration of the application dictionary. Since the value of the *clsid* key must be a unique number identifying your application, it is recommended that you generated a new value using the GUIDGEN.EXE utility. (The *ObjectWindows Reference Guide* entry for *clsid* explains other ways to generate an identifer.) Of course, modify the value of the *description* key to describe your container.

The *AppReg* structure built in the sample code is an *application registration structure*. A container may also build one or more *document registration structures*. Both structures are built alike, but each contains a different set of keys and values. The keys in an application registration structure describe attributes of the application. A document registration structure describes the type of document an application can create. A document's attributes include the data formats that it can exchange with the clipboard, its file extensions, and its document type name. The OWLOCF1 sample application does not create any document registration structures.

Creating a Registrar Object

<u>See Also</u> <u>Registering a Container</u>

Every ObjectComponents application needs to create a registrar object to manage all of its registration tasks. Insert the following line after the **#include** statements in your main .CPP file.

```
static TPointer<TOcRegistrar> Registrar;
```

The *TOcRegistrar* instance is created in your OwlMain function. Declaring the pointer of type *TPointer<TOcRegistrar>* instead of *TOcRegistrar** ensures that the *TOcRegistrar* instance is deleted.

Note: Name the variable *Registrar* to take advantage of the *TOleFactory* template for implementing a factory callback.

The next step is to modify your *OwlMain* function to allocate a new *TOcRegistrar* object to initialize the global pointer *Registrar*. The *TOcRegistrar* constructor requires three parameters: the application's registration structure, the component's factory callback and the command line string that invoked that application.

- The registration structure you create with the registration macros.
- The factory callback you create with an ObjectWindows factory template.

You can write your own callback function from scratch if you prefer, but the templates are much easier to use. For a linking and embedding ObjectWindows application that doesn't use Doc/View, the template class is called *TOleFactory*. The code in the factory template assumes you have defined an application dictionary called *AppDictionary* and a *TOcRegistrar** called *Registrar*.

- The command line string comes from the *GetCmdLine* method of *TApplication.*

Here is the code to create the registrar.

```
int OwlMain(int, char*[])
{

  // create the registrar object
  ::Registrar = new TOcRegistrar(::AppReg, TOleFactory<TScribbleApp>(),
                                 TApplication::GetCmdLine());
}
```

Factories are explained in more detail in the *ObjectWindows Reference Guide*.

After initializing the *Registrar* pointer, your OLE container application must invoke *TOcRegistrar::Run* instead of *TApplication::Run*. For OLE containers, the registrar's *Run* simply invokes the application object's *Run* to create the application's windows and process messages. In a server, however, *TOcRegistrar::Run* does more. Using the registrar's *Run* method in a container makes it easier to modify the application later if you decide to turn it into a server.

Before and After

Here is the *OwlMain* from OWLOCF1, omitting for clarity the usual **try** and **catch** statements. The lines in bold are the new code.

Before:

```
// Non-OLE OwlMain
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  return TScribbleApp().Run();
}
```

After adding the registrar object:

```
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  ::Registrar = new TOcRegistrar(::AppReg, TOleFactory<TScribbleApp>(),
                                   TApplication::GetCmdLine());
  return ::Registrar->Run();
}
```

See Also
Factory Template Classes (OWL.HLP)

Setting Up The Client Window

Turning an ObjectWindows Application Into an OLE Container

An ObjectWindows SDI application can use a frame window that does not contain a client window. Similarly, an ObjectWindows MDI application can use MDI child windows that do not contain a client window. Omitting the client window makes it harder to convert the application from one kind of frame to anotherSDI, MDI, or decorated frame. It is also awkward when building OLE 2 applications.

For example, it is easier for a container's main window to make room for a server's tool bar if the container owns a client window. To take full advantage of the ObjectWindows OLE classes, your application must use a client window. For more information about using client windows, see the *ObjectWindows Tutorial*.

The following topics discuss setting up the Client Window.

- Inheriting from OLE Classes
- Delaying the Creation of the Client Window in SDI Applications
- Creating ObjectComponents View and Document Objects

Inheriting from OLE Classes

See Also Setting Up the Client Window

ObjectWindows provide several classes that include default implementations for many OLE operations. To adapt an existing ObjectWindows program to OLE, change its derived classes to inherit from the OLE classes.

The *TOleFrame* and *TOleMDIFrame* classes both derive from decorated window classes. The OLE 2 user interface requires that containers be prepared to handle tool bars and status bars. Even if a container has no such decorations, servers might need to display their own in the container's window. The OLE window classes handle those negotiations for you. The following code shows how to change the declaration for a client window. Boldface type highlights the changes.

Before:

```
// Pre-OLE declaration of a client window
class TScribbleWindow : public TWindow {
    // declarations
};
DEFINE_RESPONSE_TABLE1(TScribbleWindow, TWindow);
```

After changing the declaration to derive from an OLE-enabled class:

```
// New declaration of the same window class
class TScribbleWindow : public TOleWindow {
    // declarations
};
DEFINE_RESPONSE_TABLE1(TScribbleWindow, TOleWindow);
```

See Also
TOleFrame (OWL.HLP)
TOleMDIFrame (OWL.HLP)

Delaying the Creation of the Client Window in SDI Applications

See Also Setting Up the Client Window

ObjectWindows applications create their main window in the *InitMainWindow* method of the *TApplication*-derived class. Typically, SDI applications also create their initial client window in the *InitMainWindow* function. The following code shows the typical sequence.

```
void
TDrawApp::InitMainWindow()
{
  // Construct the decorated frame window
  TDecoratedFrame* frame = new TDecoratedFrame(0, "Drawing Pad",
                                               new TDrawWindow(0), true);
    // more declarations to init and set the main window
}
```

When used in the OLE frame and client classes, however, that sequence presents a timing problem for OLE. The OLE client window must be created after the OLE frame has initialized its variables pointing to ObjectComponents classes. To meet this requirement, an SDI OLE application should create only the frame window in the *InitMainWindow* function. Create the client window in the *InitInstance* method of your application class. Boldface type highlights the changes.

```
void
TDrawApp::InitMainWindow()
{
  // construct the decorated frame window
  TOleFrame* frame = new TOleFrame("Drawing Pad", 0, true);

    // more declarations to init and set the main window
}


void
TDrawApp::InitInstance()
{
  TApplication::InitInstance();

  // create and set client window
  GetMainWindow()->SetClientWindow(new TDrawWindow(0));
}
```

See Also
TApplication (OWL.HLP)

Creating ObjectComponents View and Document Objects

See Also Setting Up the Client Window

For every client window capable of having linked or embedded objects, you must create a *TOcDocument* object to manage the embedded OLE objects, and a *TOcView* object to manage the presentation of the OLE objects. The *CreateOcView* method from the *TOleWindow* class creates both the container document and the container view. Add a call to *CreateOcView* in the constructor of your *TOleWindow*-derived class.

```
// Pre-OLE declaration of a client window constructor
TScribbleWindow::TScribbleWindow(TWindow* parent, char far* filename)
: TWindow(parent, 0, 0)
{

}

// New declaration of client window constructor
TScribbleWindow::TScribbleWindow(TWindow* parent, char far* filename)
: TOleWindow(parent, 0)
{


  // Create TOcDocument object to hold OLE parts
  // and TOcView object to provide OLE services.
  CreateOcView(0, false, 0);
}
```

Notice that unlike the *TWindow* constructor, the *TOleWindow* constructor does not require a title parameter. It is unnecessary because *TOleWindow* is always the client of a frame. *TWindow*, on the other hand, can be used as a non-client windowa pop-up, for example.

See Also
TOleWindow (OWL.HLP)
TOcView

Programming the User Interface

Turning an ObjectWindows Application Into an OLE Container

The next set of adaptations provide standard OLE user interface features such as menu merging and drag and drop. The following topics discuss programming the user interface.

The following topics discuss programming the user interface.

- Handling OLE-Related Messages and Events
- Supporting Menu Merging
- Updating the Edit Menu
- Assigning a Tool Bar ID

Handling OLE-Related Messages and Events

Programming the User Interface

ObjectComponents notifies your application's windows of OLE-related events by sending the WM_OCEVENT message. The ObjectWindows OLE classes provide default handlers for the various WM_OCEVENT event notifications. Furthermore, the ObjectWindows classes also process a few standard Windows messages to add additional features of the standard OLE user interface. For example, if a user double-clicks within the client area of your container window, a handler in *TOleWindow* checks whether the click occurred over an embedded object and, if so, activates the object. Similarly, the *TOleWindow::EvPaint* method causes each embedded object to draw itself. The following table lists the methods implemented by the client window (TOleWindow) and frame window(*TOleFrame*, *TOleMDIFrame*) classes. If you override these handlers in your derived class you must invoke the base class version.

Method	Message	Class	Description
EvSize	WM_SIZE	Frame	Notifies embedded servers of the size change.
EvTimer	WM_TIMER	Frame	Invokes IdleAction so that DLL servers can carry out command enabling.
EvActivateApp	WM_ACTIVATEAPP	Frame	Notifies embedded servers about being activated.
EvLButtonDown	WM_LBUTTONDOWN	Client	Deactivates any in-place active object.
EvRButtonDown	WM_RBUTTONDOWN	Client	Displays pop-up verb menu if cursor is on an embedded object.
EvLButtonDblClk	WM_LBUTTONDBLCLK	Client	Activates any embedded object under the cursor.
EvMouseMove	WM_MOUSEMOVE	Client	Allows user to move or resize an embedded object.
EvLButtonUp	WM_LBUTTONUP	Client	Informs the selected object of position or size changes.
EvSize	WM_SIZE	Client	Informs TOcView object that window has changed size.
EvMdiActivate	WM_MDIACTIVATE	Client	Informs TOcView object that window has changed size.
EvMouseActivate	WM_MOUSEACTIVATE	Client	Forwards the message to the top-level parent window and returns the code to activate the client window.
EvSetFocus	WM_SETFOCUS	Client	Notifies any in-place server of focus change.
EvSetCursor	WM_SETCURSOR	Client	Changes cursor shape if within an embedded object.
EvDropFiles	WM_DROPFILES	Client	Embeds dropped file(s).
EvPaint	WM_PAINT	Client	Causes embedded objects to paint.
EvCommand	WM_COMMAND	Client	Processes command IDs of verbs.
EvCommandEnable	WM_COMMANDENABLE	Client	Processes command IDs of verbs.

In some cases, you might need to know what action the base class handler took before you decide what to do in your overriding handler. This is particularly true for mouse-related messages. If the base class handled a double-click action, for example, the user intended the action to activate an object and you probably don't want your code to reinterpret the double-click as a different command. The code that follows shows how to coordinate with a base class handler. These three procedures let the user draw on the surface of the client window with the mouse.

```
void
TMyClient::EvLButtonDown(uint modKeys, TPoint& pt)
{
  if (!Drawing) {
    SetCapture()
    Drawing = true;
    // additional GDI calls to display drawing
  }
}

void
TMyClient::EvMouseMove(uint modKeys, TPoint& pt)
{
  if (Drawing) {
    // additional GDI calls to display drawing
  }
}

void
TMyClient::EvLButtonUp(uint modKeys, TPoint& pt)
{
  if (Drawing) {
    Drawing = false;
    ReleaseCapture();
  }
}
```

As an OLE container, however, the client window may contain embedded objects. Mouse events performed on these objects should not result in any drawing operation. This code shows the handlers updated to allow and check for OLE related processing. Boldface type highlights the changes.

```
void
TMyClient::EvLButtonDown(uint modKeys, TPoint& pt)
{
  TOleWindow::EvLButtonDown(modKeys, pt);

  if (!Drawing && !SelectEmbedded()) {
    SetCapture()
    Drawing = true;
    // additional GDI calls to display drawing
  }
}

void
TMyClient::EvMouseMove(uint modKeys, TPoint& pt)
{
  TOleWindow::EvMouseMove(modKeys, pt);

  if (Drawing && !SelectEmbedded()) {
    // additional GDI calls to display drawing
  }
}
```

```
void
TMyClient::EvLButtonUp(uint modKeys, TPoint& pt)
{
  if (Drawing && !SelectEmbedded()) {
    Drawing = false;
    ReleaseCapture();
  }

  TOleWindow::EvLButtonUp(modKeys, pt);
}
```

The *SelectEmbedded* method is inherited from *TOleWindow*. It returns **true** if an embedded object is currently being moved. The client window calls it to determine whether a mouse message has already been processed by the OLE base class.

Typically, your derived class must call the base class handlers before processing any event or message. The *EvLButtonUp* handler, however, calls the base class last. Doing so allows the handler to rely on *SelectEmbedded* which is likely to be reset after *TOleWindow* processes the mouse-up message.

Supporting Menu Merging

See Also Programming the User Interface

The menu bar of an OLE container with an active object is composed of individual pieces from the normal menus of both the container and server. The container contributes pop-up menus dealing with the application frame or with documents. The server, on the other hand, provides the Edit menu, the Help menu, and any menus that let the user manipulate the activated object.

OLE divides the top-level menus of a menu bar into six groups. Each group is a set of contiguous top-level drop-down menus. Each group is made up of zero or more pop-up menus. The menu groups are named File, Edit, Container, Object, Window, and Help. The group names are for convenience only. They suggest a common organization of related commands, but you can group the commands any way you like.

When operating on its own, a container or server provides the menus for all of the six groups. During an in-place edit session, however, the container retains control of the File, Container and Window groups while the server is responsible for the Edit, Object, and Help groups.

The *TMenuDescr* class automatically handles all menu negotiations between the server and the container. You simply identify the various menu groups within your menu resource, and ObjectWindows displays the right ones at the right times.

To indicate where groups begin and end in your menu resource, insert SEPARATOR menu items between them. Remember to mark all six groups even if some of them are empty. The *TMenuDescr* class scans for the separators when loading a menu from a resource. It removes the separators found between top-level entries and builds a structure which stores the number of pop-up menus assigned to each menu group. This information allows ObjectWindows to merge the server's menu into your container's menu bar.

The following menu resource script, taken from the *ObjectWindows Tutorial*, illustrates defining a simple application menu before it is divided into groups.

```
COMMANDS MENU
{
  pop-up "&File"
  {
    MENUITEM "&New",     CM_FILENEW
    MENUITEM "&Open",    CM_FILEOPEN
    MENUITEM "&Save",    CM_FILESAVE
    MENUITEM "Save &As", CM_FILESAVEAS
  }

  pop-up "&Tools"
  {
    MENUITEM "Pen &Size",  CM_PENSIZE
    MENUITEM "Pen &Color", CM_PENCOLOR
  }

  pop-up "&Help"
  {
    MENUITEM "&About",     CM_ABOUT
  }
}
```

The File menu entry belongs to the OLE File menu group. The Tools menu allows the user to edit the application's document, so it belongs to the Edit group. This application does not contain any menus belonging to the Object, Container, or Window group. And finally, the Help menu belongs to the Help group.

The following code is a modified version of the same menu resource with SEPARATOR dividers inserted to indicate where one group stops and the next begins. Boldface type highlights the changes.

```
COMMANDS MENU
{
  pop-up "&File"
  {
    MENUITEM "&New",     CM_FILENEW
    MENUITEM "&Open",    CM_FILEOPEN
    MENUITEM "&Save",    CM_FILESAVE
    MENUITEM "Save &As", CM_FILESAVEAS
  }

  MENUITEM SEPARATOR         // end of File group, beginning of Edit group

  pop-up "&Tools"
  {
    MENUITEM "Pen &Size",  CM_PENSIZE
    MENUITEM "Pen &Color", CM_PENCOLOR
  }

  MENUITEM SEPARATOR         // end of Edit group, beginning of Container
group
  MENUITEM SEPARATOR         // end of Container group, beginning of Object
group
  MENUITEM SEPARATOR         // end of Object group, beginning of Window
group
  MENUITEM SEPARATOR         // end of Window group, beginning of Help group

  pop-up "&Help"
  {
    MENUITEM "&About",     CM_ABOUT
  }
}
```

Insert separators in your application's menu to indicate the various menu groups. Then modify your code to use the *SetMenuDescr* method when assigning your frame window's menu. This example shows the menu assignment before and after adding menu merging. Boldface type highlights the changes.

Before:

```
// original menu assignment
void
TScribbleApp::InitMainWindow()
{
  TDecoratedFrame* frame;
    // Initialize frame and decorations etc. etc.

  // Assign frame's menu
  frame->AssignMenu("COMMANDS");
}
```

After including group indicators in the menu:

```
void
TScribbleApp::InitMainWindow()
{
  TOleFrame* frame;
    // Initialize frame and decorations etc. etc.
  // Assign frame's menu
  frame->SetMenuDescr(TMenuDescr("COMMANDS"));
}
```

Instead of using separators to show which drop-down menus belong to each group, you can use the *TMenuDescr* constructor whose parameters accept a count for each group. For more details, see the description of the *TMenuDescr* constructors in the *ObjectWindows Reference Guide*.

See Also
TMenuDescr (OWL.HLP)

Updating the Edit menu

Programming the User Interface

An OLE container places OLE commands on its Edit menu. The *TOleWindow* class has default implementations for all of them. It invokes standard dialogs boxes where necessary and processes the user's response. All you have to do is add the commands to the Edit menu of your frame window. It's not necessary to support all six commands, but every container should support at least CM_EDITINSERTOBJECT, to let the user add new objects to the current document, and CM_EDITOBJECT, to let the user choose verbs for the currently selected object.

ObjectWindows defines standard identifiers for the OLE Edit menu commands in owl/ oleview.rh. Update your resource file to include the header file and use the standard identifiers to put OLE commands on the Edit menu.

```
#include <owl/oleview.rh>
#include <owl/edit.rh>

COMMANDS MENU
{
    // File menu goes here

  MENUITEM SEPARATOR
  pop-up "&Edit"
  {
    MENUITEM "&Undo\aCtrl+Z",             CM_EDITUNDO
    MENUITEM Separator
    MENUITEM "&Cut\aCtrl+X",              CM_EDITCUT
    MENUITEM "C&opy\aCtrl+C",             CM_EDITCOPY
    MENUITEM "&Paste\aCtrl+V",            CM_EDITPASTE
    MENUITEM "Paste &Special...",        CM_EDITPASTESPECIAL
    MENUITEM "Paste &Link",              CM_EDITPASTELINK
    MENUITEM "&Delete\aDel",              CM_EDITDELETE
    MENUITEM SEPARATOR
    MENUITEM  "&Insert Object...",       CM_EDITINSERTOBJECT
    MENUITEM  "&Links...",               CM_EDITLINKS
    MENUITEM  "&Object",                 CM_EDITOBJECT
    MENUITEM SEPARATOR
    MENUITEM  "&Show Objects",           CM_EDITSHOWOBJECTS
  }
    // other menus go here
}
```

Assigning a Tool Bar ID

See Also Programming the User Interface

If your OLE container has a tool bar, assign it the predefined identifier IDW_TOOLBAR. ObjectWindows must be able to find the container's tool bar if a server needs to display its own tool bar in the container's window. If ObjectWindows can identify the old tool bar, it temporarily replaces it with the new one taken from the server. For ObjectWindows to identify the container's tool bar, the container must use the IDW_TOOLBAR as its window ID.

```
TControlBar* cb = new TControlBar(parent);
cb->Attr.Id = IDW_TOOLBAR;
```

The *TOleFrame::EvAppBorderSpaceSet* method uses the IDW_TOOLBAR for its default implementation. A container can provide its own implementation to handle more complex situations, such as merging with multiple tool bars.

See Also
TOleFrame (OWL.HLP)

Building a Container

Turning an ObjectWindows Application Into an OLE Container

A container must include OLE ObjectWindows headers, compile with a supported memory model, and link to the right libraries.

Including OLE Headers

ObjectWindows provides OLE-related classes, structures, macros and symbols in various header files. The following list shows the headers needed for an OLE container using an SDI frame window.

```
#include <owl/oleframe.h>
#include <owl/olewindo.h>
#include <ocf/ocstorag.h>
```

an MDI application includes olemdifr.h instead of oleframe.h.

Compiling and Linking

ObjectWindows containers and servers must be compiled with the large memory model. They must be linked with the OLE, ObjectComponents, and ObjectWindows libraries.

Turning a C++ Application Into an OLE Container

See Also

If you are writing a new program, consider using ObjectWindows to save yourself some work. The ObjectWindows Library contains built-in code that automatically performs some tasks common to all ObjectComponents programs. Programs that don't use ObjectWindows must undertake these chores for themselves.

If you are writing a new program, consider using the AppExpert and ObjectWindows to save yourself some work. But ObjectComponents works well in straight C++ programs without ObjectWindows, as well.

The following topics discuss the changes needed for turning a C++ application into an ObjectComponents container:

1. Register a Container
2. Create a View Window
3. Program the Main Window
4. Build the Program

The topics above illustrate each step using examples from the programs in the EXAMPLES/OCF/CPPOCF directory. The source files titled CPPOCF0 contain a windows application that does not support OLE. CPPOCF1 modifies the first program to make it an OLE container. The code samples for this discussion come from CPPOCF1. The same directory also contains CPPOCF2, an OLE server.

CPPOCF1 is a simple application that supports basic container functions: registering the application, creating objects to initialize a new document, and embedding an object in the document. For ideas about implementing other features, you might want to look at the source code for ObjectWindows OLE classes such as *TOleWindow* and *TOleView*.

The explanations that follow do not describe all the differences in the source code from CPPOCF0 to CPPOCF1. The omit details that are not specific to ObjectComponents and OLE, such as calling *RegisterClass* for a new child window.

See Also
Turning a Doc/View Application Into an OLE Container
Turning an Objectwindows Application Into an OLE Container

Registering a Container

Turning a C++ Application Into an OLE Container

Giving the system the information it needs about your container takes three steps: building a registration table, passing the table to a registrar object, and creating a memory allocator object.

The following topics discuss the three steps to registering a container:

- Building a Registration Table
- Creating the Registrar Object
- Creating a Memory Allocator

Building a Registration Table

See Also Registering a Container

A container uses the registration macros to build a registration table describing the application. A container does not need to create document registration tables except to support being a link source. (When a container is a link source, it allows other containers to create links to objects in its own documents.)

Here is the registration table from CPPOCF1:

```
REGISTRATION_FORMAT_BUFFER(100)
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,        "{8646DB80-94E5-101B-B01F-00608CC04F66}")
  REGDATA(progid,       APPSTRING ".Application.1")
  REGDATA(description,  "Sample container")
END_REGISTRATION
```

The application's header file includes this line:

```
#define APPSTRING   "CppOcf1"
```

The *progid* string is therefore "CppOcf1.Application.1."

The registration macros build a structure of type *TRegList*. Each entry in the structure contains a key, such as *clsid* or *progid*, and a value assigned to the key. Internally ObjectComponents finds the values by searching for the keys. The order in which the keys appear does not matter.

See Also
Understanding Registration

Creating the Registrar Object

See Also   Registering a Container

The registrar object records application information in the system registration database, processes any OLE switches on the application's command line, and notifies OLE that the server is running. CPPOCF1 declares a static pointer for the registrar object:

```
TOcRegistrar* OcRegistrar = 0;
TOcApp*       OcApp       = 0;
```

The second variable, *OcApp*, points to the connector object that implements OLE interfaces for the application to communicate with OLE. The registrar creates the *TOcApp* object in *WinMain*.

Create the registrar as you initialize the application in *WinMain*. Instead of entering a message loop, call the registrar's *Run* method. When *Run* returns, the application is ready to shut down. Delete the registrar before you quit. This excerpt from the CPPOCF1 *WinMain* function shows all the steps.

```
int PASCAL
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  char far* lpCmdLine, int nCmdShow)
{
  try {
    TOleAllocator allocator(0); //required for OLE2
    MSG msg;

    // Initialize OCF objects
    OcRegistrar = new TOcRegistrar(::AppReg, 0,
                               string(lpCmdLine), 0);
    OcRegistrar->CreateOcApp(OcRegistrar->GetOptions(), OcApp);

        // per-instance and per-task initialization code goes here

    // Standard Windows message loop
    while (GetMessage(&msg, 0, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  catch (TXBase& xbase) {
    MessageBox(GetFocus(), xbase.why().c_str(), "Exception caught", MB_OK);
  }

  // free the registrar object
  delete OcRegistrar;
  return 0;
}
```

TOcRegistrar Constructor Parameters

The *TOcRegistrar* constructor takes four parameters:

- *::AppReg*, is the application registration structure already built with the registration macros.
- *ComponentFactory* is a callback function.
  The callback is responsible for creating any of the application's OLE components, including the application itself, as required. The callback contains the application's message loop, as well.
- *cmdLine* is a *string* object holding the application's command line.
  The registrar searches the command line for OLE-related switches such as Automation or Embedding, and it sets internal running mode flags accordingly.
- 0 is a null pointer to a document list.
  Because CPPOCF1 does not register any document types, this list is empty. A container registers

document types to support being a link source.

Besides recording information in the registration database, the registrar object also creates the *TOcApp* connector object when you call *CreateOcApp*.

See Also

TOcRegistrar

Building a Registration Table

Creating a Memory Allocator

Registering a Container

The beginning of the *WinMain* procedure creates a *TOleAllocator*:

```
TOleAllocator allocator(0);      // use default memory allocator
```

The allocator's constructor initializes the OLE libraries and its destructor releases them when the object goes out of scope. Passing 0 to the constructor tells it to let OLE use its standard memory functions whenever allocating memory on behalf of this application.

Creating a View Window

Turning a C++ Application Into an OLE Container

ObjectComponents imposes one design requirement: a compound document must have its own window, separate from the application's main window. To keep the distinction clear, we'll call the main window the *frame* window, because it uses the WS_THICKFRAME style and has a visible border on the screen. The second window has no visible border. We'll call it the *view* window because that is where the application displays its data. The view window always exactly fills the frame window's client area, so from the user's point of view the frame window appears to be the only window. ObjectComponents needs the view window, though, because it expects to send some event messages to the application and some to the view. (View windows are sometimes called client windows, too.)

In an SDI application like the CPPOCF1 sample program, the frame window controls the view window. When the frame window receives a WM_SIZE message, it moves the view to keep it aligned with the frame's client area. When it receives WM_CLOSE, it destroys both itself and the view window.

In an MDI application, each child window creates its own view. The child window does what the SDI frame does: creates and manages a view for the document it displays.

The following topics discuss tasks involved with creating a view window:

- Creating, Resizing, and Destroying the View Window
- Creating a TOcDocument and TOcView
- Handling WM_OCEVENT
- Handling Selected View Events
- Painting the Document
- Activating and Deactivating Objects

Creating, Resizing, and Destroying the View Window

Creating a View Window

Before creating the view window, the application must first register a class for the view window. CPPOCF1 registers both classes in *InitApplication*.

CPPOCF1 creates the view window in its factory because the factory is in charge of creating new documents on request. The code for the view window, as you'll see, connects the new document to OLE by creating some ObjectComponents helper objects. The factory calls this function to create the view window:

```
HWND CreateViewWindow(HWND hwndParent)
{
  HWND hwnd = CreateWindow(VIEWCLASSNAME, "",
    WS_CHILD | WS_CLIPCHILDREN | WS_CLIPSIBLINGS | WS_VISIBLE | WS_BORDER,
    10, 10, 300, 300,
    hwndParent, (HMENU)1, HInstance, 0);
  return hwnd;
}
```

CPPOCF1 resizes and destroys the view window when the frame window receives WM_SIZE and WM_CLOSE messages.

```
void
MainWnd_OnSize(HWND hwnd, UINT /*state*/, int /*cx*/, int /*cy*/)
{
  if (IsWindow(HwndView)) {
    TRect rect;
    GetClientRect(hwnd, &rect);
    MoveWindow(HwndView, rect.left, rect.top, rect.right, rect.bottom,
true);
  }
}

void
MainWnd_OnClose(HWND hwnd)
{
  if (IsWindow(HwndView))
    DestroyWindow(HwndView);
  DestroyWindow(hwnd);
}
```

The view window always fills the frame window's client area exactly. If the user opens and closes documents or embeds objects, the changes show up in the view window.

Creating a TOcDocument and TOcView

See Also		Creating a View Window

If the user embeds several objects in the container's view window, they all become part of a single compound document. If the container supports file I/O, then the user can save and load different documents.

For every document the container opens or creates, it needs one view window and two helper objects: *TOcDocument* and *TOcView*. The document helper manages the collection of objects inserted in the document. The view helper connects the document to OLE. More specifically, it implements interfaces that OLE can call to communicate with the document. When OLE tells the view object that something noteworthy has occurred, the view object sends a message to the view window.

The sample CPPOCF1 container declares two global pointers to hold the two helper objects. (A program that opens more than one document at a time needs more than one pair of variables.)

```
TOcDocument*  OcDoc       = 0;
TOcView*      OcView      = 0;
```

CPPOCF1 creates and destroys the two helpers when it creates and destroys the view window that displays the document.

```
bool
ViewWnd_OnCreate(HWND hwnd, CREATESTRUCT FAR* /*lpCreateStruct*/)
{
  OcDoc  = new TOcDocument(*OcApp);   // create document helper
  OcView = new TOcView(*OcDoc);       // create view connector
  if (OcView)
    OcView->SetupWindow(hwnd);        // attach view to window
  return true;
}

void
ViewWnd_OnDestroy(HWND /*hwnd*/)
{
  // code to de-activate objects goes here (explained later)

  if (OcView)
    OcView->ReleaseObject();     // do not delete the view; it is a COM
object
  OcDoc->Close();                // release the server for each embedded
object
  delete OcDoc;                  // delete the document helper; it is not a
COM object
}
```

The WM_CREATE message handler for the view window creates both helpers and then calls *OcView->SetupWindow*. The *SetupWindow* method tells the *TOcView* object where to send event messages. in this case, it sends messages to *hwnd*, the view window. The view window now receives WM_OCEVENT messages.

When the view window is destroyed, it makes three calls to dispose of the helper objects. *OcView->ReleaseObject* signals that the view window is through with the *TOcView* connector object. You shouldn't call **delete** for a *TOcView* object because the OLE system might still need more information before it allows the view to shut down. ReleaseObject tells the TOcView object that you don't need it any longer. The view subsequently destroys itself as soon as all other OLE clients finish with it, as well. The TOcView destructor is protected to prevent you from calling it directly.

A container document only has other OLE clients if it registers support for being a link source. in that case, other applications can create links to objects in the container's document.

The *TOcDocument* object, on the other hand, is not a connector object and so you can destroy it with **delete** in the usual way. First, however, you should call *Close* to release the server applications that OLE may have invoked to support each linked or embedded object.

See Also

TOcDocument

TOcView

Handling WM_OCEVENT

See Also Creating a View Window

Because the *TOcView::SetupWindow* method bound the *TOcView* connector to the view window, the connector sends its event notification messages to the window. All ObjectComponents events are sent in the WM_OCEVENT message, so the view window procedure must respond to WM_OCEVENT.

```
long CALLBACK _export
ViewWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
  switch (message) {
.
.
.
  // other message crackers go here
    HANDLE_MSG(hwnd, WM_OCEVENT,  ViewWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}
```

The HANDLE_MSG message cracker macro for WM_OCEVENT is defined in the ocf/ ocfevx.h header. The same header also defines a another cracker for use in the WM_OCEVENT message handler.

```
// Subdispatch OC_VIEWxxxx messages
long
ViewWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
{
  switch (wParam) {
    // insert an event cracker for each OC_VIEWxxxx message you want to
handle
    HANDLE_OCF(hwnd, OC_VIEWPARTINVALID, ViewWnd_OnOcViewPartInvalid);
  }
  return true;
}
```

The WM_OCEVENT message carries an event ID in its *wParam*, just as WM_COMMAND messages carry command IDs. OC_VIEWPARTINVALID is one possible event, indicating that it is time to repaint a linked or embedded object. The HANDLE_OCF macro calls the handler you designate for each ObjectComponents event, just as HANDLE_MSG calls the handler for for a window message.

CPPOCF1 chooses to handle only the OC_VIEWPARTINVALID message. To handle others, add one HANDLE_OCF macro for each event ID.

See Also
WM_OCEVENT message

Handling Selected View Events

Creating a View Window

Each HANDLE_OCF macro calls a different handler function. In the example, the handler function is called *ViewWnd_OnOcViewPartInvalid*. ObjectComponents sends this message to a container when one of the OLE data objects in its document needs to be repainted.

```
bool
ViewWnd_OnOcViewPartInvalid(HWND hwnd, TOcPart& part)
{
  HDC dc = GetDC(hwnd);
  SetMapMode(dc, MM_ANISOTROPIC);
  SetWindowOrg(dc, 0, 0);
  SetViewportOrg(dc, 0, 0);
  RECT rect = part.GetRect();
  LPtoDP(dc, (POINT*)&rect, 2);
  InvalidateRect(hwnd, &rect, true);
  ReleaseDC(hwnd, dc);
  return true;
}
```

The *TOcPart* parameter represents the object that needs painting. ObjectComponents creates a *TOcPart* object for every linked or embedded object in a container document. CPPOCF1 handles this message by asking the part for its coordinates and invalidating that part of its client area. The *InvalidateRect* command results in a WM_PAINT message, and the *ViewWnd_OnPaint* procedure responds by drawing the document.

Painting the Document

See Also Creating a View Window

Painting a compound document requires two steps: drawing the container's own data and drawing all the linked or embedded objects. Here's the basic frame for a paint procedure:

```
void
ViewWnd_OnPaint(HWND hwnd)
{
  PAINTSTRUCT ps;
  HDC dc = BeginPaint(hwnd, &ps);
  //
  // Do your regular painting here
  //

  // Now draw embedded objects
  ViewWnd_PaintParts(hwnd, dc, false);
  EndPaint(hwnd, &ps);
}
```

The code for *ViewWnd_PaintParts* is the same in most applications.

```
bool
ViewWnd_PaintParts(HWND hwnd, HDC dc, bool metafile)
{
  // get logical coordinates of area to draw
  TRect clientRect;
  GetClientRect(hwnd, &clientRect);
  TRect logicalRect = clientRect;
  DPtoLP(dc, (POINT*)&logicalRect, 2);

  // loop through all the parts and draw each one
  ViewData& viewData = GetViewData(hwnd);
  for (TOcPartCollectionIter i(viewData.OcDoc->GetParts()); i; i++) {
    TOcPart& part = *i.Current();
    if (part.IsVisible(logicalRect)) {
      TRect rect = part.GetRect();
      part.Draw(dc, rect, clientRect, asDefault);
      if (metafile)
        continue;

      // If an object is selected, draw whatever mark indicates that state
      if (part.IsSelected()) {
        // Draw some XOR rectangle around 'rect'
      }
    }
  }
  return true;
}
```

CPPOCF1 is a very simple container. Because it holds only one embedded object at a time, it doesn't really have to create a loop to handle painting multiple parts. If it expanded to permit multiple objects, however, it would not have to change its paint procedure.

The *TOcPart* class manages linked or embedded objects in a container document. The *TOcPart::Draw* method asks the server to draw its object. The *Draw* method does not need to be told the position of the object. The *TOcPart* object receives that information when it is constructed, as you will see in "Handling selected application events."

The **for** loop creates an iterator object to enumerate all the parts in the document. The ++ operator advances the iterator to point to successive parts. The expression **i.Current()* evaluates to a different part each time through the loop.

See Also
TOcPart

Activating and Deactivating Objects

Creating a View Window

After embedding an object into a compound document, the user might decide to edit the object. In most containers, the user activates an object by double-clicking it. CPPOCF1 does not support activating objects, but the code to do it is straightforward. You intercept WM_LBUTTONDBLCLK messages, check the mouse coordinates, and if they fall on an object you activate it.

To enumerate the document's embedded and linked objects, use a **for** loop with a *TOcPartCollectionIter* object, as the paint procedure does to draw all the parts. To find the coordinates of an object, call *TOcPart::GetRect*. To activate a part, call *TOcPart::Activate*.

Before a container closes a compound document, it should always check that no object is activated. CPPOCF1 includes this loop in the WM_DESTROY handler of its view window:

```
for (TOcPartCollectionIter i(OcDoc->GetParts()); i; i++) {
  TOcPart& p = *i.Current();
  p.Activate(false);
}
```

Passing **false** to Activate terminates any editing session.

Programming the Main Window

Turning a C++ Application Into an OLE Container

The view window manages tasks related to a single document. It opens and closes the document and draws it on the screen. The frame window manages tasks for the whole application. It responds to menu commands, and it creates and destroys the view window.

The following topics discuss programming the main window:

- Creating the Main Window
- Handling WM_OCEVENT
- Handling Selected Application Events
- Handling Standard OLE Menu Commands

Creating the Main Window

Programming the Main Window

When the application creates its main window, it must bind the window to its *TOcApp* object. (The *TOcApp* object was created in the factory callback function.)

```
bool
MainWnd_OnCreate(HWND hwnd, CREATESTRUCT FAR* /*lpCreateStruct*/)
{
  HwndMain = hwnd;
  if (OcApp)
    OcApp->SetupWindow(hwnd);
  return true;
}
```

The *TOcApp* object sends messges about OLE events to the application's main window. *SetupWindow* tells the *TOcApp* where to direct its event notifications.

Handling WM_OCEVENT

Programming the Main Window

TOcApp sends event notifications in the WM_OCEVENT message. Like the view window, the frame window also must handle WM_OCEVENT. The frame window receives notification of events that concern the application as a whole, not just a particular document. The frame window procedure sends WM_OCEVENT messages to a handler that identifies the event and calls the appropriate handler routine. Both routines closely resemble the corresponding code for the view window.

```
// Standard message-handler routine for main window
long CALLBACK _export
MainWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
  switch (message) {
.
.
.
// other message crackers go here
    HANDLE_MSG(hwnd, WM_OCEVENT,  MainWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}

// Subdispatch OC_... messages
long
MainWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
{
  switch (wParam) {
    HANDLE_OCF(hwnd, OC_VIEWTITLE, MainWnd_OnOcViewTitle);
  }
  return true;
}
```

Handling Selected Application Events

The only ObjectComponents event that CPPOCF1 handles in its main window is OC_VIEWTITLE. This message is sent when a server engaged in open editing wants to use the container's title in its own window caption. The OLE user interface guidelines require the server to show the source of the object it is editing.

```
const char*
MainWnd_OnOcViewTitle(HWND /*hwnd*/)
{
  return APPSTRING;
}
```

Because CPPOCF1 always has only a single document, it returns the name of the application as the title of its view. Because it always has only one view, it can handle the OC_VIEWTITLE event in the main window procedure. Most containers handle this message in the view window and return the name of the application and the name of the document combined in a single string.

Handling Standard OLE Menu Commands

Programming the Main Window

An OLE container places OLE commands on its Edit menu. It's not necessary to use all of them. CPPOCF1 supports one, Insert Object. This command lets users add new objects to the current document.

```
void
MainWnd_OnCommand(HWND hwnd, int id, HWND /*hwndCtl*/, uint /*codeNotify*/)
{
  switch (id) {
    case CM_INSERTOBJECT: {
      try {
        TOcInitInfo initInfo(OcView);           // begin initializing info
structure
        if (OcApp->Browse(initInfo)) {          // show Insert Object dialog
box
          TRect rect(30, 30, 100, 100);             // only top and left are
used
          new TOcPart(*OcDoc, initInfo, rect);  // add new object to
document
        }
      }
      catch (TXBase& xbase) {
        MessageBox(GetFocus(), xbase.why().c_str(), "Exception caught",
MB_OK);
      }
      break;
    }
    case CM_EXIT: {
      PostMessage(hwnd, WM_CLOSE, 0, 0);
      break;
    }
  }
}
```

The code for inserting, dropping, or pasting an object into a document always begins with a *TOcInitInfo* structure. *TOcInitInfo* holds information describing the object about to be created: what container will receive it, whether to link or embed it, whether it already exists or will be newly created, and if it exists, where the data resides and in what format.

The constructor for *TOcInitInfo* receives a pointer to the view where you want the new object to appear. The next command, *OcApp->Browse*, invokes the standard Insert Object dialog box offering the user a choice of all the objects any server registered in the system can create. When the user chooses one, the *Browse* command places more information in *initInfo*.

The final step to insert a new OLE object is to create a *TOcPart* connector. *TOcPart* implements all the OLE services that a linked or embedded object is required to provide. It plugs into OLE, gets the data for the new object, adds itself to the list of parts in *OcDoc*, and draws itself on the screen in the position given by *TRect*.

For examples showing how to implement other OLE Edit menu commands, look at the source code for event handlers in OWL/OLEWINDO.CPP.

Building the Program

Turning a C++ Application Into an OLE Container

To build the server, you need to include the right headers, use a supported memory model, and link to the right libraries.

The following topics discuss those tasks:

- Including ObjectComponents Headers
- Compiling and Linking

Including ObjectComponents Headers

Building the Program

The following list shows the ObjectComponents headers for a container that does not use ObjectWindows.

```
#include <ocf/ocapp.h>       // TOcRegistrar, TOcModule, TOcApp (application
connector)
#include <ocf/ocdoc.h>       // TOcDocument (compound document manager)
#include <ocf/ocview.h>      // TOcView (document view connector)
#include <ocf/ocpart.h>      // TOcPart (linked/embedded object connector)
#include <ocf/ocfevx.h>      // WM_OCEVENT message crackers
```

Compiling and Linking

Building the Program

ObjectComponents applications that do not use ObjectWindows can use either the medium or large memory model. Link them with the OLE and ObjectComponents libraries.

To build CPPOCF0, CPPOCF1, and CPPOCF2, move to the program's directory and type this at the command prompt:

```
make MODEL=l
```

This command builds all three programs using the large memory model.

The make file that builds this example program refers to the OCFMAKE.GEN file.

Creating an Automation Controller

See Also

An *automation controller* is an application that controls a server's automated objects by sending commands to OLE for other programs to execute. Writing an automation controller is easier than writing a container, a server, or an automation object because sending commands doesn't require any user interface. You don't need to create any windows or use the Clipboard or draw objects on the screen.

In order to send commands to an OLE object, the automation controller must know the names of methods and properties the object's server exposes to OLE. Generally these names come from the server's type library. The controller uses the names in creating C++ proxy classes whose methods send commands to the server. It's possible to browse through available automation objects at run time and discover what commands they support, but to make use of commands discovered at run time usually requires a scripting language.

The following topics discuss creating an automation controller and enumerating items in an automated collection:

- Steps for Building an Automation Controller
- Enumerating Automated Collections

See Also
Automating an Application

Steps for Building an Automation Controller

See Also

These are the coding steps required to make one program control another.

1. Include header files
2. Create a TOleAllocator object
3. Declare proxy classes - to represent each OLE object you want to automate. Derive the classes from *TAutoProxy*.
4. Implement proxy classes
5. Create and use proxy objects
6. Compile and link

See Also
Enumerating Automated Collections

Including Header Files

See Also   Steps For Building An Automation Controller

An automation controller needs to include the following headers:

```
#include <ocf/autodefs.h>
#include <ocf/automacr.h>
```

The autodefs.h header defines automation classes such as *TAutoProxy*. The automacr.h header defines the macros a controller uses to implement proxy class methods.

See Also
ObjectComponents Header Files

Creating a TOleAllocator Object

See Also Steps For Building An Automation Controller

Like automation servers, automation controllers must also create a *TOleAllocator* object to initialize the OLE libraries and (optionally) to give OLE a memory allocation function. To create a *TOleAllocator* object, add this line to your program.

```
TOleAllocator OleAlloc;
```

The constructor for *TOleAllocator* initializes the OLE libraries and its destructor releases them. Create an object of type *TOleAllocator* before you begin OLE operations and be sure the object is not destroyed until all OLE operations have ended. A good place to create the *TOleAllocator* is at the beginning of *WinMain* or *OwlMain*.

See Also
TOleAllocator

Declaring Proxy Classes

See Also   Steps For Building An Automation Controller

A proxy class is a C++ stand-in for an automated OLE object. You create a proxy class whose interface corresponds to that of the OLE object. By deriving the proxy class from *TAutoProxy*, you connect it to ObjectComponents. When a *TAutoProxy* object is constructed, it calls an OLE API to request the *IDispatch* interface of the automated object that the proxy represents. When you call a function of the proxy class, the proxy sends the corresponding command to the automation server.

An automation controller declares one proxy class for every type of object it wants to control. in simple cases, a single proxy class might be enough. Controlling a complex application that creates several different kinds of automatable objects requires more proxies. To control a spreadsheet, for example, you might need a proxy application class, a proxy spreadsheet class, and a proxy cell class.

The easiest way to declare and implement proxy classes is with the AutoGen utility. AutoGen reads the server's type library and generates C++ source code for the proxy classes a controller needs to send any commands to the server. Simply compile the generated code into your application, construct proxy objects when you need them, and call their member functions to send commands.

As an example of a proxy class, here is the code that AutoGen generates for the automated class *TCalc* in the AutoCalc sample program. The opening comment shows descriptive information from AutoCalc's entries in the registration database including the value of AutoCalc's *version*, *clsid*, and *description* registration keys. The comments for individual members show the documentation strings that AutoCalc assigns to each member in its automation definition table, the dispatch ID that ObjectComponents assigned to identify each command, and whether the member is a function or a property.

```
// TKIND_DISPATCH: TCalc 1.2 {877B6207-7627-101B-B87C-0000C057CE4E}\409
// Automated Calculator Class
class TCalc : public TAutoProxy {
  public:
    TCalc() : TAutoProxy(0x409) {}
    // Pending operand
    long GetOperand();     // [id(1), prop r/w]
    void SetOperand(long); // [id(1), prop r/w]
    // Calculator accumulator
    long GetAccumulator();     // [id(0), prop r/w]
    void SetAccumulator(long); // [id(0), prop r/w]
    // Pending operation
    TAutoString GetOp();     // [id(3), prop r/w]
    void SetOp(TAutoString); // [id(3), prop r/w]
    // Evaluate operand, op
    TBool Evaluate(); // [id(4), method]
    // Clear accumulator
    void Clear(); // [id(5), method]
    // Update display
    void Display(); // [id(6), method]
    // Terminate calculator
    void Quit(); // [id(7), method]
    // Button push sequence
    TBool Button(TAutoString Key); // [id(8), method]
    // Calculator window
    void GetWindow(TCalcWindow&); // [id(9), propget]
    // Test of object as arg
    long LookAtWindow(TCalcWindow& Window); // [id(10), method]
    // Array as collection
    void GetArray(TCalcArray&); // [id(11), propget]
    // Application object
    void GetApplication(TCalc&); // [id(12), propget]
};
```

The constructor of an automation proxy class must pass to its base class, *TAutoProxy*, a number representing a locale setting. The locale tells what language the automation controller uses when it sends commands to objects. in the example, the number is 0x409, which is the locale ID for American English. AutoGen chooses this locale by reading the system settings when it runs, but you are free to change it to whatever locale you prefer.

The function members of class *TCalc* each send a different command to the calculator object. Read-write properties get two commands, one for getting the value and one for setting it. *GetOp* and *SetOp*, for example, write and read the value representing the next operation the calculator will perform. Other commands, such as *Display* and *Quit*, make the calculator perform some action..

See Also

TAutoProxy

TLocaleId

Implementing Proxy Classes

See Also Steps For Building An Automation Controller

Simply declaring methods doesn't accomplish much, of course. You also have to implement them. Each method must send a command through ObjectComponents to the automated object. Here is part of the implementation code that AutoGen generates for the *TCalc* proxy object. Every method simply calls the same three macros.

```
// TKIND_DISPATCH: TCalc 1.2 {877B6207-7627-101B-B87C-0000C057CE4E}\409
Automated Calculator Class
TAutoString TCalc::GetOp()
{
  AUTONAMES0("Op")
  AUTOARGS0()
  AUTOCALL_PROP_GET
}
void TCalc::SetOp(TAutoString val)
{
  AUTONAMES0("Op")
  AUTOARGS0()
  AUTOCALL_PROP_SET(val)
}
TBool TCalc::Evaluate()
{
  AUTONAMES0("Evaluate")
  AUTOARGS0()
  AUTOCALL_METHOD_RET
}
void TCalc::Clear()
{
  AUTONAMES0("Clear")
  AUTOARGS0()
  AUTOCALL_METHOD_VOID
}
void TCalc::Display()
{
  AUTONAMES0("Display")
  AUTOARGS0()
  AUTOCALL_METHOD_VOID
}
void TCalc::Quit()
{
  AUTONAMES0("Quit")
  AUTOARGS0()
  AUTOCALL_METHOD_VOID
}
void TCalc::GetWindow(TCalcWindow& obj)
{
  AUTONAMES0("Window")
  AUTOARGS0()
  AUTOCALL_PROP_REF(obj)
}
```

The three macros supply all the code needed for each function. The first two macros, AUTONAMES and AUTOARGS, specify what arguments you want to pass. They are explained in more detail below. None of the methods in the example takes any arguments. The AUTOCALL_*xxxx* macros tell whether the command is a function or a property and what kind of value it returns.

Macro	Description
AUTOCALL_METHOD*n*(id, arg...)	Calls a method with *n* arguments that returns a value.
AUTOCALL_METHOD*n*V(id, arg...)	Calls a method with *n* arguments that returns void.
AUTOCALL_METHOD*n*_REF(id, prx, arg...)	Calls a method with *n* arguments that returns a proxy object.
AUTOCALL_PROPGET(id)	Retrieves the value of a property.
AUTOCALL_PROPSET(id, arg)	Assigns a value to a property.
AUTOCALL_PROPREF(id, obj)	Retrieves the value of a property that contains an object. (Objects must be passed by reference.)

Note: When an automation command passes an object as a parameter or a return value, be sure to pass by reference, not by assignment. for example, access functions for a property implemented as an object should follow this form:

```
GetObjectX( X& obj );
SetObjectX( X& obj );
```

Passing objects by assignment makes it impossible to provide C++ type safety.

See Also

AUTOARGS Macros

AUTOCALL_xxxx Macros

AUTONAMES Macros

Specifying Arguments in a Proxy Method

Specifying Arguments in a Proxy Method

See Also Implementing Proxy Classes

The first two macros in the implementation of a proxy method indicate what arguments you intend to pass. The server can decide that some arguments to a method are optional. You must pass all required arguments, and you can choose to pass any subset of the optional arguments.

For example, a server might expose a method that takes ten arguments, of which five are optional. Optional arguments have default values. Your controller might have a use for only one of the optional arguments, always accepting the default values for the other four. in that case, you can set up your proxy implementation so that you have to pass only six arguments instead of ten.

The AUTONAMES macro lists any optional arguments that you do want to use. It lists them by the names the server assigns to them. (AutoGen reads the names from the server's type library for you.) If you intend to pass only one of five optional arguments, then you list only one argument in AUTONAMES.

The first argument passed to an AUTONAMES macro always indentifies the automation method that this proxy command invokes. The names of arguments come after. If the automation server uses ObjectComponents, then the names used in AUTONAMES come from the server's automation definition table. The function name is the external name in an EXPOSE_METHOD macro, and the argument names come from subsequent OPTIONAL_ARG macros.

The second parameter in a proxy method implementation, AUTOARGS, lists all the arguments that the controller chooses to pass for this command. It tells what will be pushed onto the command stack. AUTOARGS must always list all the required arguments in order first. At the end of the list come any optional arguments from the AUTONAMES macro. If there are five required arguments and the controller wants to pass only one of five optional arguments, then the list in AUTOARGS includes six arguments, the optional one last.

The names used for required arguments are just dummy names. Their position in the list indicates which argument they represent. The names for optional arguments must be the same names used in AUTONAMES. for optional arguments, the name itself is what identifies a particular parameter.

See Also

AUTOARGS Macros

AUTONAMES Macros

EXPOSE_METHOD Macros

OPTIONAL_ARG Macro

Writing Definition Macros

Creating and Using Proxy Objects

See Also Steps For Building An Automation Controller

Through a proxy class you can talk to an OLE object, but first the object has to exist. The *TAutoProxy* class defines a member function called *Bind* that asks OLE to create an object. The parameter passed to *Bind* determines the type of object to create. The most convenient identifier is usually a name the automation object has recorded in the registration database. (the object's unique clsid number also works but is harder to remember and write.) This is what an automation controller does to make OLE create a calculator object:

```
TCalc calc;                            // create proxy object
calculator.Bind("Calc.Application");   // make OLE create real object
```

The string passed to *Bind* is what the automation server registered as its progid:

```
REGDATA(progid, "Calc.Application")    // from server's registration table
```

The destructor for *TAutoProxy* calls the *Unbind* method, so when *calculator* goes out of scope, OLE destroys the actual calculator object.

While *calculator* remains in scope, the controller program issues commands by calling methods on the proxy object. The commands in the following example add 1234 + 4321 and display the result in the calculator's window.

```
calc.SetOperand(1234);
calc.SetOp("Add");
calc.Evaluate();
calc.SetOperand(4321);
calc.Button("+");
calc.Evaluate();
calc.Display();
```

See Also
REGDATA Macro (OWL.HLP)
TAutoProxy::Bind
TAutoProxy::Unbind

Compiling and Linking

Steps For Building An Automation Controller

Automation servers and controllers must be compiled with the medium or large memory model. (They run faster in medium model.) They must be linked with the OLE and ObjectComponents libraries.

The integrated development environment (IDE) chooses the right build options for you when you ask for OLE support. To build any ObjectComponents program from the command line, create a short makefile that includes the OCFMAKE.GEN file found in the EXAMPLES subdirectory.

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj myprogram.obj
!include $(BCEXAMPLEDIR)\ocfmake.gen
```

EXERES and OBJRES hold the name of the file to build and the names of the object files to build it from. The last line includes the OCFMAKE.GEN file. Name your file MAKEFILE and type this at the command line prompt:

```
make MODEL=m
```

MAKE, using instructions in OCFMAKE.GEN, will build a new makefile tailored to your project. The new makefile is called WIN16Mxx.MAK.

Enumerating Automated Collections

See Also

Many automated objects have properties that represent a set of related itemsfor example, integers in an array, structures in a linked list, or a group of objects such as the buttons on the face of the calculator. To expose a collection, the automation server must implement a collection object with access functions. As OLE sees it, a collection object implements the standard *IEnumVARIANT* interface.

The following topics explain what a controller must do to use a collection object and enumerate items in the server:

1. Declare a proxy collection class
2. Implement the proxy collection class
3. Declare a collection property
4. Send commands to the collection

See Also
Steps For Building An Automation Controller

Declaring a Proxy Collection Class

See Also Enumerating Automated Collections

A proxy collection class usually supplies member functions to find out how many items are in the collection, to retrieve individual items randomly by their position in the list, and to enumerate the items in the list sequentially. (on the server's side, ObjectComponents calls this iterating. The controller uses the server's iterator to enumerate the items.)

Here is the proxy class that AutoGen creates to enumerate the collection of calculator buttons in AutoCalc.

```
// TKIND_DISPATCH: TButtonList 1.2 {877B6204-7627-101B-B87C-0000C057CE4E}\
409
// Button Collection
class TButtonList : public TAutoProxy {
  public:
    TButtonList() : TAutoProxy(0x409) {}
    // Button Count
    long GetCount(); // [id(1), propget]
    // Button Iterator
    void Enumerate(TAutoEnumerator<TCalcButton>&); // [id(-4), propget]
    // Button Collection Item
    void Item(TCalcButton&, short Index); // [id(0), method]
};
```

The only thing here that wasn't in the previous proxy classes is the use of the *TAutoEnumerator* template. *TAutoEnumerator* encapsulates the code for manipulating the *IEnumVARIANT* interface of a collection object. The type you pass to the template is the type of value the collection contains. In the example, *TCalcButton* is another proxy class representing an automated button object in the server.

See Also
TAutoEnumerator

Implementing the Proxy Collection Class

See Also		Enumerating Automated Collections

This is the code that AutoGen writes to implement the proxy collection class. *Count* and *Item* are straightforward. The *Enumerate* method does several new things, however.

```
// TKIND_DISPATCH: TButtonList 1.2 {877B6204-7627-101B-B87C-0000C057CE4E}\
409 Button Collection
long TButtonList::GetCount()
{
  AUTONAMES0("Count")
  AUTOARGS0()
  AUTOCALL_PROP_GET
}
void TButtonList::Enumerate(TAutoEnumerator<TCalcButton>& obj)
{
  AUTONAMES0(DISPID_NEWENUM)
  AUTOARGS0()
  AUTOCALL_PROP_REF(obj)
}
void TButtonList::Item(TCalcButton& obj, short Index)
{
  AUTONAMES0("Item")
  AUTOARGS1(Index)
  AUTOCALL_METHOD_REF(obj)
}
```

First, the parameter to the *Enumerate* method is a reference to an object of the type that the collection contains. on successive calls, *Enumerate* returns collection items through this parameter. The data type for the parameter must use the *TAutoEnumerator* template.

Second, the method is identified to AUTONAMES0 as DISPID_NEWENUM. This is a predefined constant from oleauto.h representing the standard dispatch ID (which happens to be 4) for an enumerating command. The AUTONAMES0 macro accepts a dispatch ID instead of a function name. (the other AUTONAMES macros, those that expect argument names as well, require a name string for the function.)

Finally, an enumerator is a property of its object and it passes an object by value, so it the enumerator implementation ends with the AUTOCALL_PROP_REF macro.

See Also

AUTOCALL_xxxx Macros

AUTONAMES Macros

Declaring a Collection Property

Enumerating Automated Collections

TButtonsList is now fully defined as a proxy class for the server's collection object. What's needed now is a way to ask the controller for the collection. In AutoCalc, the collection of buttons is a property of the calculator's automated window object.

```
class TCalcWindow : public TAutoProxy {
  public:
    TCalcWindow() : TAutoProxy(0x409) {}
    // Button Collection
    void GetButtons(TButtonList&); // [id(5), propget]
```

The window class exposes the collection through a *GetButtons* command that returns the value of the collection property. *GetButtons* needs the *TButtonList* class to declare its parameter type.

Sending Commands to the Collection

Enumerating Automated Collections

This code from the sample program CallCalc sends the calculator commands that press its buttons. in the code, *window* is the automated window object. *TCalcButton* is the proxy class for individual buttons. *TButtonList* is the proxy object for the collection.

```
TButtonList buttons;                  // declare a collection object
window.GetButtons(buttons);           // bind buttons to automated
collection object
TAutoEnumerator<TCalcButton> list;    // create an enumerator of
TCalcButton objects
buttons.Enumerate(list);              // bind list to the server's
iterator
TCalcButton button;                   // declare a button object

// list.Step advances to the next item in list
for (i = IDC_FIRSTBUTTON; list.Step(); i++) {
  list.Object(button);                // bind button to an automated
button object
  button.SetActivate(true);           // press the calculator button
}
```

The *buttons*, *list*, and *button* variables are each created in one step and then bound to a server object in another. Each of them is a proxy object for something the server created; *buttons*, for example, is the proxy object for a collection of automated button objects. Simply declaring a proxy object, however, does not attach it to any particular automated object in the server. To be able to send commands, a proxy object must be bound to something with an automation interface (*IDispatch* or *IEnumVARIANT*). Because the server defines the collection of buttons as a property of the calculator's window, this command retrieves the collection and connects it to the buttons proxy object:

```
window.GetButtons(buttons);  // bind buttons to automated collection object
```

The two other lines where the comments indicate binding takes place similarly connect *list* to the collection's iterator object and *button* to individual button objects in the collection.

A simple assignment statement might seem more intuitive than the binding step, but the only value that could be assigned in these cases is simply a pointer to an automation interface. A pointer carries no type information; a pointer to a collection's *IDispatch* looks just like the pointer to a button's *IDispatch*. Binding to an existing C++ object preserves information about what kind of automation object it represents.

Support for OLE in Borland C++

ObjectComponents is a set of classes for creating OLE 2 applications in C++. The following topics discuss questions you may have about using Borland C++ to create OLE 2 application:

- What does ObjectComponents do?
- Where Should You Start?
- What is OLE?
- What does OLE look like?
- What is ObjectComponents?
- How do I use ObjectComponents?
- How does ObjectComponents work?
- What ObjectComponents programming tools are available?
- Where do I look for information?
- What do these new OLE terms mean?

What Does ObjectComponents Do?

See Also Support for OLE in Borland C++

ObjectComponents makes OLE programming easy. It supports all the following OLE 2 capabilities:

- Linking
- Embedding
- In-place editing
- Drag-and-drop operation
- OLE Clipboard operations
- Compound document storage
- Automation servers and controllers
- DLL servers
- Localization
- Registration

Using ObjectComponents offers these features::

- an easy upgrade path to linking and embedding for existing C++ applications, especially if they use ObjectWindows
- Easy automation of existing C++ applications, whether or not they use ObjectWindows
- Default implementations of standard OLE 2 user interface elements, such as the Insert Object and Paste Link dialog boxes
- The ability to create OLE 2 applications with AppExpert, which generates and understands the new OLE classes
- Compatibility with other OLE applications, including OLE 1 applications, whether or not they were built with Borland tools
- Virtually no operating overhead imposed on ObjectWindows applications that choose not to use OLE

See Also
OLE 2 Features Supported by ObjectComponents

Where Should You Start?

See Also Support for OLE in Borland C++

That depends on what you want to know and what application you want to create. The following topics offer introductory discussions:

- Writing Applications
- Learning About Object Components

See Also
Where Do I Look for Information?

Writing Applications

Where Should You Start?

The right starting place depends on whether you are creating a new application or adapting an existing one.

The following topics discuss your choices:

- Creating a New Application
- Converting an Application Into an OLE Container
- Converting Application Into a Linking and Embedding Server
- Adding Automation Support
- Other Useful Topics

Creating a New Application

Writing Applications

You can generate a complete OLE application almost instantly using AppExpert. AppExpert fully supports all the new features of ObjectComponents. To start an OLE application from scratch, simply choose Project|AppExpert from the IDE menu. For more information about AppExpert, consult the *User's Guide*.

The programs AppExpert creates use the ObjectWindows Library. If you are new to ObjectWindows, begin with the *ObjectWindows Tutorial* book.

Converting an Application into an OLE Container

Writing Applications

Where you should start depends on whether your application uses ObjectWindows, and if so, whether it uses the Doc/View model.

The following table gets you started:

ObjectWindows?	Doc/View?	Where to start
Yes	Yes	Turning a Doc/View Application Into an OLE Container
Yes	No	Turning an Objectwindows Application Into an OLE Container
No	No	Turning a C++ Application Into an OLE Container

Converting Application into a Linking and Embedding Server

Writing Applications

Where you should start depends on whether your application uses ObjectWindows, and if so, whether it uses the Doc/View model.

The following table gets you started:

ObjectWindows?	Doc/View?	Where to start
Yes	Yes	Turning a Doc/View Application Into an OLE Server
Yes	No	Turning an Objectwindows Application Into an OLE Server
No	No	Turning a C++ Application Into an OLE Server

Adding Automation Support

Writing Applications

The process of adding automation support to an existing application is the same regardless of whether the application uses ObjectWindows or the Doc/View model. For help creating an automation server, a controller, or a type library, select the indicated topic.

- **Automation server**: Automating an Application
- **Automation client**: Creating an Automation Controller
- **Type library**: Creating a Type Library

Other Useful Topics

Writing Applications

Here are some topics common to different kinds of OLE applications. For help with them, select the indicated topic.

- **DLL server**: Making a DLL Server
- **Localization**: Localizing Symbol Names
- **Registration**: Building Registration Tables
- **Compiling and linking**: Building an ObjectComponents Application
- **Exception handling**: Exception Handling in ObjectComponents

Learning about ObjectComponents

Where Should You Start?

The following topics help you become familiar with ObjectComponents.

Understanding OLE

- What is OLE? - an introduction to OLE
- What does OLE look like? - illustrations showing how common OLE interactions look onscreen

Surveying the new classes

- Using ObjectComponents - summarizing new classes and messages in ObjectComponents and ObjectWindows

Understanding how ObjectComponents works

- How ObjectComponents Works - how ObjectComponents classes mediate between OLE and your own C++ classes

Finding example programs

- Example Programs - a brief description of some of the sample ObjectComponents programs in Borland C++

Finding the right documentation

- Books and Online Help - all the parts of the documentation that describe ObjectComponents

Understanding terms

- Glossary of OLE terms - definitions of terms used in the documentation. Skimming the glossary is also a good way to introduce yourself to the features of ObjectComponents.

What Is OLE?

See Also Support for OLE in Borland C++

OLE, which stands for object linking and embedding, is an operating system extension that lets applications achieve a high degree of integration. OLE defines a set of standard interfaces so that any OLE program can interact with any other OLE program. No program needs to have any built-in knowledge of its possible partners.

Programmers implement OLE applications by creating objects that conform to the Component Object Model (COM). COM is the specification that defines what an OLE object is. COM objects support interfaces, composed of functions for other objects to call. OLE defines a number of standard interfaces. COM objects intended for public access expose their interfaces in a registration database. Interfaces have unique identifiers to distiguish them.

ObjectComponents encapsulates the COM specification for creating objects and provides default implementations of the interfaces used for two common OLE tasks: linking and embedding, and automation. *Linking* and *embedding* lets one application incorporate live data from other OLE applications in its documents. Automation lets one application issue commands to control another application.

The following topics discuss common uses for OLE.

- Linking and Embedding
- Automation

See Also
Glossary of OLE Terms

Linking and Embedding

See Also What is OLE?

Linking and *embedding* refer to the transfer of data from one program to another. The first program, the server, sends its data to the second program, the *container*. For example, cells from a spreadsheet can be dropped into a word processing document. Of course you don't need OLE to pass data from one Windows program to another. You can do that much with just the Clipboard. The difference between OLE and the Clipboard is that in OLE the receiving program doesn't have to know anything at all about the format of the data in the object. Any OLE server application can give its data to any OLE container application. Thanks to OLE, the container doesn't care whether the object it receives is a metafile, a bitmap, or ASCII text. The server passes whatever data it uses internally and the container accepts it. Furthermore, the object remains dynamic even after being transplanted. When the container wants to display, modify, or save the object, it calls OLE to do it. OLE, working behind the scenes, calls the server to execute the user's command. The object belongs to the container's document, but OLE maintains a live connection back to the server. The user can continue to edit the object using all of the server's tools. As a result, the user can combine objects from different servers into a single document without losing the ability to update and modify any object as the document evolves.

See Also
Overview of Linking and Embedding Classes

automation

See Also What is OLE?

Automation happens when one program issues commands to another. If you write a calculator program, for example, you might allow other programs to issue commands like these:

- Press the nine button.
- Press the plus button.
- Press the six button.
- Press the equals button.
- Tell me what's in the Total window.

These are commands a person might normally issue through the calculator's user interface. With automation, the calculator exposes its internal functions to other programs. The calculator becomes an *automation object*, and programs that send commands to it are automation controllers. OLE defines standard interfaces that let a controller ask any installed server to create one of its objects. OLE also makes it possible for the controller to browse through a list of automated commands the server supports and execute them.

See Also
Overview of Automation Classes

What Does OLE Look Like?

See Also Support for OLE in Borland C++

The linking and embedding features of OLE include a standard user interface for performing common operations such as placing OLE objects in container documents and activating them once they are linked or embedded. The OLE standards cover menu commands, dialog boxes, tool bars, drag and drop support, and painting conventions, so that the user interface for OLE operations is consistent across applications. Together, ObjectComponents and ObjectWindows execute most of the interface tasks for you.

Understanding OLE programming can be difficult without a clear grasp of the interface you are trying to create. The following sections present pictures of a container showing what happens onscreen at each step in a common sequence of OLE operations. The user runs a container, inserts objects from several OLE servers into the document, edits an object, and saves the document. The descriptions of these steps introduce the following OLE features:

- Inserting an Object
- Editing an Object in Place
- Activating Deactivating and Selecting an Object
- Finding an Object's Verbs
- Linking an Object
- Opening an Object to Edit it

See Also
Glossary of OLE Terms

Inserting an Object

What Does OLE Look Like?

The example program called SdiOle is an OLE container using the single-document interface (SDI) and written with ObjectWindows and ObjectComponents. The source code for SdiOle is in EXAMPLES/OWL/OCF/ SDIOLE.

The SdiOle Edit menu contains five standard OLE commands that most containers possess: Paste Special, Paste Link, Insert Object, Links, and Object.

ObjectWindows implements all five of the standard commands for you if you like, but a container does not have to use them all.

Like the common dialog boxes in Windows for opening files and choosing fonts, the Insert Object dialog box is a standard resource implemented by the system. For consistency, it is best to use the standard dialog boxes unless your application has some unusual requirement that the standard dialog box does not meet.

The box under Object Type lists all the kinds of objects available in the system. Whenever a server installs itself, it tells the system what objects it can create. The system keeps this information in its registration database. The Insert Object dialog box queries the database and shows all the types that OLE can create for you using the available server applications.

In the illustration, the user has chosen to insert a Quattro Pro spreadsheet. The Result box at the bottom of the dialog box explains what will happen if the user clicks OK now. Because the Create New button is selected, clicking OK will embed a new, empty spreadsheet object into the user's open document.

Editing an Object in Place

What Does OLE Look Like?

The example program called SdiOle is an OLE container using the single-document interface (SDI) and written with ObjectWindows and ObjectComponents. The source code for SdiOle is in EXAMPLES/OWL/OCF/ SDIOLE.

The SdiOle Edit menu contains five standard OLE commands that most containers possess: Paste Special, Paste Link, Insert Object, Links, and Object.

If Quattro Pro is the server application that creates the active object, Quattro Pro will take over the SdiOle window and display its own menus and tool bars. All the Quattro Pro menu and tool bar commands can be executed right there in SdiOle. The feature of OLE that lets a server take over a container's main window is called *in-place editing*. It lets the user edit the object in its place, without switching back and forth between different windows. The programming task that makes this possible is called *menu merging*, combining menus from two programs in one menu bar.

Although many programs let you paste data from other programs into your documents, without OLE you cannot continue to edit the objects after they are transferred.

SdiOle is a very simple application and knows nothing about columns and rows or fonts and shading. But, even though the Quattro Pro server created and formatted the object, that data in the object belongs to the container. When the user chooses File|Save from the SdiOle menu bar, what gets written is an SdiOle document, not a Quattro Pro document. With the help of ObjectComponents and the OLE system, SdiOle marks an area in its own file to store the data for the embedded object. When the user chooses File|Load to reload the same document, the spreadsheet cells will still be there. If the user tries to edit the object again, OLE invokes Quattro Pro to take over the SdiOle window once more. The object remains associated with the application that created it even though the object is stored in a foreign file.

When OLE places the data for an object directly into the container's document as it has the data for this spreadsheet, the object is said to be embedded. Besides embedding, OLE also links objects to container documents.

Activating, Deactivating, and Selecting an Object

What Does OLE Look Like?

The example program called SdiOle is an OLE container using the single-document interface (SDI) and written with ObjectWindows and ObjectComponents. The source code for SdiOle is in EXAMPLES/OWL/OCF/ SDIOLE.

The SdiOle Edit menu contains five standard OLE commands that most containers possess: Paste Special, Paste Link, Insert Object, Links, and Object.

An embedded object is outlined by a thick gray rectangle. The presence of this rectangle indicates that the object is active. The activation rectangle appears when you double-click the object. Usually activating an object initiates an editing session, but the server decides whether to follow that convention. For example, embedded sound objects might play when activated. In most cases, only one object can be active at a time.

When an activation rectangle has small black boxes spaced around it, they are called *grapples*. The user can resize the object by clicking a grapple and dragging the mouse. Also, the user can move the object by clicking anywhere else on the activation rectangle and dragging. ObjectWindows uses the *TUIHandle* class to draw rectangles and grapples around objects.

When the user clicks the mouse button outside the activated object, the activation rectangle goes away. The object is now *inactive*. Deactivating an object tells OLE that you are through editing. The server relinquishes its place, and the container's window returns to normal. The only commands on the menu bar are the ones SdiOle put there. The tool bar and window caption are back to normal, as well.

You can *select* an inactive object without activating it. When you press the mouse button over an inactive object, the container draws a thin black rectangle to show that you have selected it. Like the activation rectangle, it has grapples. The user can move and resize a selected object just like an active object.

Finding an Object's Verbs

What Does OLE Look Like?

The example program called SdiOle is an OLE container using the single-document interface (SDI) and written with ObjectWindows and ObjectComponents. The source code for SdiOle is in EXAMPLES/OWL/OCF/ SDIOLE.

The SdiOle Edit menu contains five standard OLE commands that most containers possess: Paste Special, Paste Link, Insert Object, Links, and Object.

When an object is selected, the container modifies its menus to offer a choice of whatever actions the object's server can do with the object. OLE calls these actions verbs. Conventionally, the container displays available verbs in two places: on its Edit menu and on a SpeedMenu. For example, if a SpeedMenu pops up when the user right clicks, the first three commands on the SpeedMenu are always Cut, Copy, and Delete. The fourth item, Notebook Object, changes depending on the opject selected. When an opbect from Paradox is inserted, for example, the fourth item becomes Paradox 5 Object.

The smaller cascading menu lists the particular verbs that the server supports. Quattro Pro has only two verbs. It can edit an object or open an object. The Edit verb initiates an in-place editing session The Open verb inititates an open editing session.

The final item, Convert, is the same for all objects. It invokes another standard OLE dialog box that lets the user convert an object from one server's data format to another. The Convert command is useful when, for example, you have Paradox installed on your machine, but someone gives you a compound document with an embedded object from some other database application. If Paradox knows how to convert data from the other database, then the Convert command binds the foreign database object to Paradox.

The speed menu for a selected object shows where verbs appear on the Edit menu. When no object is selected, the last command on the Edit menu is disabled and says simply Object. When an object is selected, the Object command changes to describe the selected object. In the example, Object changes to Notebook Object.

Linking an Object

What Does OLE Look Like?

The example program called SdiOle is an OLE container using the single-document interface (SDI) and written with ObjectWindows and ObjectComponents. The source code for SdiOle is in EXAMPLES/OWL/OCF/ SDIOLE.

The SdiOle Edit menu contains five standard OLE commands that most containers possess: Paste Special, Paste Link, Insert Object, Links, and Object.

By default, the Insert Object command creates a brand new empty object, and embeds it. Instead of embedding an object, you can choose to link it.

When OLE links an object, it does not store the object's data in the container's document. It stores only the name of the server file where the data is stored along with the location of the data within the file and a snapshot of the object as it appears onscreen. The snapshot is usually a metafile. The container doesn't receive a copy of the object; it receives a pointer to the object. OLE still draws the object in the container's document, just as though it was embedded, but the container doesn't own the data.

If the server document that holds the data for the linked object is deleted, then the user can no longer activate and edit the linked object. On the other hand, if the data in the server document is updated, then the updates appear automatically in all the container documents that have been linked to the same object. If several documents *embed* the same object, then they are creating copies, and changes made in one document have no effect on the copies in other documents.

What if you select the Create from File button in the Insert Object dialog box? Instead of creating a new empty object, you choose a file with existing data and OLE invokes the server that created the file. You can embed data from the file, but if the user has checked the Link box, when the user clicks OK, OLE does not copy data from CHECKS.DB into the server's document. It creates a link that refers back to the data stored in the original file.

The text in the Result box at the bottom of the dialog box explains what will happen when the user clicks OK. The EXAMPLE.DFL document now contains two OLE objectsthe embedded Quattro Pro spreadsheet and the linked Paradox table. Neither of the two objects is active. The spreadsheet is inactive and the database table is selected. Because the database table is linked, ObjectWindows draws the selection rectangle with a dashed line.

Opening an Object to Edit It

What Does OLE Look Like?

The example program called SdiOle is an OLE container using the single-document interface (SDI) and written with ObjectWindows and ObjectComponents. The source code for SdiOle is in EXAMPLES/OWL/OCF/SDIOLE.

The SdiOle Edit menu contains five standard OLE commands that most containers possess: Paste Special, Paste Link, Insert Object, Links, and Object.

TheEdit and Open are the two most common verbs, and Quattro Pro and Paradox, for example, both use them. Choosing the Open verb   makes the same table visible in two windows - the container window where it is linked and the server window where it is being edited. When finished editing in the server window, the user chooses File|Close and returns to the container. Any changes made during the editing session automatically appear in the container window afterward.

Contrast this editing session with another. In this session, the container window remains unchanged. The SdiOle window has only its own commands and its own tool bar. The editing takes place in a separate window that OLE opened just for this session. Returning to the server to edit is called open editing. Some servers support only open editing, not in-place editing.

These are common linking and embedding operations: the user links or embeds an object, selects it, activates it, edits it in place or open, and saves the compound document complete with its OLE object. The standard way is to link and embed objects with the Insert Object dialog box, but there are other ways as well. The Paste, Paste Special, and Paste Link commands can all create OLE objects from data on the Clipboard. You can also link or embed objects by dragging them from one applicaton and dropping them on another.

What Is ObjectComponents?

See Also Support for OLE in Borland C++

ObjectComponents features

For a listing of ObjectComponents features, see OLE 2 Features Supported by ObjectComponents.

The Borland OLE 2 Support Library

Microsoft's OLE 2 operating system extensions require the programmer to implement a variety of interfaces depending on the tasks an application undertakes. Borland has developed an OLE engine, already used in several of its commercial applications, that simplifies the programmer's job by implementing a smaller set of high-level interfaces on top of OLE. The engine resides in a library called BOCOLE.DLL. The BOCOLE support library provides default implementations for many standard OLE interfaces.

C++ programmers can make use of the OLE support in BOCOLE.DLL through a set of new classes collectively called the ObjectComponents Framework (OCF). Instead of implementing OLE-style interfaces, you create objects from the ObjectComponents classes and call their methods. Your own classes can gain OLE capabilities simply by inheriting from the ObjectComponents classes. ObjectComponents translates between C++ and OLE.

Interacting with OLE 2

The ObjectComponents classes implement OLE-style interfaces for talking to the BOCOLE support library. Your programs reach OLE by calling methods from ObjectComponents classes. When OLE sends information to you, ObjectComponents sends messages to your application using the standard Windows message mechanisms. The ObjectComponents classes also contain default implementations for all the OLE messages. You can override the default event handlers selectively to modify your application's responses.

ObjectComponents is not part of the ObjectWindows Library. That means C++ programs that don't use ObjectWindows can still take full advantage of ObjectComponents for linking, embedding, and automation. But ObjectWindows can simplify your work even more. ObjectWindows 2.5 introduces new classes such as *TOleWindow* and *TOleDocument* that inherit from ObjectComponents classes to bring OLE support into Borland's C++ application framework. An ObjectWindows application that uses the Doc/View model doesn't need to use ObjectComponents directly at all. A few simple changes to your Doc/View program will have you linking and embedding almost instantly. Programs that don't use the Doc/View model can do the same thing with just a little more work.

See Also
What is OLE?

OLE 2 Features Supported by ObjectComponents

See Also What is ObjectComponents?

The following list summarizes the OLE 2 capabilities that ObjectComponents gives your applications. The descriptions assume you are using ObjectWindows, as well. All the same features are available through ObjectComponents without ObjectWindows, but then you have to code explicitly some things that ObjectWindows does by default.

- **Linking and embedding**: to embed data from one application in the document of another, ObjectComponents gives you classes to represent the data in the object and an image of the data for drawing on the screen. The data must be separable from its graphical representation because in OLE transactions they are sometimes handled by different programs. When the container asks the server for an object to embed, the server must provide data and a view of the data. The server can also be asked to edit the object even after it is embedded and to read or write the object to and from the container's document file. The ObjectComponents classes handle both sides of these negotiations for you.
- **Clipboard operations**: the default event handlers for the ObjectComponents messages handle cutting and pasting for you. If you add to your menus standard commands such as Insert Object and Paste Link, ObjectComponents will implement them for you.
- **Drag and drop operations**: the default event handlers for ObjectComponents messages help you here, too. If the user drops an OLE object on your container's window, ObjectComponents inserts it in your document. If the user double-clicks the embedded object, ObjectComponents activates it. If the user drags the object, ObjectComponents moves it.
- **Standard OLE 2 user interface**: OLE defines standard user interface features and asks OLE programmers to comply with them. Built into ObjectComponents are dialog boxes for commands like Insert Object, Paste Special, and Paste Link; a pop-up menu that appears whenever the user right-clicks an embedded object; and an item on the container's Edit menu that always shows the verbs (server commands) available for the active object. ObjectComponents even arranges to modify the container's window if the server takes over the container's tool bar, status bar, and menus for in-place editing.
- **Compound files**: A new ObjectComponents class (*TOcStorage*) encapsulates file input and output to compound files. If you convert an ObjectWindows Doc/View application into an ObjectComponents container, the document writes itself to compound files automatically, creating storages and substorages within the file as needed.
- **EXE and DLL servers**: ObjectComponents lets you construct your OLE server as either a standalone executable program or as an in-process DLL server. DLL servers respond to clients more quickly because a DLL is not a separate process. OLE doesn't have to serialize calls or marshall parameters to communicate between a DLL server and its client.
- **Automation**: ObjectComponents permits C++ classes to be automated without structural changes to the classes themselves. It accomplishes this with nested classes that have direct access to the existing class members. These nested classes instantiate small command objects that reach the members through standard C++ mechanisms, avoiding the use of restrictive, non-portable stack manipulations. The command objects support hooks for undoing, recording, and filtering automation commands. A program can even send itself automation commands using standard C++ code.
- **Type libraries**: A type library describes for OLE all the classes, methods, properties, and data members available for controlling an automated application. Once you create an automation server, you can ask ObjectComponents to build and register the type library for you.
- **Registration**: OLE requires applications to register themselves with the system by providing a unique identifier string. For servers, this string and much other information besides must be recorded in the system's registration database as part of the program's installation process. With ObjectComponents, all you have to do is list all the information in one place using macros. Every time your server starts up, ObjectComponents confirms that the database accurately reflects the server's status. When necessary, ObjectComponents records or updates registration entries automatically.
- **Localization**: OLE servers need to speak the language of their client programs. If an automation server is marketed in several countries, it needs to recognize commands sent in each different language. A linking and embedding server registers strings that describe its objects to the user, and those too should be available in multiple languages in order to accommodate whatever language the user might request. If you provide translations for your strings, ObjectComponents uses the right strings at the right time. Add your translations to the program's resources and mark the original strings as localized when you register

them. At run time, ObjectComponents quickly and efficiently retrieves translations to match whatever language OLE requests. For more about localization, see "Registering localized entries" on page 373.

See Also
Automation
Compound file I/O classes
Creating a type library
Linking and Embedding
Localizing Symbol Names
Making a DLL Server
Overview of automation classes
Overview of linking and embedding classes
Registering the application
Using ObjectComponents
What does OLE look like?

Using ObjectComponents

See Also Support for OLE in Borland C++

To use ObjectComponents, you will want to survey the classes and messages in ObjectComponents, as well as new classes in ObjectWindows that help you take advantage of ObjectComponents. You will need to understand how ObjectComponents uses C++ exception handling, how to build an ObjectComponents application, and what files to distribute with your application.

The following topics discuss these and other topics:

- Overview of Classes and Messages
- Exception Handling in ObjectComponents
- Building an ObjectComponents Application
- Distributing Files with Your Application

See Also

OLE 2 features supported by ObjectComponents

What is ObjectComponents?

Overview of Classes and Messages

See Also Using ObjectComponents

The following topics introduce the ObjectComponents classes and messages you are likely to use most often. Subsequent chapters describe their use in more detail.

- Linking and Embedding Classes
- Connector Objects
- Automation Classes
- ObjectComponents Messages
- Messages and Windows
- New ObjectWindows OLE Classes

See Also

Automation

Linking and Embedding

Overview of Linking and Embedding Classes

See Also Overview of Classes and Messages

The following classes support linking and embedding, but if your program uses ObjectWindows you won't need to work directly with most of them.

Class	Description
TOcApp	Connects containers and servers to OLE. It implements COM interfaces for the application.
TOcDocument	Represents a compound document. It holds parts (embedded objects).
TOcModule	A mix-in class for deriving the application object in a linking and embedding program. It coordinates some basic housekeeping chores related to registration and memory management.
TOcPart	Represents an embedded or linked object in a document.
TOcRegistrar	Records application information in the system registration database and tells OLE when the application starts and stops. Also creates the TOcApp object and responds when OLE wants a server to make something.
TOcRemView	Represents a remote view for a server document. The server creates a remote view for every object it donates to a container. The remote view is drawn in the container's window.
TOcView	Responsible for displaying a part. A container needs a view for every part it embeds.

Although ObjectComponents includes classes for documents and views, it does not require applications to use the ObjectWindows Doc/View model. If you do use the Doc/View model, the new TOleDocument and TOleView classes make OLE programming even easier. ObjectWindows is not required, however. Any C++ program can use the ObjectComponents Framework.

See Also
Linking and embedding enums
Linking and embedding messages
Linking and embedding structs
New ObjectWindows OLE Classes
Overview of Automation Classes

Connector Objects

See Also Overview of Classes and Messages

A few of the ObjectComponents classes actually implement COM interfaces. (COM stands for Component Object Model. COM is the standard that defines what an OLE object is.) Most of the supported interfaces are not standard OLE interfaces; they are custom interfaces that communicate with OLE through the BOCOLE support library. But like any COM object they do implement IUnknown (by deriving from TUnknown).

The classes that define COM objects for linking and embedding are *TOcApp*, *TOcView*, *TOcRemView*, and *TOcPart*. These classes are special because they connect your application to OLE. They are called *connector objects*. An ObjectComponents application must create connector objects in order to interact with other OLE applications.

Because they are COM objects, connector objects have one peculiarity: their destructors are protected so you cannot call delete to destroy them. Readers familiar with OLE will recognize that the connector objects have internal reference counts that track the number of clients using them. Often you are not the only user of your own connectors. For example, when a server creates a *TOcRemView* to paint an object in a container's window, the container becomes a client of the same object. The server must not destroy the view object until the container is through with it, otherwise OLE could end up attempting to address functions that no longer exist in memory.

The Component Object Model decrees that an object must maintain an internal reference count. When an object provides anyone a pointer to one of its interfaces, the object also increments its own reference count. When the client finishes with the pointer, it calls Release and the object decrements its reference count. As long as the count is greater than zero, the object must not be destroyed. When the count reaches 0, the object destroys itself.

ObjectComponents shields you from the details of reference counting. You never have to increment or decrement a reference count. You cannot delete COM objects, however, because the delete command pays no attention to the reference count. Instead, call the connector's *ReleaseObject* method.

See Also
TOcApp
TOcPart
TOcRemView
TOcView

Overview of Automation Classes

See Also Overview of Classes and Messages

Following are some ObjectComponents classes used for automation.

Class	Description
TAutoBase	Simplifies clean-up chores when an automated object is destroyed. Make it the base class for your automated classes if you want that help.
TAutoProxy	the base class for an automation controller's proxy objects. Controllers create C++ proxy objects to represent the OLE objects they want to manipulate. The proxy objects become connected to OLE when they derive from TAutoProxy.
TOleAllocator	Initializes the OLE libraries and, optionally, passes OLE a custom memory allocator for managing any memory the system allocates on the program's behalf.
TRegistrar	Records application information in the system registration database and tells OLE when the application starts and stops.

There are more automation classes than the above table shows, but many of them are internal to the ObjectComponents implementation. Most of the work in automating an existing application is done with macros. Automating a class means writing two tables of macros, one in the class declaration and one in the class implementation. The macros describe the methods you choose to expose. Within the parent class they create nested classes, one for each command. ObjectComponents knows how to make a nested object execute the method it exposes, and the nested class calls members of the parent class directly.

The connector objects that ObjectComponents creates to implement COM interfaces for an automation program are considered internal. ObjectComponents makes them for you when they are needed.

See Also

Automation data types

Automation enumerated types and type definitions

Automation macros

Automation structs

New ObjectWindows OLE classes

Overview of linking and embedding classes

ObjectComponents Messages

See Also Overview of Classes and Messages

When ObjectComponents needs to tell an application about signals and events that come from OLE, it sends a message through the normal Windows message queues. The message it sends is WM_OCEVENT. The value in the message's wParam identifies a particular event. Only applications that support linking and embedding receive WM_OCEVENT messages. (They are sent by the application's TOcApp, TOcView, and TOcRemView objects. Automation applications that don't support linking and embedding have no need for any of these objects.)

Simple ObjectWindows applications don't need to process any of the events because the new OLE classes have default event handlers that make reasonable responses for you. To modify the default behavior, add event handlers to your ObjectWindows program. If you are programming without ObjectWindows, handle WM_OCEVENT in your window procedure.

The events are divided into two groups. Those that concern the application as a whole are listed in the following table. Those that call for a response from a particular document are addressed to the view window. They are listed in the next table.

Most of the events in the following tables are sent only to a server or to a container. A single application receives both kinds of messages if it chooses to support both container and server capabilities.

Application messages for TOcApp clients

wParam value	Recipient	Description
OC_APPDIALOGHELP	Container	Asks the container to show Help for one of the standard OLE dialog boxes where the user has just clicked the Help button.
OC_APPBORDERSPACEREQ	Container	Asks the container whether it can give the server border space in its frame window.
OC_APPBORDERSPACESET	Container	Asks the container to rearrange its client area windows to make room for server tools.
OC_APPFRAMERECT	Container	Requests client area coordinates for the inner rectangle of the program's main window.
OC_APPINSMENUS	Container	Asks the container to merge its menu into the server's.
OC_APPMENUS	Container	Asks the container to install the merged menu bar.
OC_APPPROCESSMSG	Container	Asks the container to process accelerators and other messages from the server's message queue.
OC_APPRESTOREUI	Container	Tells the container to restore its normal menu, window titles, and borders because in-place editing has ended.
OC_APPSHUTDOWN	Server	Tells the server when its last embedded object closes down. If the server has nothing else to do, it can terminate.
OC_APPSTATUSTEXT	Container	Passes text for the status bar from the server to the container.

A view is the image of an object as it appears onscreen. When an OLE server gives an object to a container, the object contains data. The server also provides a view of the data so OLE can draw the object onscreen. Sometimes the word view also refers to the window where the container draws a compound document with all its embedded parts. Each object has its own small view, and the container has a single larger view of the whole document with all its embedded objects.

View messages for TOcView and TOcRemView clients

wParam value	**Recipient**	**Description**
OC_VIEWATTACHWINDOW	Server	Asks server window to attach to its own frame window or container's window.
OC_VIEWBORDERSPACEREQ	Container	Asks whether server can have space for a tool bar within the view of an embedded object.
OC_VIEWBORDERSPACESET	Container	Asks container to rearrange its windows so the server can show its tool bar within an embedded object.
OC_VIEWBREAKLINK	Server	Asks server to break a link to the currently selected data.
OC_VIEWCLIPDATA	Server	Asks server to provide clipboard data in a particular format.
OC_VIEWCLOSE	Server	Asks server to close its document.
OC_VIEWDRAG	Server	Asks server to provide visual feedback as the user drags its embedded object.
OC_VIEWDROP	Container	Tells container an object has been dropped on its window and asks it to create a *TOcPart*.
OC_VIEWGETITEMNAME	Server	Asks the server for a moniker identifying the currently selected data.
OC_VIEWGETPALETTE	Server	Asks server for the color palette it uses to draw an object.
OC_VIEWGETSCALE	Container	Asks container to give scaling information.
OC_VIEWGETSITERECT	Container	Asks container for the site rectangle that a part occupies.
OC_VIEWINSMENUS	Server	Asks server to insert its menus in a composite menu bar.
OC_VIEWLOADPART	Server	Asks server to load an embedded object stored in the container's data file.
OC_VIEWOPENDOC	Server	Asks server to open a document with the specified path.
OC_VIEWPAINT	Server	Asks server to draw or redraw an object at a particular position in a given device context.
OC_VIEWPARTACTIVATE	Container	Indicates that an embedded part has become active.
OC_VIEWPARTINVALID	Container	Indicates that embedded objects needs to be redrawn.
OC_VIEWPARTSIZE	Server	Asks server the initial size of its view in pixels.
OC_VIEWSAVEPART	Server	Asks server to write the data for an object into the container's file.
OC_VIEWSCROLL	Container	Asks container to scroll its view window.
OC_VIEWSETSCALE	Server	Asks server to handle scaling.
OC_VIEWSETLINK	Server	Asks server to create a link to the currently

		selected data.
OC_VIEWSETSITERECT	Container	Asks container to set the site rectangle.
OC_VIEWSETTITLE	Container	Asks the container to add to its title bar the name of the server for the active object.
OC_VIEWSHOWTOOLS	Server	Asks server to display its tool bar in container's window.
OC_VIEWTITLE	Container	Asks container for the caption in its frame window.

See Also
Messages and Windows

Messages and Windows

See Also Overview of Classes and Messages

Because the view and part objects expect to send notification messages to a particular document, every ObjectComponents application is expected to create a new window for each open document. Document windows should not be frame windows; they should be client windows that exactly fill the client area of a parent frame window. In an SDI application, the parent is the application's main frame window. In an MDI application, the parent is an MDI child frame. ObjectWindows programs should use *TOleWindow* for client windows. Many ObjectWindows applications, including all those that use the Doc/View model, already possess client windows.

See Also
Creating a View Window
ObjectComponents Messages
Setting up the Client Window

New ObjectWindows OLE Classes

See Also Overview of Classes and Messages

Another set of new classes integrates ObjectWindows with ObjectComponents. Internally, the new ObjectWindows classes use the the ObjectComponents classes to connect with OLE for you. Depending on the complexity of your ObjectWindows application, you might not need to interact directly with ObjectComponents at all.

The following table briefly summarizes the most important new ObjectWindows classes.

New classes	**Base classes**	**Description**
TOleFrame	TDecoratedFrame	Provides OLE user interface support for the main window of an SDI application.
TOleMDIFrame	TMDIFrame and TOleFrame	Provides OLE user interface support for the main window of an MDI application.
TOleWindow	TWindow	Used as the client of a frame window, provides support for embedding objects in a compound document.
TStorageDocument	TDocument	Adds the ability to work with OLE's compound file structure. It is the natural class to choose for compound documents with embedded objects.
TOleDocument	TStorageDocument	Implements the Document half of an OLE-enabled Doc/View pair.
TOleView	TOleWindow, TView	Implements the View half of a Doc/View pair.
TOleFactory<>	TOleFactoryBase	Implements the function OLE calls when an application should create an object.
TOleDocViewFactory<>	TOleFactoryBase	Implements the function OLE calls when an application should create an object.
TOleAutoFactory<>	TOleFactoryBase	Implements the function OLE calls when an application should create an object.
TOleDocViewAutoFactory<>	TOleFactoryBase	Implements the function OLE calls when an application should create an object.
TAutoFactory<>	TOleFactoryBase	Implements the function OLE calls when an application should create an object.
TOcAutoFactory<>	TOleFactoryBase	Implements the function OLE calls when an application should create an object.

The ObjectWindows OLE classes create ObjectComponents objects for you as needed. For example, whenever a container or a server creates a compound document, it also creates a a *TOcView* (or *TOcRemView*) object to implement the interfaces that tie a document to OLE. *TOleView::CreateOcView* does that for you. Furthermore, when the new *TOcView* object sends event messages to the view window, *TOleView* processes them for you with handlers like *EvOcViewSavePart* and *EvOcViewInsMenus*. The default event handlers manage much of the OLE user interface for you.

See Also

Overview of automation classes

Overview of linking and embedding classes

Turning a Doc/View application into an OLE container

Turning a Doc/View application into an OLE server

Turning an ObjectWindows application into an OLE container

Turning an ObjectWindows application into an OLE server

Exception Handling in ObjectComponents

See Also Using ObjectComponents

ObjectWindows 2.5 modifies the hierarchy of exception classes. *TXBase* is the new base class for all exception classes. *TXOwl* derives from it, as do the new exception classes summarized in the following table.

Class	Purpose
TXAuto	Exceptions that occur during automation
TXObjComp	Exceptions that occur during ObjectComponents linking and embedding operations
TXOle	Exceptions that occur while processing OLE API commands
TXRegistry	Exceptions that occur while using the system registration database

Because the exception classes all derive from TXBase, a general-purpose catch statement often takes a TXBase& as a parameter. The catch statement in the following example receives any exception thrown by ObjectWindows or ObjectComponents:

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
  try {
    Registrar = new TOcRegistrar(AppReg, TOleFactory<TMyApp>(),
                                 TApplication::GetCmdLine());
    return Registrar->Run();
  }
  catch (TXBase& x) {
    ::MessageBox(0, x.why().c_str(), "Exception", MB_OK);
  }
  return -1;
}
```

See Also
TXAuto
TXBase (OWL.HLP)
TXObjComp
TXOle
TXOle and Error Codes
TXRegistry

TXOle and OLE Error Codes

See Also Exception Handling in ObjectComponents

Most of the OLE API functions pass back a return value of type HRESULT (or the nearly identical SCODE). The return value indicates whether the call was successful, and it can also encode other status information. When a public member function of an ObjectComponents class results in a call to an OLE interface and the interface call fails, then ObjectComponents turns the OLE return result into a C++ exception object of type TXOle. This allows you to handle OLE error codes via the standard C++ **try** and **catch** constructs.

The *TXOle* class defines a variable, *Stat*, which holds the return value passed back from from a failed OLE API call. Therefore, a **catch** statement taking a *TXOle&* as a parameter has access to the OLE error code. The following code shows an example of a routine where the error value is simply returned back to the caller. This is useful if the function is called from an application that cannot handle C++ exceptions.

```
HRESULT
TMyAppDescriptor::CheckTypeLib(TLangId lang, const char far* file)
{
  HRESULT stat = HR_NOERROR;

  // Create OCF classes and invoke OCF methods to perform operation
  try {
    TOleCreateList typeList(new TTypeLibrary(*this, lang), file);
    .
.
.
  }

  catch(TXOle& x) {                        // Catch OLE exception
    stat = ResultFromScode(x.Stat);     // Create HRESULT from SCODE
  }
  return stat;                             // Return OLE error code
}
```

The previous example uses the *ResultFromScode* macro to cast an SCODE to an HRESULT. The OLE headers define various other macros that allow you to break down, assemble, and convert the various components of the value returned from an OLE API call.For more information, search for the topic "Error Handling Functions and Macros" in OLE.HLP.

If ObjectComponents catches a *TXOle* exception internally, it displays a dialog box showing the OLE return code. OLE documents the codes only in the header files where they are defined. To make what information there is more accessible, the DOCS/OLE_ERR.TXT file extracts information from that header and presents the codes in numerical order.

See Also
TXOle

Building an ObjectComponents Application

Using ObjectComponents

All ObjectComponents applications require exception handling and RTTI. Do not set any compiler options that disable these features.

Linking and embedding applications must use the large memory model. Automation applications can use the medium model as well (and they run faster in medium model).

The integrated development environment (IDE) sets the appropriate compiler and linker options for you automatically when you select OCF in the TargetExpert.

To build any ObjectComponents program from the command line, create a short makefile that includes the OWLOCFMK.GEN found in the EXAMPLES subdirectory. If your application does not use ObjectWindows, include the OCFMAKE.GEN instead. Here, for example, is the makefile that builds the AutoCalc sample program:

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj autocalc.obj
HLP = MYPROGRAM
!include $(BCEXAMPLEDIR)\ocfmake.gen
```

EXERES and OBJRES hold the name of the file to build and the names of the object files to build it from. HLP is optional. Use it if your project includes an online Help file. Finally, your makefile should include OWLOCFMK.GEN or OCFMAKE.GEN.

Name your file MAKEFILE and type this at the command line prompt:

```
make MODEL=l
```

MAKE, using instructions in the included file, will build a new makefile tailored to your project. The new makefile is called WIN16L*xx*.MAK. The final two digits of the name tell whether the makefile uses diagnostic or debugging versions of the libraries. 01 Indicates a debugging version, 10 a diagnostic version, and 11 means both kinds of information are included. The same command also then runs the new makefile and builds the program. If you change the command to define MODEL as d, the new makefile is WIN16Dxx.MAK and it builds the program as a DLL.

For more information about how to use OCFMAKE.GEN and OWLOCFMK.GEN, read the instructions at the beginning of MAKEFILE.GEN, found in the same directory.

The following table shows the libraries an ObjectComponents program links with.libraries for building ObjectComponents programs.

The ObjectComponents library must be linked first, before the ObjectWindows library.

Medium model	Large model	DLL libraries	Description
OCFWM.LIB	OCFWL.LIB	OCFWI.LIB	ObjectComponents
OWLWM.LIB	OWLWL.LIB	OWLWI.LIB	ObjectWindows
BIDSM.LIB	BIDSL.LIB	BIDSI.LIB	Class libraries
OLE2W16.LIB	OLE2W16.LIB	OLE2W16.LIB	OLE system DLLs
IMPORT.LIB	IMPORT.LIB	IMPORT.LIB	Windows system DLLs
MATHWM.LIB	MATHWL.LIB		Math support
CWM.LIB	CWL.LIB	CRTLDLL.LIB	C run-time libraries

Distributing Files with Your Application

Using ObjectComponents

When you distribute your application, you need to distribute along with it some libraries that ObjectComponents requires. Your installation program should install the files for the user, being careful not to replace any more current versions the user might already have.

The following files are part of OLE 2 and should be distributed with any OLE application, whether it uses ObjectComponents or not.

compobj.dll	ole2conv.dll	ole2disp.dll
ole2.dll	ole2nls.dll	ole2prox.dll
storage.dll	Typelib.dll	stdole.tlb
ole2.reg		

All these files belong in the user's WINDOWS/SYSTEM directory. Microsoft requires that if you distribute any of the files, you must distribute all of them. Call RegEdit to merge OLE2.REG with the user's registration database. (The RegEdit registration editor comes with Windows.) Double-clicking OLE2.REG in the File Manager accomplishes the same thing.

Any program that uses ObjectComponents should also distribute BOCOLE.DLL.

In addition, if your program uses the DLL version of OWL, of the container class libraries, or of the run-time library, you should distribute those as well.

How ObjectComponents Works

See Also Support for OLE in Borland C++

The following topics are not essential for using ObjectComponents, only for understanding what goes on behind the scenes when you create ObjectComponents connector objects.

- How ObjectComponents Talks to OLE
- How ObjectComponents Talks to You
- Linking and Embedding Connections
- Automation Connections

The essential function of ObjectComponents is to connect you with OLE. ObjectComponents is an intermediate layer standing between OLE on one side and your C++ code on the other.

See Also

OLE 2 features supported by ObjectComponents

What is ObjectComponents?

How ObjectComponents Talks to OLE

See Also How ObjectComponents Works

Fundamentally, all OLE interaction of any sort requires the implementation of standard OLE interfaces, such as IUnknown and IDispatch, as defined by the Component Object Model (COM).

An Interface is just a set of related function prototypes forming a pure base class. Every OLE object that implements the same interface can choose to implement the prescribed functions in its own way. All that matters is that the interface functions always accept the same parameters and always produce the same results. This makes it possible for any OLE object to call any standard function in any other OLE object that supports the interface.

Every OLE object must implement the IUnknown interface. One of the three functions in the IUnknown interface is QueryInterface. This common function implemented on all OLE objects lets you ask whether the object supports other interfaces that you want to use, such as automation interfaces or data transfer interfaces. This makes it possible for any OLE object to determine at run time what any other OLE object can do.

OLE defines a large number of standard interfaces that are notoriously tedious to implement. Borland's BOCOLE support library defines an alternate set of custom COM interfaces that collectively provide an alternative interface to OLE programming, one conceived at a higher level of abstraction. Client objects of the support library must still implement IUnknown, as all COM objects must, but instead of other standard OLE interfaces such as IDataObject and IMoniker, they implement interfaces defined by BOCOLE. The support library acts as an agent translating commands received through its custom interfaces into standard OLE. All the custom interfaces commands are carried out for you using standard OLE interfaces.

The custom interfaces in the BOCOLE support library have names like *IBContainer* and *IBDocument*. You'll see them used if you look in the ObjectComponents source code. Because the support library is an internal tool and subject to change, its interfaces are not documented. The complete library source code, however, comes with Borland C++, so you can refer to it if you need to track the OLE interactions minutely. You can also modify and rebuild the support library, just as you can the ObjectWindows Library, if that suits your purposes.

See Also

Messages and Windows

ObjectComponents Messages

How ObjectComponents Talks to You

See Also How ObjectComponents Works

Some of the ObjectComponents classes define COM objects. These objects derive from *TUnknown*, an ObjectComponents base class that implements the *IUnknown* interface and handles details of aggregation (a way of combining several objects into a single functional unit). They also mix in other base classes that implement interfaces from the BOCOLE support library.

the ObjectComponents objects that implement COM interfaces are called connector objects, because they connect your application to OLE. *TOcPart*, for example, is the connector object that implements the interfaces a container must support for each OLE object (part) that is placed in its document. To embed an object in your document, you take information ObjectComponents gets from the Clipboard, a drop message, or the Insert Object dialog box, and you pass the information to the *TOcPart* constructor. Among other things, the constructor (indirectly) calls a BOCOLE function to create an embedded OLE object. *TOcPart* holds the pointer to that object, queries it for interfaces, and stores the coordinates of the site where the part should be drawn. When you want the part to do something, you call *TOcPart* methods such as *Activate* and *Save*.

See Also

Connector objects

How ObjectComponents talks to OLE

Messages and Windows

ObjectComponents Messages

TOcPart

TUnknown

Linking and Embedding Connections

See Also How ObjectComponents Works

A linking and embedding application always creates a *TOcApp* object (usually it is created for you). *TOcApp* is a connector object that implements interfaces every linking and embedding application needs. Another connector object that all linking and embedding applications create is the view object, either *TOcView* for a container or *TOcRemView* for a server. You create one view object for each document you open. A view object is associated with the window where the document is drawn. The only other connector object used for linking and embedding is *TOcPart*, which containers create for each object deposited in their documents.

Of course communication through a connector object is not just one way. When you call methods on a connector object, the object calls through to OLE, but sometimes OLE needs to call you. For example, if when user chooses Insert Object and asks for an object from a server, OLE must invoke the server and ask it to create an object. The connector objects cannot, of course, call your functions the same way you can call theirs because they don't know anything about your code. When a connector object needs to communicate a request or a notification from OLE to you, it sends WM_OCEVENT message to one of your windows. *TOcApp* sends its messages to your frame window. The view and part objects send messages to the client window where you draw your document.

Communication from you to OLE happens through function calls to connector objects. Communication from OLE to you happens through messages from connector objects to your windows.

The initial wiring between you and ObjectComponents is established the first time the registrar object calls your factory callback function. The *TOcApp* object is bound to a window in *TOleFrame*::*SetupWindow*, or in the WM_CREATE handler of your main window.

See Also
Connector Objects
Factory template classes (OWL.HLP)
Messages and Windows
ObjectComponents Messages
Overview of Linking And Embedding Classes
TOcApp
TOcPart
TOcRemView
TOcView
TOleFrame::SetupWindow (OWL.HLP)

Automation Connections

See Also How ObjectComponents Works

Applications that support automation but not linking and embedding use a different set of objects. The central function of the automation layer in OLE is to pass arguments from the controller to the server, an operation with no user interface. The COM interfaces for automation are buried deeper in the implementation of ObjectComponents than the linking and embedding interfaces.

To support automation, ObjectComponents must identify exposed commands and arguments, attach type information to them, transfer values to and from the stack of VARIANT unions that OLE uses to pass values, and invoke your C++ functions when a controller sends a command. Once you set up the tables that describe what you want to expose, there is little in the automation process to customize or override. You never directly create or manipulate the connector objects for automation; ObjectComponents does it for you.

Advanced users who enjoy reading source code might like to know that *TServedObject* is the class that implements *IDispatch* and *ITypeInfo*, that *TTypeLibrary* implements *ITypeLib*, and that *TAutoIterator* implements *IEnumVARIANT*. Of these, only *TAutoIterator* is exposed as a public part of ObjectComponents. The others are considered internal implementation.

To automate a class, ObjectComponents asks you to build two descriptive tables from macros. A declaration table goes with the class declaration and declares which members are accessible to OLE. A definition table goes with the class implementation and assigns public names for controllers to use when invoking your functions. The automation macros also create nested classes within the automated parent, one for each exposed function or data member. The nested classes have an Invoke method that calls your function. Because the class is nested, it has direct access to your class through normal C++ mechanisms.

TServedObject is the connector that receives *IDispatch* commands from OLE and translates them into the appropriate *Invoke* calls. *TServedObject* finds the information it needs to do this in an object of type *TAutoClass*, which holds the symbol information from the automation tables. *TServedObject* receives dispatch IDs, looks them up in *TAutoClass*, uses the information it finds to extract arguments from the stack of VARIANT unions passed by OLE. Finally it calls *Invoke* on the appropriate nested command object.

See Also

Automation

Connector objects

Overview of Automation Classes

TAutoIterator

ObjectComponents Programming Tools

Support for OLE in Borland C++

The most powerful tool in Borland C++ to help you with ObjectComponents programming is AppExpert. AppExpert generates a complete basic application according to your specification. It fully supports both linking and embedding and automation. Use it to create containers, servers, and automation servers. ClassExpert helps you modify the generated code to make it do what you need.

The TargetExpert in the integrated development environment (IDE) also supports ObjectComponents. Click the option for OCF and it automatically sets the right build options.

For information about   tools that help you create ObjectComponents applications, see Utility Programs.

Utility Programs

ObjectComponents Programming Tools

Borland C++ 4.5 comes with some new utility programs that simplify common OLE programming chores. Some of them solve problems that other chapters explain in more detail.

AutoGen

Generates proxy classes for an automation controller. Scans the type library of an automated application and writes the source code for classes a controller uses to send commands automation commands.

DllRun

Launches a DLL server in executable mode. Any DLL server written with ObjectComponents can also run as a standalone application if you invoke it with DllRun. Running in executable mode sometimes makes it easier to debug the DLL. It also makes it possible to distribute a single program that your users can run either as an in-process server or as an independent application.

GuidGen

Generates globally unique identifiers for use in registering applications. Every server must have an absolutely unique ID. Containers need them in order to be link sources.

MacroGen

Generates automation macros for exposing functions with any number of arguments. The ObjectComponents headers declare versions of the macros for functions with up to four arguments. MacroGen saves you from having to revise the macros by hand to accommodate more arguments.

Register

Registers or unregisters any ObjectComponents EXE or DLL. Usually the applications register themselves if necessary when they run, or in response to command-line switches. Developers, however, sometimes need to register and unregister different versions of an application over and over. Register is especially useful for DLLs because you can't pass command-line switches to a DLL.

WinRun

A background program that makes it possible to launch Windows programs from the command line prompt in a DOS box. WinRun makes it possible to run GUI programs (such as Register) from a make file.

The source code for all the utilities but WINRUN is in the OCTOOLS directory.

You might find it helpful to install these tools in the integrated development environment (IDE). For more information, open the EXAMPLES\IDE\IDEHOOK\ IDEHOOKIDE file and read the instructions in OLETOOLS.CPP.

Where Do I Look for Information?

See Also   Support for OLE in Borland C++

You can find information about programming with ObjectComponents in the online Help, in the printed manuals, and in the directories of sample programs.

Throughout the documentation, OLE refers to OLE 2.0 unless version 1 is indicated explicitly.

- Books
- Online Help
- Example Programs

See Also
Where should you start?

Books

Where do I Look for Information?

The chapters that follow describe how to build programs that perform all these functions.

Chapter Title	Topic
Support for OLE in Borland C++	Overview of ObjectComponents
Creating an OLE container	How to build an application that receives OLE objects in its documents
Creating an OLE server	How to build an application that creates OLE objects for containers to use
Automating an application	How to build an application that other programs can control
Creating an automation controller	How to build an application that controls other applications

For complete reference material covering all the new OLE-related classes and macros in ObjectComponents andObjectWindows, see the *ObjectWindows Reference Guide*.

The ObjectComponents material in this book and in the *ObjectWindows Reference Guide* is also in the online Help for Borland C++.

The *ObjectWindows Tutorial* develops a sample application from scratch. The later steps use add OLE container, server, and automation capabilities.

Online Help

Where do I Look for Information?

Borland C++ includes three online Help files covering the OLE API. For the most part, ObjectComponents makes knowledge of OLE interfaces unnecessary, but if you want to understand more about how ObjectComponents works, or if your application requires advanced programming at the OLE interface level, then you might find these files useful.

Help file	Topic
OCF.HLP	ObjectComponents chapters from the *ObjectWindows Programmer's Guide* and the *ObjectWindows Reference Guide*
OWL.HLP	Reference material for new OLE-enabled classes in ObjectWindows
OLE.HLP	OLE 2 overviews and reference

Example Programs

Where do I Look for Information?

One of the best ways to learn about programming is to study working code. AppExpert is a good place to start. Use it to generate the code for servers, containers, automation servers, and DLL servers. In addition, Borland C++ comes with a variety of sample programs that show off ObjectComponents. Some of them are described in this list.

EXAMPLES/OCF

ObjectComponents without ObjectWindows

- **AutoCalc**: an automation server; draws a calculator onscreen and lets a controller click the buttons
- **CallCalc**: an automation controller to manipulate the calculator in AutoCalc
- **CppOcf**: Three-step linking and embedding tutorial that starts with a simple C++ program, turns it into a container, and then into a server
- **Localize**: Pulls translated strings from XLAT resources to reflect language settings
- **RegTest**: Registers, validates the registration, and unregisters an ObjectComponents application

EXAMPLES/OWL/TUTORIAL

ObjectWindows tutorial examples

- **OwlOcf**: Three-step linking and embedding tutorial that starts with a simple ObjectComponents program, turns it into a container, and then into a server.
- **Step14 - Step17**: the final steps of the tutorial application described in *ObjectWindows Tutorial*; shows how to be a linking and embedding container or server, how to be an automation server or controller, and how to support both automation and linking and embedding at the same time.

EXAMPLES/OWL/OCF

ObjectComponents with ObjectWindows

- **MdiOle**: A multidocument interface application with container capabilities
- **SdiOle**: A single document interface application with container capabilities
- **Tic Tac Toe**: A linking and embedding server

SOURCE/OCTOOLS

Source code for programming utilities

- **AutoGen**: Scans a type library and generates proxy classes for an automation controller
- **DllRun**: Runs a DLL server in executable mode
- **GuidGen**: Generates globally unique identifiers (GUIDs)
- **Register**: Registers a DLL server

Glossary of OLE Terms

A
B
C
D
E
G
I
J
L
O
P
R
S
T
V

The definitions in this list explain common terms in OLE programming. Read it for an introduction to important programming topics, or refer to it for clarification as you read other ObjectComponents chapters.

The definitions of advanced concepts assume you already know something about OLE and its standard interfaces. For more information about OLE, refer to the three OLE online Help files.

A

Activate

Aggregation

Automated Object

Automated Application

Automation

Automation Controller

Automation Server

B

BOCOLE Support Library

C

COM Object

Compound Document

Compound File

Connector Object

Container

D

DLL Server

Document

E

Activate

Glossary of OLE Terms

The user *activates* a linked or embedded object by double-clicking it. Activating an object causes the server to execute the object's primary verb. For document-style objects, the primary verb is generally initiates an editing session, either in-place or open. For other objects, such as movies and sounds, the primary verb is usually Play. Activating is not the same as selecting; see the entry for *Select*.

Aggregation

Glossary of OLE Terms

A way of combining several OLE objects to make them function as a single bigger object. Objects are aggregated at run time. You can aggregate with objects that you did not design. An object aggregates to delegate commands or to inherit and override the functionality of other objects.

Aggregation is an advanced programming technique. In order for aggregated objects to act as a unit, all the aggregated objects must delegate any *QueryInterface* call they receive to the primary object, usually called the outer object. The outer object begins an aggregation by passing its own *IUnknown* pointer. The second object remembers the outer *IUnknown* pointer and routes all requests for an interface to the outer object. If the outer object does not support a requested interface, it forwards the request to the first in what might be a chain of aggregated objects. A client can reach all the interfaces supported by any of the auxiliary objects through the *IUnknown* of the outer object.

Automated Object

See Also Glossary of OLE Terms

An OLE object that publishes commands other applications can send it. An automation server creates automated objects. The automated object can be the application itself or something that the application creates.

See Also
automation server

Automated Application

See Also Glossary of OLE Terms

An OLE object that publishes commands other applications can send it. An automation server creates automated objects. The automated object can be the application itself or something that the application creates.

See Also
automation server

Automation

Glossary of OLE Terms

The ability of an application to define a set of commands for other applications to invoke.

Automation Controller

Glossary of OLE Terms

An application that invokes commands to control automated objects or applications. A controller is sometimes called an *automation client*.

Automation Server

Glossary of OLE Terms

An application that exposes some of its own internal function calls as a set of commands that other programs can invoke. An *automation object* is what the server creates for other programs to control.

BOCOLE Support Library

Glossary of OLE Terms

A DLL of OLE implementation utility interfaces that ObjectComponents calls internally. The support library implements a number of custom OLE interfaces designed by Borland. The BOCOLE.DLL file should be distributed with any ObjectComponents program. Its custom interfaces are considered internal and so are not documented. The source code for the BOCOLE support library, however, is included with Borland C++.

COM Object

Glossary of OLE Terms

An object whose architecture conforms to the Component Object Model, a Microsoft specification that forms the basis of the OLE system. Briefly stated, the characteristics of COM objects are

- They communicate through predefined interfaces.
- They all support the *IUnknown* interface, and *IUnknown* includes the *QueryInterface* method for getting other optional interfaces.
- They keep a reference count of their clients and delete themselves if the count reaches zero.

Only COM objects can communicate with OLE. Some of the classes in ObjectComponents are COM objects (see *Connector object*). ObjectComponents shields you from the details of interface methods, interface pointers, and reference counters. It connects you to OLE using familiar C++ and Windows programming models such as inheritance and messages.

Compound Document

Glossary of OLE Terms

A document that contains OLE objects brought in from other applications. A compound document might contain pieces of information from a spreadsheet, a database, and a word processor, all in one document that the user loads or saves with a single command. The objects from other applications are either linked or embedded in the container's document.

Compound File

Glossary of OLE Terms

A single disk file that the operating system divides into independent compartments called *storages*. In effect, each storage has its own file I/O pointer so you can read, write, rewrite, and erase data in any one storage without needing to maintain offsets to other storages in the same file. Compound files are useful for storing compound documents because you can create a new storage for each linked or embedded object. OLE extends the file system by implementing interfaces to support compound files.

Connector Object

Glossary of OLE Terms

An ObjectComponents class that communicates with OLE for you. Connector objects connect parts of your application to OLE. *TOcApp*, for example, performs OLE functions for the application. *TOcView* performs OLE functions for one view of a document. *TOcPart* performs OLE functions for a linked or embedded object. The connector objects are partners that work together with corresponding parts of your application. You call their methods and they send you messages. Connectors are Component Object Model objects and implement COM interfaces. (Not all ObjectComponents classes are connectors.)

Container

Glossary of OLE Terms

An application that permits OLE to embed or link objects from other applications into its own documents. Containers are also called *clients* of the servers that give them objects.

DLL Server

A server whose code is in a dynamic-link library rather than an executable file. The advantage of a DLL server is speed. When OLE invokes an .EXE server to support an embedded object, it has to create a a separate process and marshall data to pass it between the two applications. A DLL, on the other hand, is part of the same system task as its client, so OLE calls from a container to a DLL server run much more quickly. See "Making a DLL server" on page 374.

Document

Glossary of OLE Terms

This word has two different meanings for programmers. First, a document is a set of data that an application loads in response to File|Open. A document can be a spreadsheet, or a bitmap, or a letter, or any other set of data that an application treats as a whole.

Sometimes it is useful to distinguish between the data in a document and the appearance of the data onscreen. A spreadsheet, for example, might be able to display a single set of data as either a table of numbers or a chart. One document can be displayed different ways. In such cases, *document* refers only to the data, and each possible representation of the document is called a *view*.

ObjectWindows programmers are familiar with an application architecture called the *Doc/View model* that separates the code for managing document data from the code for displaying the data. ObjectComponents also has a document class and view classes, but they are not part of the ObjectWindows Doc/View model. The document class keeps track of the objects embedded in a document and the view classes draw the objects onscreen.

Embedded Object

Glossary of OLE Terms

Data from a server application deposited by OLE in a container's document. OLE lets the user paste, drag, or insert objects into a container. If during these actions the user chooses to create an *embedded* object, then all the data in the object is copied to the container's document. When the user loads or saves the document, the data for the embedded object is written to the file along with the container's own native data.

Contrast embedded objects with *linked* objects, where the the data for the OLE object is stored in another application and the container receives only a reference to the object's file.

EXE Server

Glossary of OLE Terms

A server application compiled and linked into an executable file. A server can also be built as a library; see DLL *server*.

GUID

Glossary of OLE Terms

Globally unique identifier, a 16-byte value. OLE uses GUIDs to identify applications, the objects they produce, and the interfaces that objects implement. For linking and embedding, OLE needs GUIDs to match embedded objects to their servers even after the user transfers a compound document from system to system. If two servers had the same ID, OLE might accidentally invoke the wrong one. Each server and object type must have an absolutely unique ID. Tools such as GUIDGEN create the ID for you. For more information, see the *clsid* entry in the *ObjectWindows Reference Guide*.

IDispatch Interface

Glossary of OLE Terms

The OLE interface that all automated objects implement. With the four methods of the *IDispatch* interface, you can ask any automated object for information about its automated commands, look up the identifiers for particular commands, or invoke any command. For more information, see the OLEAUTO.HLP Help file.

In-place Editing

Glossary of OLE Terms

Editing an OLE object in the container's window. During in-place editing, the container lets the server display its own menus and tool bars in the container's window. The purpose of in-place editing is to let the user edit any object in a document without leaving the document's window. Contrast Open editing.

In-process Server

Glossary of OLE Terms

A server whose code is in a dynamic-link library rather than an executable file. The advantage of a DLL server is speed. When OLE invokes an .EXE server to support an embedded object, it has to create a a separate process and marshall data to pass it between the two applications. A DLL, on the other hand, is part of the same system task as its client, so OLE calls from a container to a DLL server run much more quickly.

Interface

See Also Glossary of OLE Terms

A set of function prototypes, usually declared as an abstract base class. OLE objects are able to communicate with each other because they implement standard interfaces, sets of functions that the system defines. The system defines only what functions an interface contains; it does not implement the functions. Each object implements the functions for itself. The interfaces are defined in the OLE system headers such as compobj.h and ole2.h. The OLE system communicates with applications and objects by calling the functions it assumes each one has implemented. For more about the OLE interface model, see the entry for *Component Object Model* (COM). For examples of standard OLE interfaces, see *IDispatch* and *IUnknown*.

Besides the standard interfaces, an object can define and implement its own custom interfaces. Of course the system can't call functions from custom interfaces because it doesn't know they exist, but other applications that know about the custom interface can use it. Internally, ObjectComponents works through a set of Borland custom interfaces. See BOCOLE support library.

ObjectComponents shields you from having to understand or implement particular interfaces. Advanced users who want to manipulate interfaces directly or mix in their own custom interfaces are free to do so.

See Also

BOCOLE support library

IUnknown Interface

Glossary of OLE Terms

The root interface that all OLE objects and interfaces must implement. With the three methods of the *IUnknown* interface, you can ask any object for a pointer to another interface it might also support, and you can adjust the object's reference count. For more information, see the Help file, OLE.HLP.

Linked Object

Glossary of OLE Terms

An object that appears in a container document but whose data really resides in another file. When dragging or pasting an object into a container, the user can choose to create a *link* to the object instead of embedding it. The container does not receive or store the linked object's data in its own document. Instead, it receives only a string identifying the location of the actual data, which can be in a file.

Several containers can link to the same object. In that case, all the containers receive the same string pointing to the same object. If the data in the original object changes, then the changes are reflected automatically in all the documents that link to it. If the user *embeds* one object in several containers, then each container has its own copy of the object's data and changes in one copy do not affect the other copies.

Link Source

Glossary of OLE Terms

The document that a link refers to, the source for the data in a linked object. Usually the link source is a server document, but it is not uncommon for containers to export link source data so that other applications can link to objects embedded or linked in the container's document. For information on becoming a link source.

.

Localization

Glossary of OLE Terms

Adapting an application to display strings in the user's language, whatever that might be. OLE servers need to speak the language of their client programs. If an automation server is marketed in several countries, it needs to recognize commands sent in each different language. A linking and embedding server registers strings that describe its objects to the user, and those too should be available in multiple languages in order to accommodate whatever language the user might request. ObjectComponents lets you place translations for all your strings in your resource file as XLAT resources. ObjectComponents chooses the right string at the right time.

ObjectComponents Framework

Glossary of OLE Terms

A set of C++ classes from Borland International that encapsulate linking and embedding functions as well as automation functions. Internally the ObjectComponents classes implement standard and custom OLE interfaces. With ObjectComponents you write for OLE using familiar programming models such as inheritance and window messages instead of implementing COM interfaces.

ObjectWindows Library

Glossary of OLE Terms

A set of C++ classes from Borland International that encapsulate standard Windows programming functions such as managing windows and dialog boxes. The current version of ObjectWindows introduces some new classes, such as *TOleWindow* and *TOleView*, that use ObjectComponents classes to acquire OLE capabilities. The new classes make it very easy to add OLE support to existing ObjectWindows applications.

OLE

Glossary of OLE Terms

Object linking and embedding, an extension to the Windows system. (In newer versions of Windows, OLE is an integral part of the system , not an extension.) the new commands that OLE implements and the interfaces it defines add many new features to the system, including linking and embedding, automation, and compound file I/O.

OLE Interface

See Also Glossary of OLE Terms

A set of function prototypes, usually declared as an abstract base class. OLE objects are able to communicate with each other because they implement standard interfaces, sets of functions that the system defines. The system defines only what functions an interface contains; it does not implement the functions. Each object implements the functions for itself. The interfaces are defined in the OLE system headers such as compobj.h and ole2.h. The OLE system communicates with applications and objects by calling the functions it assumes each one has implemented. For more about the OLE interface model, see the entry for Component Object Model (COM). For examples of standard OLE interfaces, see IDispatch and IUnknown.

Besides the standard interfaces, an object can define and implement its own custom interfaces. Of course the system can't call functions from custom interfaces because it doesn't know they exist, but other applications that know about the custom interface can use it. Internally, ObjectComponents works through a set of Borland custom interfaces. See BOCOLE support library.

ObjectComponents shields you from having to understand or implement particular interfaces. Advanced users who want to manipulate interfaces directly or mix in their own custom interfaces are free to do so.

See Also
BOCOLE support library

Open Editing

Glossary of OLE Terms

Editing an OLE object in the server's own window. Open editing happens when the user executes the Open verb. During open editing, the server's window opens up in front of the container's window. When the user finishes editing the object, the server window disappears and the modifications become visible back in the container window. Contrast in-place editing.

Part

See Also Glossary of OLE Terms

An object linked or embedded in a compound document. An ObjectComponents container creates an object of class *TOcPart* to represent each object linked or embedded in its document.

Part is the container's word for an object created by a server. In the server's code, the same object is created as a normal server document. ObjectComponents presents the document to OLE as an OLE object. The container, when it receives the OLE object, creates a *TOcPart*. When the part needs to be painted, the part object communicates through OLE with the server's view object.

See Also
TOcPart

Reference Counting

Glossary of OLE Terms

Away of remembering how many clients an object has. Every section of code that requires the object to exist calls the object's *AddRef* method to increment the reference count. When the client code is done, it calls the object's *Release* method to decrement the reference count. If a *Release* call causes the count to reach 0, then the object is allowed to destroy itself.

Every OLE object has *AddRef* and *Release* methods because they are part of the *IUnknown* interface. Knowing who is a client and when to call *AddRef* or *Release* is sometimes complicated. ObjectComponents manages reference counting for you. Only advanced users will find any need to call *AddRef* or *Release* directly.

Registrar Object

See Also Glossary of OLE Terms

An object of type *TRegistrar* or *TOcRegistrar*. Every ObjectComponents application needs a registrar object. The registrar processes the application's command line, sets running mode flags, verifies the application's entries in the system registration database, and calls the application's factory function to launch the application.

Registration: giving information about the application to the system. OLE programs perform two different kinds of registration. When an application is first installed, ObjectComponents writes information from the application's registration tables into the system registration database. This information is static and needs to be recorded only once. The registrar object performs this task.

Subsequently whenever the user launches the application, ObjectComponents tells OLE that the application is running and it registers a factory for each type of document the application can produce. When the application ends, ObjectComponents unregisters the factories. The *TOcApp* or *TRegistrar* object performs this task.

See Also
Registration Database

Registration Database

Glossary of OLE Terms

A structured repository of information about applications installed on a particular computer. In 16-bit Windows, the database is kept in the REG.DAT file. In 32-bit Windows, the database is called the system registry and resides in private system files. Applications record their information during installation. The information includes identifiers for the application and its documents, descriptions of the application and its documents, the path to the application file, the default extension of the application's document files, and other details that help the OLE system associate servers with their objects.

Registration Table

Glossary of OLE Terms

A table built with registration macros and containing information about an application or about the types of documents an application creates. The macros create a structure of type TRegList. The registrar object reads the registration structure and copies any necessary information to the system registration database.

Remote View

Glossary of OLE Terms

The view of its own object a server draws in a container's window. When an ObjectComponents server is launched to manage an object linked or embedded in a container's document, the server creates a *TOcRemView* object and a *TOcDocument* object. The view object draws in the container's window. The document object loads and saves the object's data.

Select

Glossary of OLE Terms

The user *selects* an object by clicking it once. The selected object does not become active and cannot be edited. Conventionally a container indicates that an object is selected by drawing a rectangle with grapples around the object. (*Grapples* are small handles for moving the rectangle.) the container might permit the user to select several objects at once to move or delete as a group, but usually only one object per child window can be active at a time.

Server

Glossary of OLE Terms

An application that creates objects for other applications to use. In this documentation, *server* usually refers to either a linking and embedding server or an automation server. A linking and embedding server creates data objects that containers can paste, drop, or insert into their own compound documents. An automation server creates objects that other applications can manipulate by sending commands for the object to execute. (A single application can choose to create both kinds of objects. It is even possible to link and embed automated objects.)

System Registration Database

Glossary of OLE Terms

A structured repository of information about applications installed on a particular computer. In 16-bit Windows, the database is kept in the REG.DAT file. In 32-bit Windows, the database is called the *system registry* and resides in private system files. Applications record their information during installation. The information includes identifiers for the application and its documents, descriptions of the application and its documents, the path to the application file, the default extension of the application's document files, and other details that help the OLE system associate servers with their objects.

Type Library

Glossary of OLE Terms

A file describing the commands an automation controller supports. Creating a type library is the standard way for an automation server to publish the programming interface it implements. The type library tells what objects the server creates and describes the objects' properties and methods. Type information is read by compilers and interpreters that process automation commands. Some applications also allow the user to browse the type information.

Any ObjectComponents automation server generates a type library if you invoke it with the TypeLib command line switch. Type libraries conventionally use the .TLB or .OLB extension. An automation server registers the location of its type library during installation.

Verb

Glossary of OLE Terms

A command that a linking and embedding server can execute with its objects. The server tells the container what verbs it supports and the container displays the verb strings on its own Edit menu. To execute a verb, the user selects an object and then chooses a verb from the menu. The container updates the verb menu each time the user selects a new object.

The server can support any verbs it chooses. Most servers support the Edit and Open verbs for in-place or open editing. Depending on the kind of data it owns, a server might choose to offer other verbs such as Play and Rewind.

View

Glossary of OLE Terms

The graphical representation of data. The term is used to distinguish between the way the data is painted and the data itself, usually called the *document*. A single word processor document, for example, might have three different views: a page layout view, a draft view without fancy fonts, and a print preview view.

In ObjectComponents, containers create views to draw their compound documents. Servers also create views to draw the objects they create. Both create a *TOcDocument* object to manage the data and a view object, either *TOcView* or *TOcRemView*, to draw the document.

In ObjectWindows, *Doc/View* refers to a particular application architecture supported by the framework that also treats data and its representation in separate classes.

Creating an OLE Server

See Also

An OLE server is an application that creates and manages data objects for other programs. Existing programs can be turned into linking and embedding servers.

The following topics discuss adapting three kinds of programs:

- Turning a Doc/View Application Into an OLE Server
- Turning an Objectwindows Application Into an OLE Server
- Turning a C++ Application Into an OLE Server

In addition, the following related topics round out this discussion of creating an OLE server:

- Understanding Registration
- Making a DLL Server

The ObjectComponents Framework classes support servers as well as containers, and new ObjectWindows classes make this support easily available. The easiest kind of program to convert is one that uses ObjectWindows and the Doc/View model, but ObjectComponents simplifies the task of writing a server application even without the ObjectWindows framework.

OLE applications can also be automation servers.

See Also

Automating an Application

Turning a Doc/View Application Into an OLE Server

Turning a Doc/View Application Into an OLE Server

See Also Creating an OLE Server

Turning a Doc/View application into an OLE server requires only a few modifications, and many of them are the same as the changes required to create a container. If you have already modified your application, then much of your server work is already done.

The following topics discuss the changes you will need to make to turn a Doc/View application into an OLE server:

1. Connect Objects to OLE
2. Register a Linking and Embedding Server
3. Draw, Load, and Save Objects
4. Build the Server

That's all you need to do. After performing these steps, your OLE server supports all the following features:

- Source for linking
- Source for drag and drop
- Registration
- Source for embedding
- In-place editing
- Compound document storage

To modify the default behavior ObjectComponents provides for common OLE options, you can additionally override the default handlers for messages that ObjectComponents sends.

See Also
Turning a Doc/View Application Into an OLE Server

Connecting Objects to OLE

Turning a Doc/View Application Into an OLE Server

Your application, window, document, and view objects need to make use of new OLE-enabled classes. The constructor for the application object expects to receive an application dictionary object, so create that first.

The following topics discuss the steps needed to connect to OLE:

- Creating an Application Dictionary
- Deriving the Application Object from TOcModule
- Inheriting from OLE Classes

Creating an Application Dictionary

See Also   Connecting Objects to OLE

An application dictionary tracks information for the currently active process. It is particularly useful for DLLs. When several processes use a DLL concurrently, the DLL must maintain multiple copies of the global, static, and dynamic variables that represent its current state in each process. For example, the DLL version of ObjectWindows maintains a dictionary that allows it to retrieve the *TApplication* corresponding to the currently active client process. If you convert an executable server to a DLL server, it must also maintain a dictionary of the *TApplication* objects representing each of its container clients.

The DEFINE_APP_DICTIONARY macro provides a simple and unified way to create the dictionary object for any type of application, whether it is a container, a server, a DLL, or an EXE. Insert this statement with your other static variables:

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

For any application linked to the static version of the DLL library, the macro simply creates a reference to the application dictionary in ObjectWindows. For DLL servers using the DLL version of ObjectWindows, however, it creates an instance of the *TAppDictionary* class.

Note: Name your dictionary object *AppDictionary* to take advantage of the factory templates such as *TOleDocViewFactory* (as explained later in "Creating a registrar object").

See Also
DEFINE_APP_DICTIONARY macro (OWL.HLP)
Factory Template Classes (OWL.HLP)
TApplication (OWL.HLP)
Turning a Doc/View Application Into an OLE Server

Deriving the Application Object from TOcModule

See Also Connecting Objects to OLE

The application object of an ObjectComponents program needs to derive from *TOcModule* as well as *TApplication*. *TOcModule* coordinates some basic housekeeping chores related to registration and memory management. It also connects your application to OLE. More specifically, *TOcModule* manages the connector object that implements COM interfaces on behalf of an application.

If the declaration of your application object looks like this:

```
class TMyApp : public TApplication {
  public:
    TMyApp() : TApplication(){};
   .
   .
   .
};
```

Then change it to look like this:

```
class TMyApp : public TApplication, public TOcModule {
  public:
    TMyApp(): TApplication(::AppReg["description"], ::Module,
&::AppDictionary){};
   .
   .
   .
};
```

The constructor for the revised *TMyApp* class takes three parameters.

- A string naming the application
  AppReg is the application's registration table." The expression *::AppReg["description"*] extracts a string that was registered to describe the application.
- A pointer to the application module
  Module is a global variable of type *TModule** defined by ObjectWindows
- the address of the application dictionary
  AppDictionary is the application dictionary object.

See Also
TApplication (OWL.HLP)
TModule (OWL.HLP)
TOcModule (OWL.HLP)
Registering a Linking and Embedding Server
Turning a Doc/View Application Into an OLE Server

Inheriting From OLE Classes

See Also Connecting Objects to OLE

A server makes the same changes to its OLE classes that a container makes. ObjectWindows wraps a great deal of power in its new window, document, and view classes. To give an ObjectWindows program OLE capabilities, change its derived classes to inherit from OLE classes.

Here are some examples:

```
// old declarations (without OLE)
class TMyDocument: public TDocument    { /* declarations */ };
class TMyView: public TView            { /* declarations */ };
class TMyFrame: public TFrameWindow    { /* declarations */ );

// new declarations (with OLE)
class TMyDocument: public TOleDocument { /* declarations */ };
class TMyView: public TOleView         { /* declarations */ };
class TMyFrame: public TOleFrame       { /* declarations */ );
```

When you change to OLE classes, be sure that those methods in your classes which refer to their direct base classes now use the OLE class names.

```
void TMyView::Paint(TDC& dc, BOOL erase, TRect& rect)
{
  TOleView::Paint(dc, erase, rect);
  // paint the view here
}
```

It is generally safer to allow the OLE classes to handle Windows events and Doc/View notifications. This is particularly true for the *Paint* method and mouse message handlers in classes derived from *TOleView*. *TOleView::Paint* knows how to paint the objects embedded in your document. (Servers are often containers as well, and a server's object might have other objects embedded in it.) Similarly, the mouse handlers of *TOleView* let the user select, move, resize, and activate an OLE object embedded or linked in your document.

See Also
Turning a Doc/View Application Into an OLE Server

Registering a Linking and Embedding Server

See Also

To register your application with OLE, first create registration tables describing the application and the kinds of documents it creates. The tables create structures that you pass to the registrar object when you create it. Call the registrar's *Run* method to start the application.

The following topics discuss the tasks involved with linking and embedding:

- Building Registration Tables
- Creating a Registrar Object
- Processing the Command Line

See Also

Turning a Doc/View Application Into an OLE Server

Understanding Registration

Building Registration Tables

See Also Registering a Linking and Embedding Server

databaseServers implement OLE objects that any container can use. Different servers implement different types of objects. Every type of object a server creates must have a globally unique identifier (GUID) and a unique string identifier. Every server must record this information, along with other descriptive information, in the registration database of the system where it runs. OLE reads the registry to determine which objects are available, what their capabilities are, and how to invoke the application that creates objects of each type.

ObjectComponents simplifies the task of registration. You call macros to build a table of keys with associated values. ObjectComponents receives the table and automatically performs all registration tasks.

Servers and containers use the same macros for registration, but servers must provide more information than containers. Here are the commands to build the registration tables for a typical server. This example comes from the STEP15.CPP and STEP15DV.CPP file in the EXAMPLES\OWL\TUTORIAL directory of your compiler installation.

```
REGISTRATION_FORMAT_BUFFER(100)          // allow space for expanding macros


BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,      "{5E4BD320-8ABC-101B-A23B-CE4E85D07ED2}")
  REGDATA(description,"Drawing Pad Server")
END_REGISTRATION


BEGIN_REGISTRATION(DocReg)
  REGDATA(progid,     "DrawPad.Document.15")
  REGDATA(description,"Drawing Pad (Step15--Server)")
  REGDATA(menuname,   "Drawing Pad 15")
  REGDATA(extension,  "p15")
  REGDATA(docfilter,  "*.p15")
  REGDOCFLAGS(dtAutoOpen | dtAutoDelete | dtUpdateDir | dtCreatePrompt |
dtRegisterExt)
  REGDATA(insertable, "")
  REGDATA(verb0,      "&Edit")
  REGDATA(verb1,      "&Open")
  REGFORMAT(0, ocrEmbedSource,  ocrContent,  ocrIStorage,  ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent,  ocrMfPict|ocrStaticMed, ocrGet)
  REGFORMAT(2, ocrBitmap,       ocrContent,  ocrGDI|ocrStaticMed, ocrGet)
  REGFORMAT(3, ocrDib,          ocrContent,  ocrHGlobal|ocrStaticMed,
ocrGet)
  REGFORMAT(4, ocrLinkSource,   ocrContent,  ocrIStream, ocrGet)
END_REGISTRATION
```

The macros in the example build two structures. The first structure is named *AppReg* and the second is *DocReg*. ObjectComponents uses lowercase strings such as *progid* and *clsid* to name the standard keys to which you assign values. The values you assign are strings, and they are sensitive to case. The order of keys within the registration table doesn't matter. For information on the the full set of registration macros, see the ObjectWindows Help (OWL.HLP).

The set of keys you place in a structure depends on what OLE capabilities you intend to support and whether the structure holds application information or document information. The macros in the example show the minimum amount of information a server with one type of document should provide.

A server registers program and class ID strings (*progid* and *clsid*) for itself and for every type of document it creates. The IDs must be absolutely unique so that OLE can distinguish one application from another. The *description* strings appear on the Insert Object dialog box where the user sees a list of objects available in the system.

The following table describes all the registration keys that can be used by a server that supports linking and embedding. It shows which are optional and which required as well as which belong in the application registration table and which in the document registration table

Key	in AppReg?	in DocReg?	Description
appname	Yes	Optional	Short name for the application
clsid	Yes	Optional	Globally unique identifier (GUID); generated automatically for the DocReg structure
description	Yes	Yes	Descriptive string (up to 40 characters)
progid	No	No	Identifier for program or object type (unique string)
			Note: in DocReg for link or embed source
menuname	No	Yes	Name of server object for container menu
extension	No	Optional	Document file extension associated with server
docfilter	No	Yes	Wildcard file filter for File Open dialog box
			Note: not in DocReg if *dtHidden*
docflags	No	Yes	Options for running the File Open dialog box
debugger	No	Optional	Command line for running debugger
debugprogid	No	Optional	Name of debugging version (unique string)
debugdesc	No	No	Description of debugging version
			Note: in DocReg if using debugprogid
insertable	No	No	Indication that object can be embedded. If omitted, the document is only a link source
			Note: in DocReg for embedding
verbn	No	Yes	An action the server can execute with the object
formatn	No	Yes	A clipboard format the server can produce
aspectall	No	Optional	Options for displaying object in any aspect
aspectcontent	No	Optional	Options for displaying the object's content aspect
aspectdocprint	No	Optional	Options for displaying the object's docprint aspect
aspecticon	No	Optional	Options for displaying the object's icon aspect
aspectthumbnail	No	Optional	Options for displaying the object's thumbnail aspect
cmdline	No	Optional	Arguments to place on server's command line
path	No	Optional	Path to server file (defaults to current module path)
permid	No	Optional	Name string without version information
permname	No	Optional	Descriptive string without version information
usage	Optional	Optional	Support for concurrent clients
language	Optional	No	Language for registered strings (defaults to system's user language setting)
version	Optional	No	Major and minor version numbers (defaults to "1.0")database

The previous table assumes that the server's documents support linking or embedding. For documents that support neither, the server needs to register only *docflags* and *docfilter*.

If the server is also a container or an automation server, then you should also consult the container table or the automation table. Register all the keys that are required in any of the tables that apply to your application.

For more information about individual registration keys, the values they hold, and the macros used to register them, see Registration Keys.

The values assigned to keys can be translated to accommodate system language settings.

Place your registration structures in the source code files where you construct document templates and implement your *TApplication*-derived class. A server always creates only one application registration table (called *AppReg* in the example). The server might create several document registration tables, however, if it creates several different kinds of documents (for example, text objects and chart objects). Each registration table needs a unique *progid* value.

After creating registration tables, you must pass them to the appropriate object constructors. The *AppReg* structure is passed to the *TOcRegistrar* constructor. Document registration tables are passed to the document template constructor.

```
DEFINE_DOC_TEMPLATE_CLASS(TMyOleDocument, TMyOleView, MyTemplate);
MyTemplate myTpl(::DocReg);
```

DatabaseSome of the information in the document registration table is used only by the document template. The document filter and document flags have to do with documents, not with OLE. Previous versions of OWL passed the same information to the document template as a series of separate parameters. The old method is still supported for backward compatibility, but new programs, whether they use OLE or not, should use the registration macros to supply document template parameters.

Some of the registration macros expand the values passed to them. The REGISTRATION_FORMAT_BUFFER macro reserves memory needed temporarily for the expansion. To determine how much buffer space you need, allow 10 bytes for each REGFORMAT item plus the size of any string parameters you pass to the macros REGSTATUS, REGVERBOPT, REGICON, or REGFORMAT. For more information, see Registration macros in Object Windows Help (OWL.HLP).

See Also
Localizing Symbol Names
Registering Localized Entries
Registration Keys
REGISTRATION_FORMAT_BUFFER Macro (OWL.HLP)
Registration Macros (OWL.HLP)
Turning a Doc/View Application Into an OLE Server
Understanding Registration Macros

Creating a Registrar Object

See Also Registering a Linking and Embedding Server

Every ObjectComponents application needs a registrar object to manage all of its registration tasks. In a linking and embedding application, the registrar is an object of type *TOcRegistrar*. At the top of your source code file, declare a global variable holding a pointer to the registrar.

```
static TPointer<TOcRegistrar> Registrar;
```

The *TPointer* template ensures that the *TOcRegistrar* instance is deleted when the program ends.

Note: Name this variable *Registrar* to take advantage of the factory callback template used in the registrar's constructor.

The next step is to modify your *OwlMain* function to allocate a new *TOcRegistrar* object and initialize the global pointer *Registrar*. The *TOcRegistrar* constructor expects three parameters: the application's registration structure, the component's factory callback and the command-line string that invoked that application. The registration structure is created with the registration macros.

The factory callback is created with a class template. For a linking and embedding ObjectWindows application that uses Doc/View, the template is called *TOleDocViewFactory*. The third parameter, the command-line string, can be obtained from the *GetCmdLine* method of *TApplication*. The code in the factory template assumes you have defined an application dictionary called *AppDictionary* and a *TOcRegistrar** called *Registrar*.

```
int OwlMain(int, char*[])
{
  // Create Registrar object
  ::Registrar = new TOcRegistrar(::AppReg, TOleDocViewFactory<TMyApp>(),
         TApplication::GetCmdLine());
.
.
.
}
```

After initializing the *Registrar* pointer, your OLE container application must invoke the Run method of the registrar instead of *TApplication::Run*. *TRegistrar::Run* calls the factory callback procedure (the one the second parameter points to) and causes the application to create itself. The application enters its message loop, which is actually in the factory callback. The following code shows a sample *OwlMain* before and after adding a registrar object. Boldface type highlights changes.

Before:

```
// Non-OLE OwlMain
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  return TMyApp().Run();
}
```

After:

```
// New declaration of OwlMain
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  ::Registrar = new TOcRegistrar(::AppReg,
        TOleDocViewFactory<TMyApp>(),
        TApplication::GetCmdLine());
  return ::Registrar->Run();
}
```

The last parameter of the *TOcRegistrar* constructor is the command line string that invoked the application.

See Also
Factory Template Classes (OWL.HLP)
TOcRegistrar
TRegistrar
Turning a Doc/View Application Into an OLE Server

Processing the Command Line

See Also Registering a Linking and Embedding Server

When OLE invokes a server, it places an -Embedding switch on the command line to tell the application why it has been invoked. The presence of the switch indicates that the user did not launch the server directly. Usually a server responds by keeping its main window hidden. The user interacts with the server through the container. If the -Embedding switch is not present, the user has invoked the server as a standalone application and the server shows itself in the normal way.

When you construct a *TRegistrar* object, it parses the command line for you and searches for any OLE-related switches. It removes the switches as it processes them, so if you examine your command line after creating *TRegistrar* you will never see them.

If you want to know what switches were found, call *IsOptionSet*. For example, this line tests for the presence of a registration switch on the command line:

```
if (Registrar->IsOptionSet(amAnyRegOption))
      return 0;
```

This is a common test in *OwlMain*. If the a command line switch such as RegServer was set, the application simply quits. By the time the registrar object is constructed, any registration action requested on the command line have already been performed.

The following table lists all the OLE-related command-line switches

Switch	**What the server should do**	**OLE places switch?**
-RegServer	Register all its information and quit	No
-UnregServer	Remove all its entries from the system and quit	No
NoRegValidate	Run without confirming entries in the system database	No
-Automation	Register itself as single-use (one client only). Always accompanied by -Embedding	Yes
-Embedding	Consider remaining hidden because it is running for a client, not for itself	Yes
-Language	Set the language for registration and type libraries	No
-TypeLib	Create and register a type library	No
-Debug	Enter a debugging session	Yes

OLE places some of the switches on the program's command line. Anyone can set other flags to make ObjectComponents perform specific tasks. An install program, for example, might invoke the application it installs and pass it the -RegServer switch to make the server register itself. Switches can begin with either a hyphen () or a slash (/).

Only a few of the switches call for any action from you. If a server or an automation object sees the -Embedding or -Automation switch, it might decide to keep its main window hidden. Usually ObjectComponents makes that decision for you. You can use the Debug switch as a signal to turn trace messages on and off, but responding to Debug is always optional. (OLE uses Debug switch only if you register the *debugprogid* key.)

ObjectComponents handles all the other switches for you. If the user calls a program with the -UnregServer switch, ObjectComponents examines its registration tables and erases all its entries from the registration database. If ObjectComponents finds a series of switches on the command line, it processes them all. This example makes ObjectComponents generate a type libary in the default language and then again in Belgian French.

```
myapp -TypeLib -Language=80C -TypeLib
```

the number passed to -Language must be hexadecimal digits. The Win32 API defines 80C as the locale ID for the Belgian dialect of the French language. For this command line to have the desired effect, of course, *myapp* must supply Belgian French strings in its XLAT resources.

The RegServer flag optionally accepts a file name.

```
myapp -RegServer = MYAPP.REG
```

This causes ObjectComponents to create a registration data file in MYAPP.REG. The new file contains all the application's registration data. If you distribute MYAPP.REG with your program, users can merge the file directly into their own registration database (using RegEdit). Without a file name, RegServer writes all data directly to the system's registration database.

Note: Only EXE servers have true command lines. OLE can't pass command line switches to a DLL . ObjectComponents simulates passing a command line to a DLL server so that you can use the same code either way. The registrar object always sets the right running mode flags.

See Also

Creating a Type Library

debugprogid Registration Key

Localizing Symbol Names

Making a DLL Server

Drawing, Loading, and Saving Objects

See Also Turning a Doc/View Application Into an OLE Server

A server must coordinate with its client to process its objects when they need to be painted or saved.

The following topics discuss drawing, loading, and saving objects:

- Telling Clients When an Object Changes
- Loading and Saving the Server's Documents

See Also
Turning a Doc/View Application Into an OLE Server

Telling Clients When an Object Changes

See Also Drawing, Loading, and Saving Objects

Whenever the server makes any changes that alter the appearance of an object, the server must tell OLE. OLE keeps a metafile representation with every linked or embedded object so that even when the server is not active OLE can still draw the object for the container. If the object changes, OLE must update the metafile. The server notifies OLE of the change by calling *TOleView::InvalidatePart*. OLE, in turn, asks the server to paint the revised object into a new metafile. ObjectComponents handles this request by passing the metafile device context to the server's *Paint* procedure. You don't need to write extra code for updating the metafile.

A good place to call *InvalidatePart* is in the handlers for the messages that ObjectWindows sends to a view when its data changes:

```
bool TDrawView::VnRevert(bool /*clear*/) {
  Invalidate();                // force full repaint
  InvalidatePart(invView);     // tell container about the change
  return true;
}
```

invView is an enumeration value, defined by ObjectComponents, indicating that the view is invalid and needs repainting.

Other view notification messages that signal the need for an update include EV_VN_APPEND, EV_VN_MODIFY, and EV_VN_DELETE.

See Also

TOleWindow::InvalidatePart (OWL.HLP)

Turning a Doc/View Application Into an OLE Server

Loading and Saving the Server's Documents

See Also   Drawing, Loading, and Saving Objects

When a server gives objects to containers, the containers assume the burden of storing the objects in files and reading them back when necessary. If your server can also run independently and load and save its own documents, it too should make use of the compound file capabilities built into *TOleDocument*.

In its Open method, a server calls *TOleDocument::Open*. In its *Commit* method, a server should call *TOleDocument::Commit* and *TOleDocument::CommitTransactedStorage*.

```
// document class declaration derived from TOleDocument
class _DOCVIEWCLASS TMyDocument : public TOleDocument {
  // declarations
}

// document class implementation
bool TMyDocument::Commit(bool force) {
  TOleDocument::Commit(force);   // save linked and embedded objects
  .
  .
  .                              // code to save other document data
  TOleDocument::CommitTransactedStorage();   // write to file if transacted
mode
}

bool TDrawDocument::Open(int, const char far* path) {
  TOleDocument::Open();    // load linked or embedded objects
  .
  .
  .                        // code to load other document data
}
```

Note: By default, *TOleDocument* opens compound files in transacted mode. Transacted mode saves changes in temporary storages until you call *CommitTransactedStorage*.

See Also

Loading And Saving Compound Documents

Turning a Doc/View Application Into an OLE Server

Building the Server

Turning a Doc/View Application Into an OLE Server

To build the server application, include the OLE headers and link with the OLE libraries.

The following topics discuss steps required for building the server:

- Including OLE headers
- Compiling and Linking

Including OLE Headers

See Also   Building the Server

The headers for a server are the same as the headers for a container. A server that uses the Doc/View model and an MDI frame window needs the following headers:

```
#include <owl/oledoc.h>     // replaces DOCVIEW.H
#include <owl/oleview.h>    // replaces DOCVIEW.H
#include <owl/olemdifr.h>   // replaces MDI.H
```

An SDI application includes OLEFRAME.H instead of OLEMDIFR.H.

See Also

Building an ObjectComponents Application

Turning a Doc/View Application Into an OLE Server

Compiling and Linking

See Also Building the Server

Linking and embedding servers that use ObjectComponents and ObjectWindows require the large memory model. Link them with the OLE and ObjectComponents libraries.

The integrated development environment (IDE) chooses the right build options when you ask for OLE support. To build any ObjectComponents program from the command line, create a short makefile that includes the OWLOCFMK.GEN file found in the EXAMPLES subdirectory.

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj myprogram.obj
!include $(BCEXAMPLEDIR)\ocfmake.gen
```

EXERES and OBJEXE hold the name of the file to build and the names of the object files to build it from. The last line includes the OWLOCFMK.GEN file. Name your file MAKEFILE and type this at the command-line prompt:

```
make MODEL=l
```

MAKE, using instructions in OCFMAKE.GEN, will build a new makefile tailored to your project. The new makefile is called WIN16L*xx*.MAK.

See Also
Turning a Doc/View Application Into an OLE Server

Turning an ObjectWindows Application Into an OLE Server

Creating an OLE Server

Turning a non-Doc/View ObjectWindows container into an OLE server requires a few modifications.

The following topics discuss those modifications:

1. Register the Server
2. Set Up the Client Window
3. Modify the Application Class
4. Build the Server

Code excerpts used in the above topics are from the OWLOCF2.CPP sample in the EXAMPLES/OWL/TUTORIAL/OLE directory. OWLOCF2 converts the OWLOCF1 sample from a container to a server.

Registering the server

See Also Turning an Objectwindows Application Into an OLE Server

To register the server you describe it in registration tablesone table for the application and one for each type of document it creates. The document tables are put in a linked list, and the registrar object processes the information in all the tables. The registrar also needs an application dictionary object.

The following was created for Help purposes.The following topics discuss how to register the server.

- Creating an Application Dictionary
- Creating Registration Tables
- Creating the Document List
- Creating the Registrar Object

See Also

Understanding Registration

Creating an Application Dictionary

See Also Registering the Server

An application dictionary tracks information for the currently active process. It is particularly useful for DLLs. When several processes use a DLL concurrently, the DLL must maintain multiple copies of the global, static, and dynamic variables that represent its current state in each process. For example, the DLL version of ObjectWindows maintains a dictionary that allows it to retrieve the *TApplication* corresponding to the currently active client process. If you convert an executable server to a DLL server, it must also maintain a dictionary of the *TApplication* objects representing each of its container clients.

The DEFINE_APP_DICTIONARY macro provides a simple and unified way to create the dictionary object for any type of application, whether it is a container, a server, a DLL, or an EXE. Insert this statement with your other static variables:

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

For any application linked to the static version of the DLL library, the macro simply creates a reference to the application dictionary in ObjectWindows. For DLL servers using the DLL version of ObjectWindows, however, it creates an instance of the *TAppDictionary* class.

It is important to name your dictionary object *AppDictionary* to take advantage of the factory templates such as *TOleFactory*.

See Also

DEFINE_APP_DICTIONARY macro (OWL.HLP)

Factory Template Classes (OWL.HLP)

Creating Registration Tables

See Also		Registering the Server

Servers implement OLE objects that any container can link or embed in their own documents. Different servers implement different types of objects. Every type of object a server can create must have a 16-byte globally unique identifier (GUID) and a unique string identifier. Every server must record this information, along with other descriptive information, in the registration database of the system where it runs. OLE reads the registry to determine what objects are available, what their capabilities are, and how to invoke the application that creates objects of each type.

A server provides registration information to ObjectComponents using macros to build registration tables: one table describing the application itself and one for each type of OLE object the server creates. Here is the application registration table from OWLOCF2.

```
REGISTRATION_FORMAT_BUFFER(100)


// application registration table
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,      "{B6B58B70-B9C3-101B-B3FF-86C8A0834EDE}")
  REGDATA(description,"Scribble Pad Server")
END_REGISTRATION
```

The registration macros build a structure of items. Each item contains a key, such as *clsid* or *description*, and a value assigned to the key. The order in which the keys appear does not matter. In the example, *AppReg* is the name of the structure that holds the information in this table.

Servers that create several types of objects must build a document registration table for each type. (What the server creates as a document is presented through OLE as an object.) If a spreadsheet application, for example, creates spreadsheet files and graph files, and if both kinds of documents can be linked or embedded, then the application registers two document types and creates two document registration tables.

The OWLOCF2 sample program creates one type of object, a scribbling pad, so it requires one document registration table (shown here) in addition to the application registration table.

```
// document registration table
BEGIN_REGISTRATION(DocReg)
  REGDATA(progid,     "Scribble.Document.3")
  REGDATA(description,"Scribble Pad Document")
  REGDATA(debugger,   "tdw")
  REGDATA(debugprogid,"Scribble.Document.3.D")
  REGDATA(debugdesc,  "Scribble Pad Document (debug)")
  REGDATA(menuname,   "Scribble")
  REGDATA(insertable, "")
  REGDATA(extension,  DocExt)
  REGDATA(docfilter,  "*."DocExt)
  REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
  REGDATA(verb0,      "&Edit")
  REGDATA(verb1,      "&Open")
  REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,   ocrGet)
END_REGISTRATION
```

The *progid* key is an identifier for this document type. The string must be unique so that OLE can distinguish one object from another. The *insertable* key indicates that this type of document should be listed in the Insert Object dialog box. The *description*, *menuname*, and *verb* keys are all visible to the user during OLE operations. The *description* appears in the Insert Object dialog box where the user sees a list of objects available in the system. The *menuname* is used in the container's Edit menu when composing the string that pops up the verb menu, which is where the *verb* strings appear. The remaining registration items are used when the application opens a file or uses the Clipboard.

For more information about particular register keys, see the *ObjectWindows Reference Guide*.

Place your registration structures in the source code file where you implement your *TApplication*-derived class. If you cut and paste registration tables from other programs, be sure to modify at least the *progid* and *clsid* because these must identify your application uniquely. (Use the GUIDGEN.EXE utility to generate new 16-byte *clsid* identifiers.)

See Also

Building Registration Tables

Registration Macros (OWL.HLP)

Understanding Registration

Creating the Document List

Registering the Server

The registration tables hold information about your application and its documents, but they are static. They don't do anything with that information. To register the information with the system, an application must pass the structures to an object that know how to use them. That object is the registrar, which records any necessary information in the system registration database.

In a Doc/View application, the registrar examines the list of document templates to find each document registration structure. A non-Doc/View application doesn't have document templates, so it uses *TRegLink* instead to create a linked list of all its document registration tables.

```
static TRegLink* RegLinkHead;
TRegLink scribbleLink(::DocReg, RegLinkHead);
```

RegLinkHead points to the first node of the linked list. *ScribbleLink* is a node in the linked list. The *TRegLink* constructor follows *RegLinkHead* to the end of the list and appends the new node. Each node contains a pointer to a document registration structure. In OWLOCF2, the list contains only one node because the server creates only one type of document. The node points to *DocReg*.

OWLOCF2 declares *RegLinkHead* as a static variable because it is used in several parts of the code, as the following sections explain.

Creating the Registrar Object

See Also Registering the Server

The registrar object registers and runs the application. Its constructor receives the application registration structure and a pointer to the list of document registration structures. In a linking and embedding application, the registrar is an object of type *TOcRegistrar*. At the top of your source code file, declare a global variable holding a pointer to the registrar.

```
static TPointer<TOcRegistrar> Registrar;
```

The *TPointer* template ensures that the *TOcRegistrar* instance is deleted when the program ends.

Note: Name this variable *Registrar* to take advantage of the factory callback template used in the registrar's constructor.

Next, in *OwlMain* allocate a new *TOcRegistrar* object and initialize the global pointer *Registrar*. The *TOcRegistrar* constructor has three required parameters: the application's registration structure, the component's factory callback and the command-line string that invoked that application.

An optional fourth parameter points to the beginning of the document registration list. In a Doc/View application, this parameter defaults to the application's list of document templates. Applications that do not use Doc/View should pass a *TRegLink** pointing to the list of document registration structures.

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
  try {
    // construct a registrar object to register the application
    Registrar = new TOcRegistrar(::AppReg,      // application registration
structure
               TOleFactory<TScribbleApp>(),     // factory callback
               TApplication::GetCmdLine(),      // app's command line
               ::RegLinkHead);                  // pointer to doc
registration structures

  // did command line say to register only?
  if (Registrar->IsOptionSet(amAnyRegOption))
      return 0;
    return Registrar->Run();                    // enter message loop in
factory callback
  }
  catch (xmsg& x) {
    ::MessageBox(0, x.why().c_str(), "Scribble App Exception", MB_OK);
  }
  return -1;
}
```

TOleFactory is a template that creates a class with a factory callback function. For a linking and embedding ObjectWindows application that does not use Doc/View, the template is called *TOleFactory*. The code in the factory template assumes you have defined an application dictionary called *AppDictionary* and a *TOcRegistrar** called *Registrar*.

When the registrar is created, it compares the information in the registration tables to the application's entries in the system registration database and updates the database if necessary. The *Run* method causes the registrar to call the factory callback which, among other things, enters the application's message loop.

See Also
Factory Template Classes (OWL.HLP)
Processing the Command Line
TOcRegistrar
Turning a Doc/View Application Into an OLE Server

Setting Up the Client Window

See Also Turning an Objectwindows Application Into an OLE Server

ObjectComponents applications need to have a separate window for each document. The document window derives from *TOleFrame* and usually is made to fill the client area of a frame window. If your application does not already have a client window, you will need to add one.

The client window constructors are good places to place the two helper objects that will be required:

- Creating Helper Objects for a Document

See Also
Setting Up the Client Window

Creating Helper Objects for a Document

See Also Setting Up the Client Window

Each new document you open needs two helper objects from ObjectComponents: *TOcDocument* and *TOcView*. Because you create a client window for each document, the window's constructor is a good place to create the helpers. The *TOleWindow::CreateOcView* function creates both at once.

In OWLOCF2, the client window is *TScribbleWindow*. Here is the declaration for the class and its constructor:

```
class TScribbleWindow : public TOleWindow {
  public:
    TScribbleWindow(TWindow* parent, TOpenSaveDialog::TData& fileData);
    TScribbleWindow(TWindow* parent, TOpenSaveDialog::TData& fileData,
TRegLink* link);
```

The second constructor is new. It is useful when ObjectComponents passes you a pointer to the registration information you provided for one of your document types and asks you to create a document of that type. Here is the implementation of the new constructor:

```
TScribbleWindow::TScribbleWindow(TWindow* parent, TOpenSaveDialog::TData&
fileData,     TRegLink* link)
:
  TOleWindow(parent, 0),
  FileData(fileData)
{
   .
   .
   .
  // Create a TOcDocument object to hold the OLE parts that we create
  // and a TOcRemView to provide OLE services
  CreateOcView(link, true, 0);
}
```

The constructor receives a *TRegLink* pointer and passes it on to *CreateOcView*. The pointer points to the document registration information for the type of document being created. ObjectComponents passes the pointer to this constructor; you don't have to keep track of it yourself.

Passing true to *CreateOcView* causes the function to create a *TOcRemView* helper instead of a *TOcView*. The remote view object draws an OLE object within a container's window. When a server is launched to help a client with a linked or embedded object, it should create a remote view.

If your application supports more than one document type, you can choose to use a different *TOleWindow*-derived class for each one. You must then provide the additional constructor for each class. Alternatively, you can use a single *TOleWindow*-derived class that behaves differently depending on the *TRegList* pointer it receives.

See Also
TOcDocument
TOcRemView
TOcView
TOleWindow (OWL.HLP)

Modifying the application class

See Also Turning an Objectwindows Application Into an OLE Server

ObjectComponents requires that the class you derive from *TApplication* must also inherit from *TOcModule*. In addition, the application object needs to implement a *CreateOleObject* method with the following signature:

```
TUnknown* CreateOleObject(uint32 options, TRegList* link);
```

The purpose of the function is to create a server document for linking or embedding. The server must create a client window and return a pointer of type *TOcRemView**. Here is how OWLOCF2 declares this procedure:

```
class TScribbleApp : public TApplication, public TOcModule {
  public:
    TScribbleApp();
    TUnknown* CreateOleObject(uint32 options, TRegLink* link);
```

and here is how it implements the procedure:

```
TUnknown*
TScribbleApp::CreateOleObject(uint32 options, TRegLink* link)
{
  if (!link)  // factory creating an application only, no view required
    link = &scribbleLink;       // need to have a view for this app
  TOleFrame* olefr = TYPESAFE_DOWNCAST(GetMainWindow(), TOleFrame);
  CHECK(olefr);
  FileData.FileName[0] = 0;
  TScribbleWindow* client = new TScribbleWindow(
          olefr->GetRemViewBucket(), FileData, link);
  client->Create();
  return client->GetOcRemView();
}
```

ObjectWindows uses the *CreateOleObject* method to inform your application when OLE needs the server to create an object. The *TRegLink** parameter indicates which object to create.

- Understanding the Treglink Document List

See Also
Deriving the Application Object from TOcModule
TOcModule
TOcRemView

Understanding the TRegLink Document List

See Also		Modifying the Application Class

This topic   explains the relationship between the document registration structure, the document list, and the *CreateOleObject* method. A server builds a document registration table for each type of object that it can serve. (The variable that holds the document registration information is conventionally named *DocReg*.) the registration structure is then passed to a *TRegLink* constructor, which appends the the structure to a linked list so that all the document types can be registered.

OLE displays the *description* value for each document in the Insert Object dialog box whenever the user asks to insert an object. (OLE also displays the *description* strings for all the other available server document types.) When the user chooses to insert one of your objects into a container application, OLE launches your server and places the -Embedding switch on the command line. When the server loads, ObjectComponents calls your *CreateOleObject* method, passing the address of the registration link that was used to register the requested document type. The *TRegList* pointer lets you determine which type of object was chosen. This matters primarily for servers that register more than one document type.

The following code illustrates one possible implementation of the *CreateOleObject* method for an application that serves more than one type of object:

```
// Create a appropriate client window and return its TOcRemView pointer
TUnknown*
TServerApp::CreateOleObject(uint32 options, TRegList* link)
{
  if (link == &chartLink) {
    // Create TOleChartWindow
    // and return charWindow->GetOcRemView();
  }

  if (tpl == &worksheetLink) {
    // Create TOleWorksheetWindow
    // and return worksheetWindow->GetOcRemView();
  }

  return 0;
}
```

See Also
Processing the Command Line

Building the Server

See Also Turning an Objectwindows Application Into an OLE Server

To build the server application, include the OLE headers and link with the OLE libraries.

The following topics discuss those tasks:

- Including OLE Headers
- Compiling and Linking

See Also
Building an ObjectComponents Application

Including OLE Headers

Building the Server

ObjectWindows provides OLE-related classes, structures, macros, and symbols in various header files. The following list shows the headers needed for an OLE container using an SDI frame window.

```
#include <owl/oleframe.h>
#include <owl/olewindo.h>
#include <ocf/ocstorag.h>
```

An MDI application includes olemdifr.h instead of oleframe.h.

Compiling and Linking

See Also Building the Server

Linking and embedding servers that use ObjectComponents and ObjectWindows must be compiled with the large memory model. They must be linked with the OLE and ObjectComponents libraries.

See Also
Building an ObjectComponents Application

Turning a C++ Application Into an OLE Server

Creating an OLE Server

If you are writing a new program, consider using ObjectWindows to save yourself some work. The ObjectWindows Library contains built-in code that automatically performs some tasks common to all ObjectComponents programs. Programs that don't use ObjectWindows must undertake these chores for themselves.

The following list briefly describes what you need to do to turn a C++ application into an ObjectComponents server.If you have already turned your C++ application into a container, much of the server work is already done. The most important differences concern the registration tables and the factory callback function.

The following topics discuss each step in turning a C++ application into an OLE server.

1. Create a Memory Allocator
2. Register the Application
3. Create a View Window
4. Program the Main Window
5. Build the Server

The sections that follow illustrate each step using examples from the programs in the EXAMPLES/OCF/CPPOCF directory. The source files titled CPPOCF0 contain a Windows application that does not support OLE. CPPOCF1 turns the first program into an OLE container. CPPOCF2 adds server capabilities. The code samples for this discussion come from CPPOCF2. The CPPOCF2 server creates a simple timer display for containers to embed. The timer display increments every second.

Creating a Memory Allocator

See Also Turning a C++ Application Into an OLE Server

A server adds this line to the beginning of its *WinMain* procedure:

```
TOleAllocator allocator(0);      // use default memory allocator
```

The allocator's constructor initializes the OLE libraries and its destructor releases them when the object goes out of scope. Passing 0 to the constructor tells it to let OLE use its standard memory functions whenever allocating memory for this application.

See Also
TOleAllocator

Registering the Application

See Also Turning a C++ Application Into an OLE Server

CPPOCF2 supports basic server functions. It registers information about itself and its document type, and it creates on demand an object to embed in other applications.

The following topics discuss steps needed to register your application:

- Building Registration Tables
- Creating the Document List
- Creating the Registrar Object
- Writing the Factory Callback Function

See Also
Understanding Registration

Building Registration Tables

See Also   Registering the Application

A server uses the registration macros to build registration tables describing the application and the documents it creates. The first table describes the server itself:

```
REGISTRATION_FORMAT_BUFFER(100)
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,        "{BD5E4A81-A4EF-101B-B31B-0694B5E75735}")
  REGDATA(description,  "Sample C Server")
END_REGISTRATION
```

The registration macros build a structure of type *TRegList*. The structure is stored in a variable named *AppReg*. Each entry in the structure contains a key, such as clsid or *description*, and a value assigned to the key. Internally, ObjectComponents finds the values by searching for the keys. The order in which the keys appear does not matter.

Tthe server creates a second registration table to describe the type of document it produces. If a spreadsheet application, for example, creates spreadsheet files and graph files, it registers two document types. CPPOCF2 creates only one kind of document, a timer display. The registration structure for this document type is held in a variable named *DocReg*, as the following code shows.

```
BEGIN_REGISTRATION(DocReg)
  REGDATA(description,  "Sample C Server Document")
  REGDATA(progid,       APPSTRING".Document.1")
  REGDATA(menuname,     "CServer")
  REGDATA(insertable,   "")
  REGDATA(verb0,        "&Edit")
  REGDATA(verb1,        "&Open")
  REGDATA(extension,    "scd")
  REGDATA(docfilter,    "*.scd")
  REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
  REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,   ocrGet)
END_REGISTRATION
```

The *progid* key is an identifier for this document type. The *insertable* key indicates that this type of document should be listed in the Insert Object dialog box. The *description*, *menuname*, and *verb* keys are all visible to the user during OLE operations. The *description* appears in the Insert Object dialog box. The *menuname* is used in the container's Edit menu when composing the string that pops up the verb menu, which is where the *verb* strings appear.

The remaining registration items are used when the application opens a file or uses the clipboard. For descriptions of individual keys, see the *ObjectWindows Reference Guide*.

See Also

BuildingRegistrationTables

Registration keys

Registration Macros (OWL.HLP)

Creating the Document List

See Also Registering the Application

The registration tables hold information about your application and its documents, but they are static. They don't do anything with that information. To register the information in the system, an application must pass the structures to objects that know how to use them. That object is the registrar, which records any necessary information in the system registration database.

To accommodate servers with many document types, the registrar accepts a pointer to a linked list of all the application's document registration structures. Each node in the list is a *TRegLink* object. Each node contains a pointer to one document registration structure and another pointer to the next node.

```
TRegLink *RegLinkHead = 0;
TRegLink  regDoc(DocReg, RegLinkHead);
```

RegLinkHead points to the first node of the linked list. *RegDoc* is a node in the linked list. The *TRegLink* constructor follows *RegLinkHead* to the end of the list and appends the new node. Each node contains a pointer to a document registration structure. In CPPOCF2, the list contains only one node because the server creates only one type of document. The node points to *DocReg*.

CPPOCF2 declares *RegLinkHead* as a static variable because it is used in several parts of the code, as the following sections explain.

See Also
Understanding the TRegLink Document List

Creating the Registrar Object

See Also   Registering the Application

The registrar object records application information in the system registration database, processes any OLE switches on the application's command line, and notifies OLE that the server is running.
CPPOCF2 declares a static pointer for the registrar object:

```
TOcRegistrar* OcRegistrar = 0;
```

Create your registrar object as you initialize the application in *WinMain*. Instead of entering a message loop, call the registrar's *Run* method. When *Run* returns, the application is shutting down. Delete the registrar before you quit. The CPPOCF2 *WinMain* function shows all the steps.

```
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, char far* lpCmdLine,
int nCmdShow)
{
  HInstance = hInstance;
  try {
    TOleAllocator allocator(0);     // use default memory allocator

    string cmdLine(lpCmdLine);      // put app's command line in a C++
string

    // construct the registrar
    OcRegistrar = new TOcRegistrar(::AppReg,    // application registration
structure
          ComponentFactory,                     // factory callback function
          cmdLine,                              // application command line
          ::RegLinkHead);                       // document registration
structure list


   .
   .
   . // application initialization commands go here

    if (OcRegistrar->IsOptionSet(amEmbedding))
      nCmdShow = SW_HIDE;

   .
   .
   . // instance initialization commands go here

    OcRegistrar->Run();             // call factory asking app to create
itself and run
    delete OcRegistrar;
  }
  catch (xmsg& x) {
    MessageBox(GetFocus(), x.why().c_str(), "Exception caught", MB_OK);
  }
  return 0;
}
```

The *TOcRegistrar* constructor takes four parameters:

- *::AppReg*, is the application registration structure already built with the registration macros.
- *ComponentFactory* is a callback function. The callback is responsible for creating any of the application's OLE components, including the application itself, as required. The callback also contains the application's message loop.

- *cmdLine* is a *string* object holding the application's command line. The registrar searches the command line for OLE-related switches such as Automation or Embedding, and it sets internal running mode flags accordingly.
- *::RegLinkHead* points to the linked list of documentation registration structures.
- During its initialization, the server checks whether it was invoked by OLE or directly by the user. OLE launches the server when the user activates a linked or embedded object that the server created. OLE sets the -Embedding switch on the server's command line to indicate that the server is running only to support a client, not as a stand-alone application. When the registrar discovers the -Embedding switch on the command line, it sets an internal flag. The server tests for this flag by calling *IsOptionSet*. If OLE did launch the application, the server will draw only in the container's window and it does not need to display its own window. CPPOCF2 sets *nCmdShow* to SW_HIDE to prevent subsequent initialization code from displaying the main window.

nCmdShow is the parameter Windows passes to *WinMain* indicating the initial state of the application's main window. A well behaved Windows application passes the value to *ShowWindow* immediately after creating the main window.

The *TOcRegistrar::Run* function causes the registrar to call the application's factory callback. In this case, the callback executes the application's message loop and the application runs.

See Also
Creating the Document List
TOcRegistrar
Processing the Command Line

Writing the Factory Callback Function

See Also Registering the Application

The factory callback is a function you implement and pass to the registrar. When it is time for the application to run, or when a container tries to insert one of the server's objects, ObjectComponents invokes the callback function.

The factory callback decides what to do by reading the parameters it receives and examining the running mode flags the registrar has set. The callback is called a factory because it creates OLE component objects on request.

The requirement that every ObjectComponents application must supply a factory callback function unifies the process of creating objects. Normally the process varies depending on whether the application is a container or a server, whether it is automated, whether it is running as a DLL or an executable program, and whether the application was invoked by the user directly or by OLE. The factory callback makes it possible to revise and run the application in a variety of ways without rewriting any code. For more information about factory callbacks, look up "Factory Templates" in the *ObjectWindows Reference Guide*.

A set of factory templates such as *TOleFactory* and *TOleAutoFactory* make it easy to implement factories for ObjectWindows programs, but in a straight C++ program you have to write the factory yourself.

Factory callback procedures can have any name you like, but they must follow this prototype:

```
IUnknown* ComponentFactory(IUnknown* outer, uint32 options, uint32 id);
```

outer is used when aggregating OLE objects to make them function as a single unit. The factory's return value is also used for aggregation. Because containers don't aggregate, CPPOCF1 ignores *outer* and returns 0.

options contains the bit flags that indicate the application's running mode. The registrar object sets the flags when it processes the command line switches, before it calls the factory callback. The factory tests the flags to find out what it should do. The possible flags are defined by the *TOcAppMode* enumerated type, and they have names like *amRun* and *amShutdown*.

id is an identifier that tells the factory what kind of object to create.

The factory's parameters can direct the factory to perform one of three actions:

- Initialize the application. The first time it runs, the factory creates a *TOcModule* object. *TOcModule* connects the application to the OLE system by creating a *TOcApp* connector object. The factory also handles aggregation in this phase.
- Run the application. If the *amRun* flag is set, the factory enters the message loop. If the server is built as a DLL, then when OLE loads the server the registrar does not set the *amRun* flag and the server should not run its own message loop.
- Create an object. The *id* parameter tells the factory what kind of object to create. Because CPPOCF2 creates only one kind of object, it checks only whether *id* is greater than 0. In applications that register multiple document templates, *id* points to the template for the requested object.

The factory callback in CPPOCF2 refers to four global variables. One is *OcRegistrar*. Another is *OcApp*.

```
TOcRegistrar* OcRegistrar = 0;
TOcApp*       OcApp       = 0;
```

TOcApp is the connector object that implements OLE interfaces on behalf of the application. One of the factory's jobs is to create the connector object when the application starts and to destroy it when the application shuts down.

Here is the factory callback from CPPOCF2:

```
IUnknown*
ComponentFactory(IUnknown* outer, uint32 options, uint32 id)
{
  IUnknown* ifc = 0;
  // start the application or shut it down
  if (!OcApp) {
    if (options & amShutdown)  // no app to shutdown!
      return 0;
    OcRegistrar->CreateOcApp(options, OcApp);
  } else if (options & amShutdown) {
    DestroyWindow(HwndMain);
    return 0;
  }

  // aggregate if an outer pointer was passed
  if (id == 0)
    OcApp->SetOuter(outer);

  // enter message loop if the run flag is set
  if (options & amRun) {
    if ((options & amEmbedding) == 0) {
      HwndView = CreateViewWindow(HwndMain);
    }
    MSG msg;
    // Standard Windows message loop
    while (GetMessage(&msg, 0, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }

  // create a document if the id parameter is non-zero
  if (id) {
    OcDoc     = new TOcDocument(*OcApp);
    HwndView  = CreateViewWindow(HwndMain);
    OcRemView = new TOcRemView(*OcDoc, &DocReg);
    if (IsWindow(HwndView))
      OcRemView->SetupWindow(HwndView);
    ifc = OcRemView->SetOuter(outer);
  }
  return ifc;
}
```

The factory's *outer* parameter is 0 unless some other object is aggregating with the newly created object. Aggregated objects are components that act together as a single unit. Objects can form aggregations at run time; you do not need access to an object's source code to aggregate with it. ObjectComponents supports aggregation by passing *outer* to the application factory. If *outer* is non-zero, it points to the *IUnknown* interface of anther object that wants the newly created object to subordinate itself. To allow aggregation, the factory calls the *SetOuter* method on the object it is creating, either *TOcApp* or *TOcRemView*. *SetOuter* returns a pointer to the object's own *IUnknown* interface. The factory should return the same pointer, too.

Note: *TOcApp::SetOuter* is only called when an application automates itself. CPPOCF2 includes the call anyway in case the application later becomes an automation server.

See Also
TOcApp
TOcAppMode enum
TOcDocument
TOcRemView

Creating a View Window

ObjectComponents imposes one design requirement on servers: the server document must have its own window, separate from the application's main window. To keep the distinction clear, we'll call the main window the *frame* window, because it uses the WS_THICKFRAME style and has a visible border on the screen. The second window has no visible border. We'll call it the *view* window because that is where the application displays its data. The view window always exactly fills the frame window's client area, so from the user's point of view the frame window appears to be the only window. ObjectComponents needs the view window, though, because it expects to send some event messages to the application and some to the view.

In an SDI application like the CPPOCF2 sample program, the frame window controls the view window. When the frame window receivesa WM_SIZE message, it moves the view to keep it aligned with the frame's client area. When it receives WM_CLOSE, it destroys both itself and the view window.

In an MDI application, each child window creates its own view. The child window does what the SDI frame does: creates and manages a view for the document it displays.

The following topics discuss tasks needed to create the view window:

- Creating, Resizing, and Destroying the View Window
- Creating a New Server Document
- Handling Wm_Ocevent
- Handling Selected View Events
- Painting the Document

Creating, Resizing, and Destroying the View Window

Creating a View Window

Before creating the view window, the application must first register a class for the view window. CPPOCF2 registers both classes in *InitApplication*.

CPPOCF2 creates the view window in its factory because the factory is in charge of creating new documents on request. The code for the view window, as you'll see, connects the new document to OLE by creating some ObjectComponents helper objects. The factory calls this function to create the view window:

```
HWND CreateViewWindow(HWND hwndParent)
{
  HWND hwnd = CreateWindow(VIEWCLASSNAME, "",
    WS_CHILD | WS_CLIPCHILDREN | WS_CLIPSIBLINGS | WS_VISIBLE | WS_BORDER,
    10, 10, 300, 300,
    hwndParent, (HMENU)1, HInstance, 0);
  return hwnd;
}
```

CPPOCF2 resizes and destroys the view window when the frame window receives WM_SIZE and WM_CLOSE messages.

```
void
MainWnd_OnSize(HWND hwnd, UINT /*state*/, int /*cx*/, int /*cy*/)
{
  if (IsWindow(HwndView)) {
    TRect rect;
    GetClientRect(hwnd, &rect);
    MoveWindow(HwndView, rect.left, rect.top, rect.right, rect.bottom,
true);
  }
}

void
MainWnd_OnClose(HWND hwnd)
{
  if (IsWindow(HwndView))
    DestroyWindow(HwndView);
  DestroyWindow(hwnd);
}
```

The view window always fills the frame window's client area exactly. If the user opens and closes documents or embeds objects, the changes show up in the view window.

Creating a New Server Document

See Also   Creating a View Window

For every open document, the server needs to create two helper objects: *TOcDocument* and *TOcRemView*. The document helper manages the collection of objects inserted in the document. (It is possible for objects to be embedded within objects.) the view helper connects the document to OLE. More specifically, it implements interfaces that OLE can call to communicate with the document. When OLE tells the view object that something noteworthy has occurred, the view object sends a message to the view window. (The next two sections show how to handle the messages.)

The sample CPPOCF2 server creates the two helpers in its factory when asked to create a new document.

```
OcDoc     = new TOcDocument(*OcApp);
HwndView  = CreateViewWindow(HwndMain);
OcRemView = new TOcRemView(*OcDoc, &DocReg);
if (IsWindow(HwndView))
  OcRemView->SetupWindow(HwndView);
```

The *SetupWindow* method tells the *TOcRemView* object where to send event messages. In this case, it sends messages to *HwndView*, the view window. The view window now receives WM_OCEVENT messages.

For each new document a server creates a *TOcDocument*, a *TOcRemView*, and a view window. The same objects are deleted or released when the view window is destroyed:

```
void
ViewWnd_OnDestroy(HWND hwnd)
{
  .
  .
  .                                // other document cleanup can go here

  if (OcRemView)
    OcRemView->ReleaseObject();    // do not delete a TOcRemView object

  if (OcDoc) {
    OcDoc->Close();                // release the servers for any embedded
parts
    delete OcDoc;                  // this is not a COM object, so you can
delete it
  }

  if (IsWindow(HwndMain))
    PostMessage(HwndMain, WM_CLOSE, 0, 0);
  HwndView = 0;
}
```

When the view window is destroyed, it makes three calls to dispose of the helper objects. *OcRemView->ReleaseObject* signals that the view window is through with the *TOcRemView* connector object. You shouldn't call **delete** for a view object because the OLE system might still need more information before it allows the view to shut down. *ReleaseObject* decrements an internal reference count and dissociates the view from its window. When all the clients of the view object have released it, the count reaches 0 and the object destroys itself.

The *TOcDocument* view object, on the other hand, is not a connector object and so you can destroy it with delete in the usual way. First, however, you should call *Close* to release the server applications that OLE may have invoked to support objects embedded in the server's document.

Because CPPOCF2 never opens more than one document at a time, it declares *OcDoc* and

OcRemView as static global pointers.

```
TOcDocument*  OcDoc       = 0;
TOcRemView*   OcRemView   = 0;
```

A server that supports concurrent clients with a single instance of the application, or one that uses the multidocument interface (MDI), needs to create a different *TOcDocument* and *TOcRemView* pair for each document window.

Note: When launched to support an object in a container, servers create *TOcRemView* instead of *TOcView* because they are painting in a remote window. For simplicity, CPPOCF2 always creates a remote view even when it is launched directly by the user. The only penalty is extra overhead.

See Also

TOcDocument

TOcRemView

Handling WM_OCEVENT

See Also Creating a View Window

Because the *TOcRemView::SetupWindow* method bound the *OcRemView* connector to the view window, the connector sends its event notification messages to the window. All ObjectComponents events are sent in the WM_OCEVENT message, so the view window procedure must respond to WM_OCEVENT.

```
long CALLBACK _export
ViewWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
  switch (message) {
.
.
.  // other message crackers go here
    HANDLE_MSG(hwnd, WM_OCEVENT,  ViewWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}
```

The HANDLE_MSG message cracker macro for WM_OCEVENT is defined in the ocf/ ocfevx.h header. The same header also defines a another cracker for use in the WM_OCEVENT message handler.

```
// Subdispatch OC_VIEWxxxx messages
long
ViewWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
{
  switch (wParam) {
    // insert an event cracker for each OC_VIEWxxxx message you want to
handle
    HANDLE_OCF(hwnd, OC_VIEWCLOSE, ViewWnd_OnOcViewClose);
  }
  return true;
}
```

The WM_OCEVENT message carries an event ID in its *wParam*, just as WM_COMMAND messages carry command IDs. OC_VIEWCLOSE is one possible event, indicating that it is time to close this view. In applications that show only one view per document, OC_VIEWCLOSE also signals the close of the document. The HANDLE_OCF macro calls the handler you designate for each ObjectComponents event, just as HANDLE_MSG calls the handler for for a window message.

CPPOCF2 handles only the OC_VIEWCLOSE message. To handle others, add one HANDLE_OCF macro for each event ID.

See Also

OC_VIEWxxxx Messages

WM_OCEVENT message

Handling Selected View Events

See Also Creating a View Window

Each HANDLE_OCF macro calls a different handler function. In the example, the handler function is called *ViewWnd_OnOcViewClose*.

```
bool
ViewWnd_OnOcViewClose(HWND hwnd)
{
  DestroyWindow(hwnd);
  return true;
}
```

A server receives this message when a container closes the document that contains the server's object. CPPOCF2 responds by closing the view window. The WM_DESTROY handler also deletes or releases the helper objects associated with the server document.

See Also
OC_VIEWxxxx Messages

Painting the Document

Creating a View Window

No special code is required in the server's paint procedure. It always paints its document the same way, whether or not it is painting an embedded object.

```
void
ViewWnd_OnPaint(HWND hwnd)
{
  PAINTSTRUCT ps;
  HDC dc = BeginPaint(hwnd, &ps);
  wsprintf(Buffer, "%u", Counter);
  TextOut(dc, 0, 0, Buffer, lstrlen(Buffer));
  EndPaint(hwnd, &ps);
}
```

When the view window is created, it starts off a timer. Every time the view receives a WM_TIMER message, it increments the value in the global variable *Counter* and calls *InvalidateRect* to make the view repaint itself. On each call, the paint procedure prints the value of *Counter*.

Programming the Main Window

Turning a C++ Application Into an OLE Server

The view window manages tasks related to a single document. It opens and closes the document and draws it on the screen. The frame window manages tasks for the whole application. It responds to menu commands, and it creates and destroys the view window.

The following topics discuss tasks needed to program the main window:

- Creating the Main Window
- Handling Wm_Ocevent
- Handling Selected Application Events

Creating the Main Window

See Also Programming the Main Window

When the application creates its main window, it must bind the window to its *TOcApp* object. (The *TOcApp* object was created in the factory callback function.)

```
bool
MainWnd_OnCreate(HWND hwnd, CREATESTRUCT FAR* /*lpCreateStruct*/)
{
  if (OcApp)
    OcApp->SetupWindow(hwnd);

  HwndMain = hwnd;
  return true;
}
```

The *TOcApp* object sends messges about OLE events to the application's main window. *SetupWindow* tells the *TOcApp* where to direct its event notifications.

See Also
TOcApp

Handling WM_OCEVENT

See Also Programming the Main Window

Like the view window, the frame window also receives WM_OCEVENT messages. The frame window receives notification of events that concern the application as a whole and not just a particular document. The frame window procedure sends WM_OCEVENT messages to a handler that identifies the event and calls the appropriate handler routine. Both routines closely resemble the corresponding code for the view window.

```
// Standard message-handler routine for main window
long CALLBACK _export
MainWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
  switch (message) {
.
.
.  // other message crackers go here
    HANDLE_MSG(hwnd, WM_OCEVENT,  MainWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}

// Subdispatch OC_... Messages
long
MainWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
{
  switch (wParam) {
    HANDLE_OCF(hwnd, OC_APPSHUTDOWN, MainWnd_OnOcViewTitle);
  }
  return true;
}
```

See Also
WM_OCEVENT message

Handling Selected Application Events

See Also Programming the Main Window

The only ObjectComponents event that CPPOCF2 can handle in its main window is OC_APPSHUTDOWN. A server receives this message when the last linked or embedded object closes down. If the server was launched by OLE, it can terminate. If user launched the server directly, the server doesn't need to do anything.

```
const char*
MainWnd_OnOcAppShutDown(HWND hwnd)
{
  if (OcRegistrar->IsOptionSet(amEmbedding))
    DestroyWindow(hwnd);
}
```

The registrar sets the *amEmbedding* flag at startup if it finds the -Embedding switch on the application's command line. OLE pass the -Embedding switch when it launches a server to support a linked or embedded object.

See Also
Processing the Command Line

Building the Server

To build the server, you need to include the right headers, use a supported memory model, and link to the right libraries.

The following topics discuss tasks needed to build the server:

- Including Objectcomponents Headers
- Compiling and Linking

Including ObjectComponents Headers

Building the Server

The following list shows the headers needed for an ObjectComponents server that does not use ObjectWindows.

```
#include <ocf/ocapp.h>       // TOcRegistrar, TOcModule, TOcApp (application
connector)
#include <ocf/ocreg.h>       // registration constants and app mode flags
#include <ocf/ocdoc.h>       // TOcDocument (compound document manager)
#include <ocf/ocview.h>      // TOcView (document view connector)
#include <ocf/ocpart.h>      // TOcPart (linked/embedded object connector)
#include <ocf/ocremvie.h>    // TOcRemView (document remote view connector)
#include <ocf/ocfevx.h>      // WM_OCEVENT message crackers
```

Compiling and Linking

Building the Server

ObjectComponents applications that do not use ObjectWindows work with either the medium or large memory model. They must be linked with the OLE and ObjectComponents libraries.

To build CPPOCF0, CPPOCF1, and CPPOCF2, move to the program's directory and type this at the command prompt:

```
make MODEL=l
```

This command builds all three programs using the large memory model.

The make file that builds this example program refers to the OCFMAKE.GEN file

Understanding Registration

See Also Creating an OLE Server

The BEGIN_REGISTRATION and END_REGISTRATION macros declare a structure to hold registration keys and the values assigned to the keys. The macros that come in between, such as REGDATA and REGFORMAT, each insert an item in the structure. The main structure is of of type *TRegList* and each item in the structures is an entry of type *TRegItem*. Each item contains a key and a value for that key.

```
struct TRegItem {
  char* Key;             // standard registry key
  TLocaleString Value;  // value you assign to the key
};
```

The two parameters passed to REGDATA are a key and a value. The macros make it easy to add keys and values to the structure without having to manipulate *TRegList* and *TRegItem* objects directly yourself. At run time, ObjectComponents scans the tables and confirms that the information is stored accurately in the system registration database.

The following topics discuss more about understanding registration:

- Storing Information in the Registration Database
- Registering Localized Entries
- Registering Custom Entries

See Also
Registration Macros (OWL.HLP)
BEGIN_REGISTRATION macro (OWL.HLP)
END_REGISTRATION macro (OWL.HLP)
REGDATA macro (OWL.HLP)

Storing Information in the Registration Database

Understanding Registration

ObjectComponents copies information from the program's registration tables to the system's registration database. The *TOcRegistrar* object takes care of this chore when it is constructed. Every time a server constructs its *TOcRegistrar* object, ObjectComponents confirms that the registration information is accurately recorded. ObjectComponents compares the program's current *progid*, *clsid*, and executable path with the values previously written in the registration database. Any discrepancy causes ObjectComponents to reregister the entire program automatically.

Windows 3.1 stores the registration database in the REG.DAT file. Windows NT puts it in the system registry, a facility managed privately by the operating system. In either location, ObjectComponents follows the standard logical structure for recording information about an OLE server. To inspect the entries in your registration database, run RegEdit with the /v command-line option.

Note: In 16-bit Windows, the registration database has a capacity of 64K. If the database fills past its capacity, OLE behaves unpredicatably, and it might be necessary to erase the your REG.DAT file. Then you need to reregister the OLE applications in your system. You can register or unregister any ObjectComponents server by passing the RegServer or UnregServer switch on its command line. Unregistering unused applications or obsolete versions is a good way to converve space in the database.

To learn more about the registration database, search for the topic "Registration Database" in the online Help file for 16-bit Windows programming (WIN31WH.HLP).

Registering Localized Entries

See Also Understanding Registration

The values assigned to registration parameters often need to be localized. Besides putting translations in your resources, you must also mark the strings so that ObjectComponents can tell which ones have localized versions.

```
REGDATA(description, "@myapp_description")
```

The @ prefix tells ObjectComponents that the string *myapp_description* is an identifier for an XLAT resource where the real description is stored in several languages. For more information about localizing OLE strings, refer to the *TLocaleString* entry in the *ObjectWindows Reference Guide*.

See Also
Localizing Symbol Names

Registering Custom Entries

See Also Understanding Registration

databaseThe REGDATA macro associates strings with keys. The registrar scans the list of keys and, when needed, writes the associates strings in the system registration database. The standard keys such as *progid* and *clsid* correspond to standard entries in the database. If you want to register values for non-standard keys, use the databaseREGITEM macro. The first parameter for REGITEM is the complete key exactly as you would pass it to a function like *RegOpenKey*.

```
REGITEM("CLSID\\<clsid>\\Conversion\\Readable\\Main","FormatX,FormatY")
```

As the example shows, the REGITEM arguments can contain embedded parameter names enclosed in angle brackets. When ObjectComponents registers this item, it first replaces the expression <*clsid*> with whatever value the registration block has assigned to the clsid parameter.

For information about *RegOpenKey*, see the Help file for the Windows API.

See Also

REGDATA macro (OWL.HLP)

Registration Macros (OWL.HLP)

Making a DLL Server

Typically, linking and embedding servers are stand-alone executables that can be launched directly by the user or invoked indirectly by an OLE container. You can also implement an OLE server in a DLL. A server built as a DLL is sometimes called an in-process server because DLL code runs in the same process as its client. The terms EXE *server* and *server application* refer specifically to a server implemented in an EXE. ObjectComponents allows you to create both EXE and DLL servers. If you are using ObjectWindows, converting from one to the other requires only two simple changes.

Note: The discussion and instructions that follow apply to automation servers as well as linking and embedding servers.

The following topics discuss more about making a DLL server:

- Pros and Cons of DLL Servers
- Building a DLL Server
- Debugging a DLL Server
- Tools for DLL Servers

Pros and Cons of DLL Servers

Making a DLL Server

Advantages

The major advantage of DLL servers is performance. Because a DLL server lives in the address space of the container,it loads and responds very fast. An EXE server, on the other hand, is a   separate process and requires some form of intertask communication to interact with a container. OLE serializes intertask calls and marshals the function calls with their parameters, packaging them into the proper format for the interprocess protocol. (The protocol it uses is called LRPC, for Lightweight Remote Procedure Call.) the process of serializing the drawing commands in a metafile is particularly slow, so DLL servers substantially increase the speed of creating presentation data for linked and embedded objects.

Disadvantages

There are a few disadvantages to using a DLL server, however. While OLE supports interaction between 16-bit and 32-bit executable applications, a 16-bit Windows application cannot use a 32-bit DLL server and a Win32 application cannot use a 16-bit DLL server. Also, DLLs do not have message queues. As a result, a DLL server cannot easily perform a task in the background. ObjectWindows overcomes this limitation by running a timer so that it can still call the *IdleAction* methods of objects derived from *TApplication* or *TOleFrame*. (ObjectWindows also uses the timer for internal processes such as command-enabling for tool bars, deleting condemned windows, and resuming thrown exceptions.)

Because a DLL server becomes part of the container's process, bugs in one can interfere with the other, making DLL servers sometimes harder to debug.

DLL servers also present user interface dilemmas. For example, when a container initiates an open edit session with a server, it doesn't matter to the user whether the server is an EXE or a DLL; the user interface for open editing is the same either way. But the lifetime of a DLL server is tied to the container that loads it. When the container quits, the server DLL is unloaded. That can cause problems if the server's user interface normally allows the user to edit serveral documents at once. If the user were to create a new document while editing an embedded object, the user might want to continue editing the new document even after the container quits, but then the server is no longer in memory. This is a particular problem for MDI servers because the MDI interface allows users to open multiple documents in a single session. Typically DLL servers do not allow multidocument editing.

Finally, DLL servers have one other disadvantage. While OLE 2 provides a compatibility layer to let OLE 2 servers interact with OLE 1 clients, the compatibility layer works only for EXE servers. A DLL server cannot support an OLE 1 client.

Building a DLL Server

Making a DLL Server

Converting an ObjectWindows EXE server to a DLL server requires only a few modifications.

The following topics discuss the tasks involved with building a DLL server.

1. Update Your Document Registration Table
2. Compile and Link

Updating Your Document Registration Table

See Also Building a DLL Server

The document registration tables of DLL servers must contain the *serverctx* key with the string value "Inproc." This allows ObjectComponents to register your application as a DLL server with OLE. EXE servers do not need to use the *serverctx* key since ObjectComponents defaults to EXE registration.

The following code illustrates the document registration structure of a DLL server. It comes from the sample Tic Tac Toe program in EXAMPLES/OWL/OCF/TTT.

```
BEGIN_REGISTRATION(DocReg)
 REGDATA(progid,      "TicTacToeDll")
 REGDATA(description,"TicTacToe DLL")
 REGDATA(serverctx, "Inproc")
 REGDATA(menuname,    "TicTacToe Game")
 REGDATA(insertable,  "")
 REGDATA(extension,   "TTT")
 REGDATA(docfilter,   "*.ttt")
 REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
 REGDATA(verb0,       "&Play")
 REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
 REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,   ocrGet)
END_REGISTRATION
```

You do not need to modify your application registration structure to convert your EXE server to a DLL server. It's a good idea, however, to use different *clsid* and *progid* values, especially if you intend to switch frequently from one type to the other. You can test for the BI_APP_DLL macro to declare a registration structure that works for both DLL and EXE servers; the macro is only defined when you are building a DLL. The following code shows a sample document registration which supplies two sets of *progid* and *clsid* values.

```
REGISTRATION_FORMAT_BUFFER(100)

// Application registration structure
BEGIN_REGISTRATION(AppReg)
#if defined(BI_APP_DLL)
 REGDATA(clsid,       "{029442B1-8BB5-101B-B3FF-04021C009402}")   // DLL
clsid
 REGDATA(progid,      "TicTacToe.DllServer")                      // DLL
progid
#else
 REGDATA(clsid,       "{029442C1-8BB5-101B-B3FF-04021C009402}")   // EXE
clsid
 REGDATA(progid,      "TicTacToe.Application")                    // EXE
progid
#endif
 REGDATA(description,"TicTacToe Application")                     //
Description
END_REGISTRATION

// Document registration structure
BEGIN_REGISTRATION(DocReg)
#if defined(BI_APP_DLL)
 REGDATA(progid,      "TicTacToeDll")
 REGDATA(description,"TicTacToe DLL")
 REGDATA(serverctx, "Inproc")
#else
```

```
 REGDATA(progid,      "TicTacToe.Game.1")
 REGDATA(description,"TicTacToe Game")
 REGDATA(debugger,    "tdw")
 REGDATA(debugprogid,"TicTacToe.Game.1.D")
 REGDATA(debugdesc,  "TicTacToe Game (debug)")
#endif
 REGDATA(menuname,    "TicTacToe Game")
 REGDATA(insertable, "")
 REGDATA(extension,  "TTT")
 REGDATA(docfilter,  "*.ttt")
 REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
 REGDATA(verb0,       "&Play")
 REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
 REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,    ocrGet)
END_REGISTRATION
```

Notice that the debugger keys (*debugger*, *debugprogid*, and *debugdesc*) are not used when building a DLL server. They are relevant only when your server is an executable that a debugger can load.

See Also
Debugging a DLL Server
debugdesc Registration Key
debugprogid Registration Key

Compiling and Linking

Building a DLL Server

'ObjectWindows DLL servers must be compiled with the large memory model. They must be linked with the OLE, ObjectComponents, and ObjectWindows libraries.

The integrated development environment (IDE) chooses the right build options for you when you select *Dynamic Library* for Target Type and request OWL and OCF support from the list of standard libraries. You may choose to link with the static or dynamic versions of the standard libraries.

To build an ObjectWindows DLL server from the command line, create a short makefile that includes the OWLOCFMK.GEN file found in the EXAMPLES subdirectory. Here, for example, is the makefile that builds the sample program Tic Tac Toe:

```
MODELS=ld
SYSTEM = WIN16

DLLRES = ttt
OBJDLL = ttt.obj

!include $(BCEXAMPLEDIR)\owlocfmk.gen
```

DLLRES holds the base name of your resource file and the final DLL name. OBJDLL holds the names of the object files from which to build the sample. Finally, your makefile should include the OWLOCFMK.GEN file.

Name your file MAKEFILE and type this at the command-line prompt:

```
make MODEL=l
```

Make, using instructions in OWLOCFMK.GEN, builds a new makefile tailored to your project. The command also runs the new makefile to build the program. If you change the command to define MODEL as *d*, the above command will create a new makefile and then build your DLL using the DLL version of the libraries instead of the large model static libraries.

For more information about how to use OWLOCFMK.GEN, read the instructions at the beginning of MAKEFILE.GEN, found in the examples directory.

Debugging a DLL Server

Making a DLL Server

The same general techniques used to debug DLLs apply to DLL servers. The steps that follow describe one approach, using Turbo Debugger for Windows to set breakpoints in a DLL server.

1. Build and register the DLL server.
   a. Build your server with debugging information.
   b. Register the server using the REGISTER.EXE utility.
   c. Verify that the registration was successful by running RegEdit and looking for your servers file types. (RegEdit is a registration editor included with Windows.)
2. Launch Turbo Debugger for Windows and load a container.

   Select the File|Open menu option and enter the container's name in the Program Name field and click the OK button.

   The debugger loads the container. If the container was built without debugging information, you may receive a warning. You can safely ignore it.
3. Load the server's debugging information.
   a. Select View|Module from the debugger's main menu. This activates the dialog titled "Load module source or DLL symbols." (You can also activate the module dialog by pressing F3.)
   b. Enter the full name of DLL server file in the DLL Name field.
   c. Select the Yes option in the Debug Startup field and click the Add DLL button. The name of your DLL server (followed by *!!*) appears as the selected entry in the DLLs & Programs list.
   d. Click the Symbol Load button.

   If you receive an error message indicating that the DLL is not loaded , press the Escape key to return to the debugger's main menu and proceed to the next step. Otherwise, skip the next step and proceed to step 5.

Note: If you did not receive the error message that the DLL is not loaded, then your DLL server was already in memory before the container activated it. This happens if another container is currently running with one of your server's objects. More often, however, it indicates that your server crashed or was improperly terminated in an earlier session.

4. Run the container and insert one of your server's objects.
   a. Select the Run|Run menu option (or press F9) to start the container.
   b. Choose Insert Object from the container's Edit menu and insert your server's object.

   The debugger pops up as soon as OLE loads your DLL server.
5. Display the DLL source modules and set breakpoints.
   a. Choose View|Module from the container's main menu to see the names of the source files used to build your server. The file names appear in Source Modules list.
   b. Select source files by double-clicking the file names.
   c. Set breakpoints in your server.
   d. Choose the Run|Run menu option (or press F9) to return control to the container application.
6. If you skipped step 4, insert one of your server's objects into the container now.

The debugger stops at the breakpoints set in your source files and allows you to step through your server, inspect variables, and verify the logic of your code.

Tools for DLL Servers

Before running your DLL server, you must record its registration information in the system registration database. The Register tool does that for you. Another tool, DllRun, gives you the option of running your DLL server at any time as a standalone application, which is sometimes convenient for testing.

The following topics discuss these two useful tools:

- Registering Your DLL Server
- Running Your DLL Server

Registering Your DLL Server

Tools for DLL Servers

The REGISTER.EXE utility registers an ObjectComponents DLL server. On the command line, pass Register the name of your server followed by the -RegServer switch. Here is the command to register Tic Tac Toe:

```
register ttt.dll -RegServer
```

Even though the Register utility is a Windows application, not a DOS application, you can invoke it from a Windows DOS box. This ability is useful in makefiles. (To invoke other Windows programs from a DOS box command line, use the WinRun utility described in UTILS.TXT.)

Register can also unregister your server. Unregistering removes all entries related to your server from the registration database. It's good practice to unregister one version before you register the next. To unregister, use the -UnregServer switch. This command unregisters Tic Tac Toe:

```
register ttt.dll -UnregServer
```

Running your DLL server

See Also Tools for DLL Servers

The DLLRUN.EXE utility lets you load and run an ObjectComponents DLL server as though it were a standalone executable program. The ability to run in executable mode is useful for debugging. It also lets you give customers the choice of running your server either way without having to distribute two versions of the same application.

on the command line, pass DllRun the *progid* of the server. This is the value assigned to the *progid* key in the server's registration table. This command runs the Tic Tac Toe server:

```
dllrun TicTacToeDll
```

DllRun launches the DLL server in the executable running mode. The running mode of an ObjectComponents application is represented by a set of bit flags that you can test by calling *TOcModule::IsOptionSet*. (Remember that the application object of a linking and embedding program derives from both *TApplication* and *TOcModule*.)

The running mode bit flags are defined in the *TOcAppMode* **enum**. *AmEmbedded* is set when the server is invoked by OLE, not by the user. *AmExeModule* is set in an application that was built as an EXE. *AmExeMode* is set in an application that is running as a stand-alone executable, even if it was built as a DLL.

This code tests the flags to determine the server's running mode.

```
void
TMyApp::TestMode()
{
  if (IsOptionSet(amExeMode))             // is server running as an EXE?
    if (!IsOptionsSet(amExeModule)) {     // if so, was it built as an EXE?
      // the server is a DLL running in EXE mode
    } else {
      // the server was built as an EXE
  } else {
    // the server is a DLL running in a client's process
}
```

See Also

TOcAppMode enum

TOcModule

ObjectComponents Libraries

See Also

An ObjectComponents application links to an OCF*xxxx* library and to the OLE2W16 library.

Library	Description
OCFWI.LIB	For use with the dynamic library versions of the RTL and class libraries
OCFWIU.LIB	For use in a user DLL with the dynamic library versions of the RTL and class libraries
OCFWL.LIB	Large model
OCFWLU.LIB	Large model for use in a user DLL
OCFWM.LIB	Medium model ObjectComponents
OLE2W16.LIB	Import library for OLE system DLLs

See Also

ObjectComponents Framework Language Reference

What is ObjectComponents?

Automating an Application

Creating an OLE Container

Support for OLE in Borland C++

ObjectComponents Header Files

Header files, located in your INCLUDE/OCF subdirectory, contain declarations for class functions and definitions for data types and constants.

File	Contents
appdesc.h	TAppDescriptor, TComponentFactory
autodefs.h	Classes and **structs** used for automation
automacr.h	Macros for automation declarations and definitions
ocapp.h	OC_APPxxxx messages, TOcApp, TOcFormatName, TOcHelp, TOcMenuDescr, TOcModule, TOcNameList, TOcRegistrar, WM_OCEVENT message
ocapp.rh	IDS_CFxxxx resource IDs for strings describing standard Windows clipboard formats
ocdefs.h	TXObjComp, HR_xxxx Return constants, declaration specifiers
ocdoc.h	TOcDocument
ocfevx.h	Message cracker macros for WM_OCEVENT
ocfpch.h	**#include** statements for all ObjectComponents headers, used with pre-compiled headers
ocobject.h	TOcAspect, TOcDialogHelp, TOcDropAction, TOcInitHow, TOcInitInfo, TOcInitWhere, TOcInvalidate, TOcPartName, TOcScrollDir
ocpart.h	TOcPart, TOcPartCollection, TOcPartCollectionIter, TOcVerb
ocreg.h	OCxxxx global functions, ocrxxxx registration constants, TAppMode **enum**, TRegistrar, TXRegistry
ocremvie.h	TOcRemView
ocstorag.h	TOcStream, TOcStorage
ocview.h	OC_VIEWxxxx messages, TOcDragDrop, TOcFormat, TOcFormatList, TOcFormatListIter, TOcSaveLoad, TOcScaleFactor, TOcToolBarInfo, TOcView, TOcViewPaint
oleutil.h	DECLARE_COMBASES# macros, TOleAllocator, TUnknown, TXOle

General OLE Classes, Macros, and Type Definitions

Description

ObjectComponents provides the following utility items for use in building OLE applications, whether they support linking and embedding or automation.

Item	Meaning
HR_xxxx macros	Return values from OLE functions
_ICLASS macro	Specifier for declaring a class that implements an OLE interface
_IFUNC macro	Specifier for declaring OLE functions
_OCFxxxx macros	Specifiers for declaring ObjectComponents classes
TComponentFactory typedef	Callback function where an application creates objects that OLE requests
TLocaleId typedef	Data type for language setting identifiers
TOleAllocator class	Establishes a memory allocator for OLE to use
TUnknown class	Implements the fundamental *IUnknown* interface required of all OLE objects

Global Utility Functions

The items in this table are utility functions that ObjectComponents declares globally.

Item	Purpose
DynamicCast	Converts a pointer from one type to another if both types are related through inheritance
MostDerived	Returns a pointer to the most derived class type that fits a given object
OcRegisterClass	Writes all the information from one registration table to an output stream
OcRegistryUpdate	Merges all the information from an input stream into the registration database
OcRegistryValidate	Checks whether information in a registration table matches the corresponding information already recorded in the registration database
OcSetupDebugReg	Takes the information from a registration table and writes to an output stream the registration entries for a debugging version
OcUnRegisterClass	Removes entries from the system's registration database

ObjectComponents Exception Classes

See Also

ObjectComponents throws the types of exceptions shown in this list. All the exception classes derive from TXBase.

Class	Purpose
TXAuto	Exceptions that occur during automation
TXObjComp	Exceptions that occur during ObjectComponents linking and embedding operations
TXOle	Exceptions that occur while processing OLE API commands
TXRegistry	Exceptions that occur while using the system registration database

See Also

TXBase Class (OWL.HLP)

Exception Handling in ObjectComponents

Automation Classes

See Also Overview

ObjectComponents provides the following classes that support automation.

TAutoBase	Base class for deriving automated objects
TAutoCommand	Holds all the data for one command received by an automation server
TAutoEnumerator<>	Lets an automation controller enumerate items in a server's collection
TAutoIterator	Lets an automation server iterate items in an automated collection
TAutoObject<>	Creates a smart pointer to an automated object
TAutoObjectByVal<>	Lets an automation server automate a method that returns an object by value (not by reference)
TAutoObjectDelete	Lets an automation server automate a method that returns an object
TAutoProxy	Base class for deriving the C++ objects an automation controller creates to represent the OLE objects it wants to control
TAutoStack	Holds a set of *TAutoCommand* objects each representing a command received by an automation server
TAutoVal	Holds the data from a VARIANT union, the data format OLE uses for sending automation commands
TRegistrar	Manages system registration tasks for an automation application

See Also
Automating an Application
Automation Data Types
Automation Macros
Automation Enumerated Types and Type Definitions
Automation Structs
Overview of Automation Classes
Registration Keys

Automation Enumerated Types and Type Definitions

See Also

ObjectComponents provides the following items that support automation.

AutoDataType enum	Identifiers for automation data types
AutoSymFlag enum	Flags that describe attributes of an automation command
ObjectPtr typedef	**void** pointer to a C++ object

See Also

Automation Classes

Automation Data Types

Automation Macros

Automation Structs

Registration Keys

Automation Data Types

See Also Declarations and Definitions

ObjectComponents provides the following data types that support automation.   To use these data types, see Declarations and Definitions.

TAutoBool struct

TAutoCurrency struct

TAutoCurrencyRef struct

TAutoDate struct

TAutoDateRef struct

TAutoDouble struct

TAutoDoubleRef struct

TAutoFloat struct

TAutoFloatRef struct

TAutoInt typedef

TAutoLong struct

TAutoLongRef struct

TAutoShort struct

TAutoShortRef struct

TAutoString struct

TAutoVoid struct

See Also

Automation Classes

Automation Enumerated Types and Type Definitions

Automation Macros

Automation Structs

Data Type Specifiers in an Automation Definition

Registration Keys

Declarations and Definitions of Automation Data Types

See Also

An automation data type is a structure that exists solely to describe a single type of data. Automation definitions use these structures to assist in converting parameters and return values between the VARIANT unions that OLE uses and the C++ data types that your programs use.

For the most part, although they are structures, you cannot create instances of them because they lack constructors and contain only a single static member. They all derive from TAutoType and inherit its *GetType* method. The only thing most of these structures do is respond to *GetType* calls by returning the static ID for a data type.

Header File
ocf/autodefs.h

Description
The following table lists C++ data types that might appear in your programs and the corresponding data types that you should use in automation declarations (DECLARE_AUTOCLASS) and definitions (DEFINE_AUTOCLASS).

In the left column, find a type that your automated class uses in its arguments or its return values. The other columns tell what data type to specify in the corresponding entries of the automation declaration and definition.

C++ Type	**Declaration Type** (DECLARE_AUTOCLASS)	**Definition Type** (DEFINE_AUTOCLASS)
short	short	TAutoShort
unsigned short	short or unsigned	TAutoShort or TAutoLong
long	long	TAutoLong
unsigned long	unsigned long	TAutoLong (treated as signed long)
int	int	TAutoInt
unsigned int	int or long	TAutoInt or TAutoLong
float	float	TAutoFloat
double	double	TAutoDouble
bool (or int)	TBool	TAutoBool
TAutoDate	TAutoDate	TAutoDate
TAutoCurrency	TAutoCurrency	TAutoCurrency
char*	TAutoString	TAutoString
const char*	TAutoString	TAutoString
char far*	TAutoString	TAutoString
const char far*	TAutoString	TAutoString
string	string	TAutoString
enum	short or int	TAutoShort, TAutoInt, or user-defined AUTOENUM
T*	TAutoObject<>	T (class T must be automated)
T&	TAutoObject<>	T (class T must be automated)
const T*	TAutoObject<>	T (class T must be automated)
const T&	TAutoObject<>	T (class T must be automated)
T* (returned)	TAutoObjectDelete<>	(C++ object deleted when no refs)

T& (returned)	TAutoObjectDelete<>	(C++ object deleted when no refs)
T (returned)	TAutoObjectByVal<>	T (T copied, deleted when no refs)
void (no return)	(use AUTOFUNC*n*V macros)	TAutoVoid
short far*	short far*	TAutoShortRef
long far*	long far*	TAutoLongRef
float far*	float far*	TAutoFloatRef
double far*	double far*	TAutoDoubleRef
TAutoDate far*	TAutoDate far*	TAutoDateRef
TAutoCurrency far*	TAutoCurrency far*	TAutoCurrencyRef

See Also

Automation Data Types

Data Type Specifiers in an Automation Definition

Automation Declaration Macros

See Also

Header File
ocf/automacr.h

Description
To make parts of an automated class accessible to OLE, an automation server adds declaration macros to the declaration of the C++ class and definition macros to the implementation of the C++ class. The declaration macros create command objects for executing commands sent by the controller.

The block of automation declaration macros always begins with DECLARE_AUTOCLASS or DECLARE_AUTOAGGREGATE macro. There is no need for a matching END macro to close the declaration.

Macro	Meaning
DECLARE_AUTOCLASS(cls)	The macros that follow declare automatable members of the user-defined class *cls*.
DECLARE_AUTOAGGREGATE(cls)	The macros that follow declare automatable members of a class that is, inherits from, or delegates to a COM object.

Declaration Macros
After DECLARE_AUTOCLASS comes a series of macros, one for each class member that you choose to expose. Which particular macros you choose depends on what the members are.

Declaration Macro	Member
AUTODATA	Data
AUTOFLAG	A bit flag
AUTOFUNC	Function
AUTOITERATOR	Iterator object
AUTOPROP	Property
AUTOPROXY	Property containing an automated object
AUTOSTAT	Static member or global function
AUTOTHIS	***this**

AUTODETACH macro
In addition, an automation declaration can also include the AUTODETACH Macro. This macro does not expose a class member. It invalidates external references when the object is destroyed.

See Also
Automation Definition Macros
Automation Hook Macros
Declaring Automatable Methods and Properties
TAutoStack::operator []

Automation Definition Macros

See Also

Header File
ocf/automacr.h

Description
To make parts of an automated class accessible to OLE, an automation server adds declaration macros to the declaration of the C++ class and definition macros to the implementation of the C++ class.

The block of automation definition macros begins with DEFINE_AUTOCLASS and ends with END_AUTOCLASS, unless the object is, inherits from, or delegates to a Component Object Model (COM) object. In that case, the block of automation definition macros begins with DEFINE_AUTOAGGREGATE and ends with END_AUTOAGGREGATE.

Macro	Meaning
DEFINE_AUTOCLASS(cls)	The macros that follow define automatable members of the user-defined class *cls*.
END_AUTOCLASS(cls, name, doc, help)	The C++ class *cls* is exposed to OLE controllers under the name *name*. If the user asks OLE about the object name, the system returns the string in *doc*. If a .HLP file is registered for the object, then the context ID in *help* points to a screen that describes the object.
DEFINE_AUTOAGGREGATE(cls, aggregator)	The macros that follow define automatable members of the user-defined class *cls*, which is, inherits from, or delegates to a COM object.
END_AUTOAGGREGATE(cls, name, doc, help)	Same as END_AUTOCLASS.

Between the DEFINE and END macros comes a series of other macros describing each exposed data member or function. The macros implement methods for a class nested within your automated class. When ObjectComponents receives commands from a controller, it passes them to the nested class. The macros build wrapper functions in the nested class that enable it to call your own class directly.

Which particular macros you choose depends on what the members are.

Member	Declaration Macro
Automated application	EXPOSE_APPLICATION Macro
Auxiliary class	EXPOSE_DELEGATE Macro
Base class	EXPOSE_INHERIT Macro
Collection iterator	EXPOSE_ITERATOR Macro
Method	EXPOSE_METHOD Macros
Read-only property	EXPOSE_PROPRO
Read-write property	EXPOSE_PROPRW
Write-only property	EXPOSE_PROPWO
Shutdown method	EXPOSE_QUIT

See Also
Automation Declaration Macros
Data Type Specifiers in an Automation Definition
Defining External Methods and Properties
OPTIONAL_ARG Macro
REQUIRED_ARG Macro
TAutoStack::operator []

Automation Hook Macros

See Also

Header File
ocf/automacr.h

Description
These macros establish hooks to be invoked every time a particular automation command is executed. They are never used by themselves but always as the last parameter of some other automation declaration macro. If you add one of these hooks to the declaration of some exposed class member, then every time an automation controller attempts to execute that command, ObjectComponents first executes the code in the hook. The code can be a simple expression or it can contain calls to other functions.

Most of the macros expect to receive some expression or code as a parameter. Often the code or expression in the macro needs to refer to the arguments passed in or to the value of an automated data member. Within the macro expression, write *Arg1*, *Arg2*, *Arg3*... to refer to the received arguments. Write *Val* to refer to an automated data member.

Macro	Meaning
AUTONOHOOK	Use this macro, without arguments, to prevent anyone from hooking the command. Not even ObjectComponents can monitor the call. (For advanced uses only.)
AUTOINVOKE(code)	The code here is executed each time the automation command is executed. Create an AUTOINVOKE hook if you want to override the normal execution sequence.
AUTORECORD(code)	The code inserted here creates a record of the commands executed so that the sequence can be stored and replayed.
AUTOREPORT(code)	The code inserted here returns an error code from the automated member. If *code* evaluates to 0, OLE assumes the command succeeded. If *code* evaluates to a nonzero value, then OLE throws an exception that returns an error code to the controller. Within the code expression, use *Val* to refer to the actual value returned.
AUTOUNDO(code)	The code inserted here creates a TAutoCommand object that will undo the action of the current command.
AUTOVALIDATE(condition)	The code here should evaluate to **true** if the arguments received are valid for the command and **false** otherwise. If the expression returns **false**, OLE throws an exception that returns an error code to the controller application.

Example
This declaration ensures that an automated data member is never assigned a value outside a given range.

```
AUTODATA(Number, Number, short,
 AUTOVALIDATE(Val>=NUM_MIN && NotTooBig(Val)));
```

See Also

Automation Declaration Macros

Providing Optional Hooks for Validation and Filtering

Automation Proxy Macros

Header File

ocf/automacr.h

Description

An automation controller creates a proxy object (derived from *TAutoProxy*) to represent an automated OLE object. For every command the controller wants to send the object, it adds a method to the proxy object. The proxy methods mimic the commands the object supports. When the controller calls proxy methods, ObjectComponents sends automation commands through OLE.

The implementation of a proxy method always contains three macros: an AUTONAMES macro, an AUTOARGS macro, and an AUTOCALL macro. AUTONAMES associates names with arguments. AUTOARGS describes any arguments that do not have names. The use of names makes it possible to send partial sets of arguments and let the server assign default values to the remaining arguments.

The third macro, AUTOCALL, tells whether the command represents a method or a property of the automated object and whether the command returns a value.

To generate proxy object declarations and definitions directly, use the TYPEREAD.EXE tool (located in the OCTOOLS subdirectory.) TYPEREAD scans the type library of an automation server and generates complete proxy code for controlling the server.

You are free to subsititute your own code for the standard macros in order to handle special situations.

Macro	Description
AUTONAMES	Associates names with arguments so the caller can choose to pass only selected arguments
AUTOARGS	Describes arguments that do not have names
AUTOCALL	Tells whether the command is a method or a property and whether it returns a value

Registration Keys

See Also

Most ObjectComponents programs build registration tables describing their OLE capabilities. (Only automation controllers can omit this step.) The registration tables contain keys paired with values. The keys are standard. You decide which ones to register and you supply values for them.

Which keys you choose depends on whether your application is a server, a container, or an automation program. Some keys must be registered and some are optional. Furthermore, some apply only to the application's primary registration table, and others apply to the tables for each of the application's document types.

A registration table starts with the BEGIN_REGISTRATION macro and ends with END_REGISTRATION. In between is one macro for each key you want to register. The macro depends on the key. Most keys use the REGDATA, but there are others such as REGFORMAT and REGSTATUS. For more information, see Registration Macros in OWL.HLP.

If your application is a server, most of the information in its registration tables is recorded in the system's registration database. Putting the information there makes it possible for OLE to learn much about the server without actually loading the application into memory. For example, if an automation controller asks for information about the commands an automation server supports, OLE can locate the server's type library from an entry in the database.

Key	Meaning
appname	A short name for the application.
aspectall	Option flags that apply to all presentation aspects.
aspectcontent	Option flags for the content view of an object.
aspectdocprint	Option flags for the printed document view of an object.
aspecticon	Option flags for the iconic view of an object.
aspectthumbnail	Option flags for the thumbnail view of an object.
clsid	A GUID identifying the application.
cmdline	Arguments OLE should place on the command line when it launches the server.
debugclsid	A GUID identifying the debugging version of a server. This is always generated internally. You should never specify it directly.
debugdesc	A long string describing the debugging version of a program.
debugger	The file name and command line switches for loading your debugger.
debugprogid	A string naming the debugging version of a program. Defining this forces ObjectComponents to register debugging and non-debugging versions.
description	A string describing the application.
directory	The default directory for browsing document files.
docfilter	File specification for listing files created by the application.
docflags	Option flags for the application's documents.
extension	A three-letter file-name extension for files created by the server.
filefmt	Name of default file format.
format*n*	A Clipboard format the application supports. (Use REGFORMAT to register Clipboard formats.)
handler	A full path pointing to a library that can draw objects created by the server. Defaults to OLE2.DLL.
helpdir	Full directory where online Help for the type library resides.
iconindex	An index telling which of the icons in the server's resources represents the type of objects the server produces. (Use REGICON to register an icon.)
insertable	Indicates that the application serves its document for linking and embedding in

	container documents.
language	Locale ID currently in effect. (Set internally during automation.)
menuname	A short name for the server, used in a container's menu.
path	The path where OLE looks to find the server. This key is set internally during registration.
permid	A string that names the application without indicating any version.
permname	A string that describes the application without indicating any version.
progid	A string uniquely naming the application.
typehelp	Name of the file where online Help for the type library resides.
usage	Indicates the whether the server can support concurrent clients with a single application instance.
verb*n*	A string naming an action the server can perform with its objects.
verb*n*opt	Option flags describing the server's verbs. (Use REGVERBOPT to register verb options.)
version	Version string for the application and type library.

See Also

Localizing Registration Strings

Registering a Linking and Embedding Server

Registering a Container Application

Registering an Automation Server

Registration Macros (OWL.HLP)

Storing Information in the Registration Database

Understanding Registration

Automation struct

See Also

TAutoType

See Also

Automation Classes

Automation Data Types

Automation Enumerated Types and Type Definitions

Automation Macros

Linking and Embedding Classes

SeeAlso Overview

ObjectComponents provides the following classes for use by applications that support linking and embedding.

TOcApp	Connector object that implements BOCOLE interfaces for the application
TOcDataProvider	Provides support for serving a portion of a document
TOcDocument	Manages the parts in a container's compound document
TOcFormat	Holds information about a view's support for a particular Clipboard data format.
TOcFormatList	List of Clipboard data formats a document supports
TOcFormatListIter	Iterator for the list of Clipboard data formats a document supports
TOcFormatName	Holds strings describing a single data format that an application might encounter on the Clipboard (see TOcNameList)
TOcInitInfo	Holds information a container needs in order to place a new object in its document
TOcLinkView	A connector object that helps containers establish links to a server document
TOcModule	Base class for deriving OLE-enabled application objects
TOcNameList	Contains a collection of strings describing all the data formats that an application might find on the Clipboard
TOcPart	Connector object that a container uses to represent an object linked or embedded in one of its documents
TOcPartChangeInfo	Carries information to accompany an OC_VIEWPARTINVALID event
TOcPartCollection	Manages a collection of linked or embedded parts
TOcPartCollectionIter	Iterator for the collection of parts linked or embedded in a single document
TOcRegistrar	Manages OLE registration tasks for a linking and embedding application
TOcRemView	Connector object that a server uses to draw an object linked or embedded in a container's document
TOcScaleFactor	Carries information from a container to a server requesting that linked or embedded objects be drawn to a certain scale
TOcVerb	Holds information about an action that a server is able to perform with its own objects when they are linked or embedded in a container
TOcView	A connector object that an application uses to draw its own documents in its own frame window
TOcViewCollection	Manages a set of link views associated with a document
TOcViewCollectionIter	Enumerates the views of a compound document

See Also
General OLE Classes, Macros, and typedefs
Linking and Embedding
ObjectComponents Exception Classes
Overview of Linking and Embedding Classes
Registration Keys

Linking and Embedding Enums

ObjectComponents provides the following enumerated types for use by applications that support linking and embedding.

TOcAppMode	Flags identifying the application's running conditions
TOcAspect	Flags identifying object presentation aspects
TOcDialogHelp	Constants identifying standard OLE dialog boxes where a user can ask for help
TOcDropAction	Constants identifying actions that can result from dropping an object on a window
TOcMenuEnable	Flags that determine which OLE commands on the Edit menu should be enabled.
TOcInitHow	Constants identifying the action a container is to take on receiving a new object--either link or embed
TOcInitWhere	Constants identifying places the data for an object can reside
TOcInvalidate	Flags indicating whether an object is invalid because of a change in its data or just in its appearance
TOcPartName	Constants identifying different strings a container might request when asking for the name of an object linked or embedded in it
TOcScrollDir	Constants identifying directions a container might be asked to scroll its window

Linking and Embedding Messages

SeeAlso

ObjectComponents provides the following mesages for use by applications that support linking and embedding

OC_APPxxxx	Messages sent to an application object
OC_VIEWxxxx	Messages sent to a view object
WM_OCEVENT	Carries event signals from ObjectComponents to an application

See Also
Messages and Windows

Linking and Embedding Structs

ObjectComponents provides the following structs for use by applications that support linking and embedding.

TOcDragDrop	Holds information a container needs in order to receive an object dropped on its window
TOcFormatInfo	Holds information about one view's support for a particular Clipboard data format.
TOcMenuDescr	Holds information about a shared menu where the container and server merge their commands for in-place editing
TOcSaveLoad	Carries information an application needs to save or load a linked or embedded object
TOcToolbarInfo	Carries handles to a server's tool bars to be displayed in the container's window during in-place editing
TOcViewPaint	Carries information that tells a server how to repaint a linked or embedded object when the container invalidates part of the object's surface

ocrxxxx Constants

See Also

Header File
ocf/ocreg.h

Description
The ocreg.h header defines a number of constants used in constructing an application's registration tables. These constants all begin with *ocr*. They fall into several groups. Most of them are used with the REGFORMAT macro to describe the kinds of data transfers a document supports.

Group	Meaning
Aspect Constants	Data presentation modes (such as icon, content, or thumbnail)
Clipboard Constants	Clipboard data formats (such as text, bitmap, or link source)
Direction Constants	Data transfer directions (getting or setting)
Limit Constants	Maximum number of items that can be registered
Medium Constants	Data transfer mediums (such as disk file or Clipboard)
Object Status Constants	Aspect options (such as showing icon only or redrawing on resize)
Usage Constants	Support for multiple clients (single use or multiple use)
Verb Attributes constants	Verb option flags (never dirties and show on menu)
Verb Menu Flags	Verb display options (such as grayed, disabled, or menu bar break)

See Also
Registration Macros (OWL.HLP)

Compound File I/O Classes

TOcStorage
TOcStream

Compound File I/O Enumerated Types and Structs

STATSTG
STGC enum

appname Registration Key

See Also Registration Keys

Description

Registers a short name for the application, such as "QuattroPro 6.0" or "Scrapbook." This name appears in window title bars when the container shows what server an object comes from. The *appname* string should be localized.

A server needs an *appname* string for each document type. Because the string is usually the same for all document types, ObjectComponents lets you register the *appname* just once in the application registration table. It is added to document tables automatically. Including appname explicitly in document registration tables overrides the string set in the application table.

Containers need *appname* only if they support being a link source for other containers.

To register an *appname* string, use the REGDATA macro, passing *appname* as the first parameter and the string as the second parameter.

```
REGDATA(appname, "DrawPad Server")
```

The *appname* string is recorded in the system registry under the following key:

```
CLSID\<clsid>\AuxUserType\3 = <ApplicationName>
```

See Also
Localizing Symbol Names
REGDATA Macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)

aspectall Registration Key

See Also Registration Keys

Description

Registers option flags that affect all views of an object. The flags control how all views of the object are presented.

Linking and embedding servers can optionally register aspect status in their document registration tables. Aspect status does not apply to application registration tables, containers, or automation servers.

To register flags for all aspects, use the REGSTATUS macro, passing "all" as the first parameter and an ocrxxxx Object Status constant value as the second parameter.

```
REGSTATUS(all, ocrNoSpecialRendering)
```

See Also

aspectcontent Registration Key
aspectdocprint Registration Key
aspecticon Registration Key
aspectthumbnail Registration Key
ocrxxxx Object Status Constants
Registration Keys
Registration Macros (OWL.HLP)
REGSTATUS Macro (OWL.HLP)

aspectcontent Registration Key

See Also   Registration Keys

Description

Registers option flags for the content view of an object. The content view usually shows all the data in an object (or as much of the data as fits in the available space.) The option flags control how the content view is used.

Linking and embedding servers can optionally register aspect status in their document registration tables. Aspect status does not apply to application registration tables, containers, or automation servers.

To register flags for the content aspect, use the REGSTATUS macro, passing "content" as the first parameter and an ocrxxxx Object Status constant value as the second parameter.

```
REGSTATUS(content, ocrRecomposeOnResize)
```

See Also
aspectall Registration Key
aspectdocprint Registration Key
aspecticon Registration Key
aspectthumbnail Registration Key
ocrxxxx Object Status Constants
Registration Keys
Registration Macros (OWL.HLP)
REGSTATUS Macro (OWL.HLP)

aspectdocprint Registration Key

See Also Registration Keys

Description

Registers option flags that affect the printed document view of an object. The printed document view usually approximates how the object will appear if sent to the current printer. The option flags control how the docprint view is presented.

Linking and embedding servers can optionally register aspect status in their document registration tables. Aspect status does not apply to application registration tables, containers, or automation servers.

To register flags for the printed document aspect, use the REGSTATUS macro, passing "docprint" as the first parameter and an ocrxxxx Object Status constant value as the second parameter.

See Also
aspectall Registration Key
aspectcontent Registration Key
aspecticon Registration Key
aspectthumbnail Registration Key
ocrxxxx Object Status Constants
Registration Keys
Registration Macros (OWL.HLP)
REGSTATUS Macro (OWL.HLP)

aspecticon Registration Key

See Also   Registration Keys

Description

Registers option flags that affect the iconic view of an object. The icon view, rather than showing the object's contents, displays an icon that represents a particular kind of object. The option flags control how the icon view is presented.

Linking and embedding servers can optionally register aspect status in their document registration tables. Aspect status does not apply to application registration tables, containers, or automation servers.

To register flags for the icon aspect, use the REGSTATUS macro, passing "icon" as the first parameter and an ocrxxxx Object Status constant value as the second parameter.

```
REGSTATUS(icon, ocrOnlyIconic)
```

See Also
aspectall Registration Key
aspectcontent Registration Key
aspectdocprint Registration Key
aspectthumbnail Registration Key
ocrxxxx Object Status Constants
Registration Keys
Registration Macros (OWL.HLP)
REGSTATUS Macro (OWL.HLP)

aspectthumbnail Registration Key

See Also Registration Keys

Description

Registers option flags that affect the thumbnail view of an object. The thumbnail view usually shows a miniature representation of the object's contents. The flags control how the thumbnail view is presented.

Linking and embedding servers can optionally register aspect status in their document registration tables. Aspect status does not apply to application registration tables, containers, or automation servers.

To register flags for the thumbnail aspect, use the REGSTATUS macro, passing "thumbnail" as the first parameter and an ocrxxxx Object Status constant value as the second parameter.

See Also

aspectall Registration Key
aspectcontent Registration Key
aspectdocprint Registration Key
aspecticon Registration Key
ocrxxxx Object Status Constants
Registration Keys
Registration Macros (OWL.HLP)
REGSTATUS Macro (OWL.HLP)

AUTOARGS Macros

See Also Automation Proxy Macros

Header File
ocf/automacr.h

Description
An automation controller uses AUTOARGS to implement methods in its proxy objects. AUTOARGS macros list all the arguments that the controller passes to an automation command, identifying them by the dummy parameter names used in the function definition.

AUTOARGS macros are the second in three sets of macros used to implement methods in proxy objects. The first, AUTONAMES assigns names to any arguments that the controller wants to reference by name. The third set, AUTOCALL, tells whether the command is a method or a property and whether it returns a value.

Macro	Meaning
AUTOARGS0()	The automation command has no required arguments.
AUTOARGS1(a1)	The automation command requires argument *a1*.
AUTOARGS2(a1, a2)	The automation command requires arguments *a1* and *a2*.
AUTOARGS3(a1, a2, a3)	The automation command requires arguments *a1*, *a2*, and *a3*.

The automacr.h header defines macros that accept up to ten arguments (AUTOARGS10). To generate versions that accept more arguments, use the MACROGEN.EXE utility.

See Also
Automation Proxy Macros
AUTOCALL_xxxx Macros
AUTONAMES Macros

AUTOCALL_xxxx Macros

See Also   Automation Proxy Macros

Header File
ocf/automacr.h

Description
AUTOCALL is the third of three sets of macros that an automation controller uses to implement automation commands in proxy objects. The first two sets, AUTONAMES and AUTOARGS, describe the command's arguments. AUTOCALL macros tell whether the command represents a method or a property of the automated object and whether the command returns a value. Commands whose return value is itself an automated object must also be specially marked.

Macro	**Meaning**
AUTOCALL_METHOD_REF(prx)	The command is a method that returns a reference to an object. *prx* is an object derived from TAutoProxy and receives the return value.
AUTOCALL_METHOD_RET	The command is a method that returns a value.
AUTOCALL_METHOD_VOID	The command is a method that returns no value.
AUTOCALL_PROP_GET	The command returns the value of a property of the automated object.
AUTOCALL_PROP_REF(prx)	The command returns the value of a property and the value is itself an object. *prx* is an object derived from *TAutoProxy* and receives the return value.
AUTOCALL_PROP_SET(val)	The command assigns *val* to a property of the automated object.

See Also
Automation Proxy Macros
AUTONAMES macros
AUTOARGS macros
TAutoProxy::Invoke

AutoCallFlag enum

See Also

Header File
ocf/autodefs.h

Syntax
`enum AutoCallFlag`

Description
These flags identify types of class members. Automation servers use them in their automation declarations and automation controllers pass them when the invoke commands. Usually you don't have to use these constants directly because they are set for you by the automation declaration macros and the Automation Proxy Macros.

Constant	Meaning
acMethod	Member is a method.
acPropGet	Member gets the value of a property.
acPropSet	Member sets the value of a property.
acVoidRet	Member returns **void**.

The values are bit flags. Combine them with the bitwise OR operator (|).

See Also

Automation Declaration Macros

Automation Proxy Macros

TAutoProxy::Invoke

_AUTOCLASS Macro

Header File
ocf/autodefs.h

Description
_AUTOCLASS is the class modifier that ObjectComponents uses to declare the base classes and member objects it creates inside your classes for automation. As long as your own classes use the application's ambient memory model, you do not have to worry about _AUTOCLASS, which by default is defined as nothing. If, however, you declare your automation classes with a modifier that differs from the ambient class model, then the classes (such as TAutoBase) that ObjectComponents defines must be modified to match. To accomplish the modification, define _AUTOCLASS yourself. For example:

```
#define _AUTOCLASS __far
```

AUTODATA Macros

See Also Automation Declaration Macros

Header File
ocf/automacr.h

Description
An automation server uses AUTODATA macros in an automation declaration (after DECLARE_AUTOCLASS) to make data members of an automated class accessible through OLE.

Both forms take the same four parameters. *name* is the internal name that you assign to the data member. ObjectComponents uses the internal names to keep track of all the automated members. The only other place you use this name is in the subsequent automation definition (after DEFINE_AUTOCLASS).

member is the C++ name of the data member, the name you normally use in your source code.

In most cases, *type* should be a normal C++ data type, but if the data member is a string or an object then specify TAutoString or one of the TAutoObject classes instead. For more details, see Automation Data Types.

options is a place to insert a hook, code to be called each time the automation command is executed. Hooks can record, undo, or validate commands. *options* can be omitted, but a comma must follow the preceding argument anyway. For more details, see Automation Hook Macros.

Macro	Meaning
AUTODATA(name, member, type, options)	The command permits read and write access to a data member.
AUTODATARO(name, member, type, options)	The command permits read-only access to a data member.

See Also
Automation Data Types
Automation Declaration Macros
Automation Hook Macros

AutoDataType enum

See Also Automation Enumerated Types and Type Definitions

Header File
ocf/autodefs.h

Syntax
```
enum AutoDataType
```

Description
These flags identify automation data types. The types correspond to standard OLE 2 data types. The TAutoVal class uses the flags to guide its conversions to and from the VARIANT unions that OLE passes between programs.

Constant	Meaning
atVoid	**void**
atNull	SQL-style null
atShort	2-byte **signed int**
atLong	4-byte **signed int**
atFloat	4-byte real
atDouble	8-byte real
atCurrency	currency
atDatetime	datetime as **double**
atString	BSTR, string preceded by length
atObject	*IDispatch**
atError	SCODE
atBool	**true** = -1, **false** = 0
atVariant	VARIANT FAR*
atUnknown	*IUnknown**
atByte	**byte** or **unsigned char**
atTypeMask	Base type code without bit flags.
atOLE2Mask	Type code with bit flags.

The preceding flags are mutually exclusive. A value can belong to only one type. Any of the type flags can, however, be combined with the following bit flags.

Bit Flag	Meaning
atSafeArray	The value is a BASIC-type bounded array.
atByRef	The value is a reference to an object.
atEnum	The value is an enumeration of some type.
atAutoClass	The value is a TAutoClass object

See Also
TAutoVal

AUTODETACH Macro

See Also

Header File
ocf/automacr.h

Syntax
```
AUTODETACH
```

Description
An automation server uses this macro in its automation declaration (after DECLARE_AUTOCLASS) to ensure that whenever the automated object is destroyed OLE receives notification. Sending the object's obituary to OLE prevents crashes should a controller attempt to manipulate the nonexistent object. The obituary is necessary only if the logic of your program makes it possible for the automated object to be destroyed by non-automated means while still connected to the controller.

Deriving a class from TAutoBase serves exactly the same purpose. The advantage of AUTODETACH is that you can use it to automate classes you did not create and whose derivation you cannot control.

See Also

Automation Declaration Macros

TAutoBase

Telling OLE When the Object Goes Away

AUTOENUM Macros

See Also

Header File

ocf/automacr.h

Description

An automation server uses the AUTOENUM macros to expose enumerated values to automation controllers. For example, if the server wants the controller to pass actions into a DoThis command, the server might create an enumerated type containing values such as Play, Stop, and Rewind. To make these values available to the controller, the server must create an AUTOENUM table. In this example, the table consists of three AUTOENUM macros, one for each enumerated value.

```
DEFINE_AUTOENUM(TAction, TAutoShort);
 AUTOENUM("Play", Play)
 AUTOENUM("Stop", Stop)
 AUTOENUM("Rewind", Rewind);
END_AUTOENUM(TAction, TAutoShort)
```

Macro	Meaning
DEFINE_AUTOENUM(cls, type)	Begins an AUTOENUM table. *cls* is the name of the automated enumeration type (not the name of the C++ enumerated type). You invent this name. The only other place it appears is in the application's automation definition. *type* is the automation data type that describes what kind of values are being enumerated. For more information, see Automation Data Types.
AUTOENUM(name, val)	*name* is the public string that a controller uses to refer to one in a series of enumerated values. *val* is the internal value the server associates with *name*.
END_AUTOENUM(cls, type)	Ends an AUTOENUM table. *cls* and *type* are the same as for DEFINE_AUTOENUM.

See Also
Automation Data Types
Automation Definition Macros
Exposing Data for Enumeration

AUTOFLAG Macro

See Also Automation Declaration Macros

Header File
ocf/automacr.h

Syntax
AUTOFLAG(name, data, mask, options)

Description
An automation server uses AUTOFLAG in an automation declaration (after DECLARE_AUTOCLASS) to expose for automation a single bit from a set of bit flags.

name is an internal name that you assign to the bit. ObjectComponents uses the internal names to keep track of all the automated members. The only other place you use this name is in the subsequent automation definition (after DEFINE_AUTOCLASS.)

data is the C++ name of a data member that holds a set of bit flags.

mask is a value with one bit set marking the position of the exposed flag in *data*.

options is a place to insert a hook, code to be called each time the automation command is executed. Hooks can record, undo, or validate commands. See Automation Hook Macros for more details. *options* can be omitted, but a comma must follow the preceding argument anyway.

See Also
Automation Definition Macros
Automation Hook Macros

AUTOFUNC Macros

See Also Automation Declaration Macros

Header File
ocf/automacr.h

Description
An automation server uses AUTOFUNC macros in an automation declaration (after DECLARE_AUTOCLASS) to make member functions of an automated class accessible through OLE.

In every version of the macro the first parameter, *name*, is an internal name that you assign to the function. ObjectComponents uses the internal names to keep track of all the automated members. The only other place you use this name is in the subsequent automation definition (after DEFINE_AUTOCLASS).

func is the C++ name of the member function, the name you normally use in your source code.

ret is the type of data the function returns. *type1*, *t1*, *t2*, *t3*, and *t4* represent the data types of the parameters. In most cases, all these data types should be normal C++ data types, but if the data member is a string or an object then specify TAutoString or one of the TAutoObject classes instead. For more details, see Automation Data Types. Also, automated functions cannot return **const** values. Do not use const in a *ret* type.

options is a place to insert a hook, code to be called each time the automation command is executed. Hooks can record, undo, or validate commands. For more details, see Automation Hook Macros. *options* can be omitted, but a comma must follow the preceding argument anyway.

The automacr.h header defines versions of this macro that accept up to three arguments. To generate versions that accept more arguments, use the MACROGEN.EXE utility.

Macro	Meaning
AUTOFUNC0(name, func, ret, options)	The function takes no parameters and returns a value of type *ret*.
AUTOFUNC0V(name, func, options)	The function takes no parameters and returns **void**.
AUTOFUNC1(name, func, ret, type1, options)	The function takes one parameter of type *type1* and returns a value of type *ret*.
AUTOFUNC1V(name, func, type1, options)	The function takes one parameter of type *type1* and returns **void**.
AUTOFUNC2(name, func, ret, t1, t2, options)	The function takes two parameters of types *t1* and *t2*. It returns a value of type *ret*.
AUTOFUNC2V(name, func, t1, t2, options)	The function takes two parameters and returns **void**.
AUTOFUNC3(name, func, ret, t1, t2, t3, options)	The function takes three parameters and returns a value.
AUTOFUNC3V(name, func, t1 ,t2, t3, options)	The function takes three parameters and returns **void**.
AUTOFUNC4(name, func, ret, t1, t2, t3, t4, options)	The function takes four parameters and returns a value.
AUTOFUNC4V(name, func, t1, t2, t3, t4, options)	The function takes four parameters and returns **void**.

See Also
Automation Declaration Macros
Automation Hook Macros

AUTOINVOKE Macro

See Also   Automation Hook Macros

Header File
ocf/automacr.h

Syntax
`AUTOINVOKE(code)`

Description
An automation server uses AUTOINVOKE in an automation declaration macro to hook in user-defined code for ObjectComponents to execute every time the application receives a particular automation command. *code* is the expression or function call to execute on each command.

Create an AUTOINVOKE hook if you want to override the normal execution sequence.

See Also

Automation Declaration Macros

Automation Hook Macros

AUTOITERATOR Macros

See Also Automation Declaration Macros

Header File
ocf/automacr.h

Description
An iterator is an object used to enumerate a collection of objects. An iterator's methods let the caller step through a list of objects and examine each one in turn.

An automation server needs to create an iterator in any automated object that represents a collection of other objects. To create an iterator, the server adds one of the AUTOITERATOR macros to the class's automation definition (after DEFINE_AUTOCLASS). The iterator must also be exposed in the automation definition with the EXPOSE_ITERATOR macro.

Macro	Meaning
AUTOITERATOR(state, init, test, step, extract)	Implements a collection iterator within the automated class.
AUTOITERATOR_DECLARE(state)	Declares but does not implement a collection iterator within the automated class. Use this if your iterator's implementation is too complex for AUTOITERATOR.

The five arguments of AUTOITERATOR define the iteration algorithm for the collection class. Only one auto-iterator can exist within a class, so there is no need for a special internal name. The five arguments each represent a code fragment, and they follow the sequence of code in a **for** loop. As the examples show, because the iterator object is nested within the automated collection class, it can refer to members of the class.

Parameter	Example	Meaning
state	int Index	Declaration of state variables. This must be the same declaration previously given in AUTOITERATOR_DECLARE.
init	Index = 0	Statements (usually assignments) executed to initialize the loop.
test	Index < This->Total	Boolean expression tested each time through the loop.
step	Index++	Statements executed each time through the loop.
extract	(This->Array)[Index]	Expression that returns the successive objects in the collection.

Within the parameters, *This* (note the capital T) points to the enclosing collection object, not to the nested iterator object.

Commas cannot be used except inside parentheses. Semicolons can be used to separate multiple statements, but not to end a macro argument.

If you use AUTOITERATOR_DECLARE instead of AUTOITERATOR, then you must implement the state variables and these methods, corresponding to the steps described for AUTOITERATOR.

```
void Init();
bool Test();
void Step();
void Return(TAutoVal& v);
```

See Also
Automation Declaration Macros
EXPOSE_PROPxxxx macros
EXPOSE_QUIT Macro

AUTONAMES Macros

See Also   Automation Proxy Macros

Header File
ocf/automacr.h

Description
The AUTONAMES macros are the first in three sets of macros that an automation controller uses to implement methods in its proxy objects. AUTONAMES macros assign names to any arguments that the controller wants to reference by name. Named parameters have default values and are not required in a command. If a command has fifteen parameters and ten of them have names and default values, then the controller must always pass the five unnamed parameters and can choose to pass any subset of the remaining ten, identifying them by their names.

The second set of macros, AUTOARGS Macros, describe the data types of unnamed arguments that must always be passed in the command. The third set, AUTOCALL, tells whether the command is a method or a property and what it returns.

In the macros that follow, *id* is a numeric ID for a method, *fname* is a string naming a method, and *n1*, *n2*, *n3*, and *n4* are strings assigned as argument names. Most of the macros need the function name to identify the function, but if a function has no named arguments, then you can pass its identifying number instead.

Macro	Meaning
AUTONAMES0(id)	Function *id* has no named arguments.
AUTONAMES0(fname)	Function *fname* has no named arguments.
AUTONAMES1(fname, n1)	Function *fname* has one named argument, *n1*.
AUTONAMES2(fname, n1, n2)	Function *fname* has two named arguments, *n1* and *n2*.
AUTONAMES3(fname, n1, n2, n3)	Function *fname* has three named arguments, *n1*, *n2*, and *n3*.

The automacr.h header defines macros that accept up to ten arguments (AUTONAMES10). To generate versions that accept more arguments, use the MACROGEN.EXE utility.

See Also
Automation Proxy Macros
AUTOARGS Macros
AUTOCALL_xxxx Macros

AUTONOHOOK Macro

See Also Automation Hook Macros

Header File
ocf/automacr.h

Syntax
```
AUTONOHOOK
```

Description
An automation server uses AUTONOHOOK in an automation declaration macro to prevent anyone from hooking the command. Not even ObjectComponents can monitor the call.

AUTONOHOOK is for advanced uses only.

See Also

Automation Declaration Macros

Automation Hook Macros

AUTOPROP Macros

See Also Automation Declaration Macros

Header File
ocf/automacr.h

Description
An automation server uses AUTOPROP macros in its automation declaration (after AUTOITERATOR macros to the class's automation definition (after DEFINE_AUTOCLASS) to make properties of an automated class accessible to OLE. A property is data that cannot be read or written directly, only through a set of access functions (for example, *GetPosition* and *SetPosition*.)

A server can implement the access functions any way it likes. Because only the access functions are exposed, the property does not have to be a data member. In other words, *GetPosition* and *SetPosition* would not have to refer to a data member of type TPoint. They might query the system for the cursor position and return the answer.

The three AUTOPROP macros have similar parameters. *name* is an internal name you assign to the property. ObjectComponents uses the name to keep track of all the automated members. The only other place you use this internal name is in the corresponding automation definition.

get and *set* are the access functions. A read-only property has just a get function. A write-only property has just a set function.

type is the property's data type. This is usually a C++ data type, but string and object properties require special treatment. For more information, see Automation Data Types.

options is a place to insert a hook, code to be called each time the automation command is executed. Hooks can record, undo, or validate commands. *options* can be omitted, but a comma must follow the preceding argument anyway. For more details, see Automation Hook Macros.

Macro	Meaning
AUTOPROP(name, get, set, type, options)	The property can be read and written.
AUTOPROPRO(name, get, type, options)	The property can only be read, not changed.
AUTOPROPWO(name, set, type, options)	The property can be changed but not read (rare).

See Also
Automation Declaration Macros
TPoint (OWL.HLP)

AUTOPROXY Macro

See Also Automation Declaration Macros

Header File
ocf/automacr.h

Syntax
```
AUTOPROXY(name, proxyMember, options)
```

Description
An automation server uses AUTOPROXY in an automation declaration (after the DECLARE_AUTOCLASS macro) when automating a property whose value is an external automated object (one derived from TAutoProxy). AUTOPROXY is useful only in servers that also act as controllers and so create *TAutoProxy*-derived classes to send automation commands.

name is an internal name you assign to the property. ObjectComponents uses the name to keep track of all the automated members. The only other place you use this internal name is in the corresponding automation definition.

proxyMember is the *TAutoProxy*-derived class, the data type for the property.

options is a place to insert a hook, code to be called each time the automation command is executed. *options* can be omitted, but a comma must follow *proxyMember* anyway. For more details, see Automation Hook Macros.

See Also

Automation Declaration Macros

Automation Hook Macros

TAutoProxy

AUTORECORD Macro

See Also Automation Hook Macros

Header File
ocf/automacr.h

Syntax
```
AUTORECORD(code)
```

Description
An automation server uses AUTORECORD in an automation declaration to hook in user-defined code that creates a record of each call made to a particular automation command. *code* is the expression or function call to execute on each command. It should store whatever information the application would need to play back the same command later.

Recording is not supported in the current version of ObjectComponents.

See Also

Automation Declaration Macros

Automation Hook Macros

AUTOREPORT Macro

See Also Automation Hook Macros

Header File
ocf/automacr.h

Syntax
```
AUTOREPORT(code)
```

Description
An automation server uses AUTOREPORT in an automation declaration macro to hook in user-defined code that checks the error code from an automated member function. If *code* evaluates to 0, OLE assumes the command succeeded. If *code* evaluates to a nonzero value, then OLE throws an exception in the controller. Within the code expression, use *Val* to refer to the actual value returned.

See Also
Automation Declaration Macros
Automation Hook Macros

AutoSymFlag enum

See Also   Automation Enumerated Types and Type Definitions

Header File
ocf/autodefs.h

Syntax
```
enum AutoSymFlag
```

Description
These flags are used in the TAutoCommand class to describe attributes of an automation command. The flags tell whether the command is a method or a property, whether arguments are passed by value or by reference, and whether it should be visible in type information browsers.

Constant	Meaning
asAnyCommand	Any command: method, property access, object builder.
asOleType	Method or property exposed for OLE.
asMethod	Method.
asGet	Returns the value of a property.
asIterator	Iterator property; used to enumerate items in a collection.
asSet	Set property value.
asGetSet	Get or set a property value.
asBuild	Constructor command (not supported by OLE 2.01).
asFactory	For creating objects or determining class.
asClass	Extension to another class symbol table.
asArgument	Property that returns an object.
asBindable	Sends *OnChanged* notification.
asRequestEdit	Sends *OnRequest* edit before change.
asDisplayBind	User-display of bindable.
asDefaultBind	This property only is the default (redundant).
asHidden	Not visible to normal browsing.
asPersistent	Property is persistent.

See Also
TAutoCommand Public Constructor

AUTOSTAT Macros

See Also Automation Declaration Macros

Header File
ocf/automacr.h

Description
Use the AUTOSTAT in an automation declaration (after DEFINE_AUTOCLASS) to make static member functions and global functions accessible to OLE.

All versions of the AUTOSTAT macro have similar parameters. *name* is an internal name you assign to the function. ObjectComponents uses the name to keep track of all the automated members. The only other place you use this internal name is in the corresponding automation definition.

func is the name of the static or global function.

ret is the type of value the function returns.

type1, *t1*, *t2*, *t3*, and *t4* are the types of the function's arguments.

options is a place to insert a hook, code to be called each time the automation command is executed. Hooks can record, undo, or validate commands. See Automation Hook Macros for more details. *options* can be omitted, but a comma must follow the preceding argument anyway.

The return types and argument types are usually normal C++ data types, but string and object values require special treatment. For more information, see Automation Data Types.

Macro	**Meaning**
AUTOSTAT0(name, func, ret, options)	The static function *func* is assigned the symbol *name*. It takes no arguments and returns a value of type *ret*.
AUTOSTAT0V(name, func, options)	The static function *func* takes no arguments and returns no value.
AUTOSTAT1(name, func, ret, type1, options)	*func* takes one argument of type *type1* and returns a value of type *ret*.
AUTOSTAT1V(name, func, type1, options)	*func* takes one argument of type *type1* and returns no value.
AUTOSTAT2(name, func, ret, t1, t2, options)	*func* takes two arguments of types *t1* and *t2* and returns a value of type *ret*.
AUTOSTAT2V(name, func, t1,t2, options)	*func* takes two arguments and returns no value.
AUTOSTAT3(name, func, ret, t1,t2, t3, options)	*func* takes three arguments and returns a value.
AUTOSTAT3V(name, func, t1, t2, t3, options)	*func* takes three arguments and returns no value.
AUTOSTAT4(name, func, ret, t1, t2, t3, t4, options)	*func* takes four arguments and returns a value.
AUTOSTAT4V(name, func, t1, t2, t3, t4, options)	*func* takes four arguments and returns no value.

See Also
Automation Declaration Macros

AUTOTHIS Macro

See Also Automation Declaration Macros

Header File
ocf/automacr.h

Syntax
AUTOTHIS(name, type, options)

Description
An automation server uses the AUTOTHIS macro in its automation declaration (after DEFINE_AUTOCLASS) if it wants to expose the C++ object itself as a member of the automated OLE object.

name is an internal name you assign to the property. ObjectComponents uses the name to keep track of all the automated members. The only other place you use this internal name is in the corresponding automation definition.

type must be TAutoObject<*T*>, where **T** is the type of the automated class.

options is a place to insert a hook, code to be called each time a controller asks for this property. Hooks can record, undo, or validate commands. *options* can be omitted, but a comma must follow the preceding argument anyway. For more details, see Automation Hook Macros.

See Also
Automation Declaration Macros

AUTOUNDO Macro

See Also   Automation Hook Macros

Header File
ocf/automacr.h

Syntax
```
AUTOUNDO(code)
```

Description
An automation server uses AUTOUNDO in an automation declaration macro to hook in user-defined code that records whatever information the application needs to reverse the command later. Usually it adds information to a user-maintained undo stack. The information might include the parameters that execute the inverse of the original command, for example. To undo a series of actions, the program can pop commands off the undo stack and execute them. *code* is the expression or function that records information.

Undoing commands is not supported in the current version of ObjectComponents.

See Also

Automation Declaration Macros

Automation Hook Macros

AUTOVALIDATE Macro

See Also Automation Hook Macros

Header File
ocf/automacr.h

Syntax
```
AUTOVALIDATE(condition)
```

Description
An automation server uses AUTOVALIDATE in an automation declaration macro to hook in user-defined code that confirms the validity of received arguments before passing them on to be processed in a command. *condition* is an expression or function that evaluates to **true** if the arguments received are valid for the command and **false** if not. If the expression returns **false**, OLE throws an exception in the controller application.

See Also
Automation Declaration Macros
Automation Hook Macros

clsid Registration Key

See Also   Registration Keys

Description

Registers a globally unique identifier (GUID) for the application's class ID. A GUID is a 16-byte value and can be represented as a string.

A *clsid* GUID is required in every application registration table. You never need to specify any others. If others are needed for your documents, type library, automated classes, or debugging invocation, ObjectComponents automatically increments the low-order field of the first GUID to produce them. Be sure to allow for the full range of numbers your application actually uses when determining the next available GUID for another program.

There are several ways to acquire a *clsid*. One is to run the GUIDGEN tool in the OCTOOLS directory. Also, AppExpert automatically generates a GUID for any applications it creates that support OLE. Another way to get a GUID is to call the OLE API *CoCreateGuid*, as described in OLE.HLP. Finally, you can contact Microsoft to have a block of GUIDs assigned to you permanently.

Every application must have its own absolutely unique *clsid* string, so never use values pasted in from example programs.

To register a *clsid*, use the REGDATA macro with *clsid* as the first parameter and a GUID string as the second parameter.

```
REGDATA(clsid, "{CDE7F941-544B-101B-A9C1-04021C007002}")
```

See Also
REGDATA Macro (OWL.HLP)
Registration Macros (OWL.HLP)
Registration Keys

cmdline Registration Key

See Also Registration Keys

Description

Registers arguments OLE should place on the command line when it launches the server's executable file.

The *cmdline* key is valid in application registration tables, but is ignored for DLL servers. Any application can register it, but normally only automation servers have a use for it.

Automation servers can use the *cmdline* key to set up the *-Automation* switch. When the registrar object sees this switch, it overrides the application's registered *usage* setting and forces the program to run in single-use mode. This is useful in a server that supports linking and embedding as well as automation. As a linking and embedding server, it might support concurrent client applications with a single instance. When running as an automation server, however, most applications don't want concurrent client programs to control exactly the same instance of an object.

To register command-line options, use the REGDATA macro with *cmdline* as the first parameter and a string containing command-line arguments as the second parameter.

```
REGDATA(cmdline, "/automation")
```

See Also
REGDATA Macro (OWL.HLP)
Registration Macros (OWL.HLP)
Registration Keys

debugclsid Registration Key

See Also Registration Keys

Description

A GUID identifying the debugging version of a server. You should never register this key directly. It is always generated for you automatically if you register *debugprogid*.

This key is ignored in DLL servers.

See Also

debugprogid Registration Key

Registration Keys

debugdesc Registration Key

See Also   Registration Keys

Description

A string describing the debugging version of your program. When used in registering a document, this string appears in the Insert Object menu. When used in registering an application, it appears in object browsers. The string can contain up to 40 characters and can be localized.

The *debugdesc* key is required in application and document registration tables for any program that registers the *debugprogid* key. Otherwise it is irrelevant.

To register the *debugdesc* key, use the REGDATA macro, passing *debugdesc* as the first parameter and the descriptive string as the second parameter.

```
REGDATA(debugdesc, "My Application (debugging)")
```

This key is ignored in DLL servers.

See Also
debugclsid Registration Key
debugger Registration Key
debugprogid Registration Key
REGDATA macro (OWL.HLP)
Registration Macros (OWL.HLP)
Registration Keys

debugger Registration Key

See Also Registration Keys

Description

Registers the path and file name for loading your debugger.

The *debugger* key is valid in any registration table. It is required if you also register *debugprogid*. Otherwise it is irrelevant.

To register the *debugger* key, use the REGDATA macro, passing *debugger* as the first parameter and the command line string for the debugger application as the second parameter. When OLE invokes the debugger, it places the second parameter string on the command line ahead of the server's .EXE path. The debugger string can optionally contain a full path and debugger command line switches.

```
REGDATA(debugger, "TDW") // assumes TDW is somewhere on the path
```

This key is ignored in DLL servers.

See Also
debugclsid Registration Key
debugdesc Registration Key
debugprogid Registration Key
REGDATA macro (OWL.HLP)
Registration Macros (OWL.HLP)
Registration Keys

debugprogid Registration Key

See Also Registration Keys

A string identifying the debugging version of a program. Just as a *progid* string does, this string has two parts divided by a period. The first part is your programs name, and the second part is *.debug*.

Assigning a value to the *debugprogid* key causes ObjectComponents to create two sets of entries for the server in the registration database. When you choose Insert Object from the Edit menu, both entries appear in the list. Choosing the debugging entry causes ObjectComponents to invoke your debugger together with the server. Without the ability to register a duplicate debugging entry, it is difficult to debug the server when OLE invokes it.

ObjectComponents generates a *clsid* for the debugger entry automatically.

The *debugprogid* key is optional for application registration tables and irrelevant for document registration tables. If you register *debugprogid*, you also need to register *debugdesc* and *debugger*.

To register a *debugprogid*, use the REGDATA macro, passing *debugprogid* as the first parameter and the ID string as the second parameter.

```
REGDATA(debugprogid, "MyApp.Debug")
```

This key is ignored in DLL servers.

See Also
debugclsid Registration Key
debugdesc Registration Key
debugger Registration Key
progid Registration Key
REGDATA macro (OWL.HLP)
Registration Macros (OWL.HLP)
Registration Keys
Updating Your Document Registration Table

DECLARE_AUTOAGGREGATE Macro

See Also Automation Declaration Macros

Header File
ocf/automacr.h

Syntax
```
DECLARE_AUTOAGGREGATE(cls)
```

Description

DECLARE_AUTOAGGREGATE(cls) introduces the automation declaration for classes that are, inherit from, or delegate to Component Object Model (COM) objects.

The automation declaration declares automatable members of *cls*, a a user-defined C++ class. It belongs with the declaration of *cls*. Be sure to use DEFINE_AUTOAGGREGATE in the corresponding automation definition.

The DECLARE_AUTOCLASS macro also introduces automation declarations. Use it for classes that are not, and do not inherit from or delegate to, COM objects.

See Also
Automation Declaration Macros
DECLARE_AUTOCLASS macro
DEFINE_AUTOAGGREGATE macro

DECLARE_AUTOCLASS Macro

See Also Automation Declaration Macros

Header File
ocf/automacr.h

Syntax
```
DECLARE_AUTOCLASS(cls)
```

Description

DECLARE_AUTOCLASS(cls) introduces a block of macros that declare automatable members of the user-defined class *cls*.

An automation server uses DECLARE_AUTOCLASS to begin a block of macros that make automatable members of a user-defined C++ class accessible to OLE. *cls* is the name of the user's C++ class.

The block of declaration macros usually appears in the definition of the automatable C++ class. A corresponding block of automation definition macros must appear in the implementation of the automatable C++ class.

See Also
Automation Declaration Macros
Automation Definition Macros
DEFINE_AUTOCLASS macro
END_AUTOCLASS macro

DECLARE_COMBASES*n* Macros

See Also

Header File
ocf/oleutil.h

Description
Use the COMBASES macros to create C++ objects that conform to the OLE Component Object Model (COM). COM objects support OLE interfaces and let you derive classes that interact with OLE directly, not through ObjectComponents. These macros are meant for advanced users.

COM objects are used as base classes for other objects. The derived class must inherit from both your COM class and from the ObjectComponents TUnknown class. *TUnknown* implements the controlling *IUnknown* interface for your object.

To create a COM class:

1. Precede the declaration of the class with one of the DECLARE_COMBASES macros. Which you choose depends on how many interfaces (besides IUnknown*)* the COM class supports.
2. Precede the implementation of your COM class with the corresponding DEFINE_COMBASES macro.
3. Derive your final class multiply from *TUnknown* and your new COM class (in that order).

The first macro argument is *name.* It is the name of the COM class you are creating. *i1*, *i2*, *i3*, and *i4* are the names of the interfaces your COM class supports.

Macro	Meaning
DECLARE_COMBASES1(name, i1)	Declare a COM class *name* that inherits from the *i1* interface class.
DECLARE_COMBASES2(name, i1, i2)	Declare a COM class *name* that inherits from the *i1* and *i2* interface classes.
DECLARE_COMBASES3(name, i1, i2, i3)	Declare a COM class *name* that inherits from the *i1. i2*, and *i3* interface classes.
DECLARE_COMBASES4(name, i1, i2, i3, i4)	Declare a COM class *name* that inherits from the *i1. i2*, *i3*, and *i4* interface classes.

See Also

DEFINE_COMBASES*n* Macros

TUnknown

DEFINE_AUTOAGGREGATE Macro

See Also Automation Definition Macros

Header File
ocf/automacr.h

Syntax

```
DEFINE_AUTOAGGREGATE(cls, AggregatorFunction)
```

Description

An automation server uses DEFINE_AUTOAGGREGATE to begin a block of macros that define automatable members of *cls*, a user-defined C++ class. The block ends with the END_AUTOAGGREGATE macro.

The DEFINE_AUTOCLASS macro does the same thing but without the extra *AggregatorFunction* parameter. Use DEFINE_AUTOAGGREGATE when the C++ class you are automating is, inherits from, or delegates to a Component Object Model (COM) object. The *AggregatorFunction* parameter points to the aggregating function for reaching the COM object. For example, if *Aggregate* is the name of the COM object, the aggregater function might be any of these expressions:

```
Aggregate           // automated C++ object is the COM object
OcApp->Aggregate    // automated C++ object delegates to COM object
MyBase::Aggregate   // automated C++ object inherits from COM object
```

When the automation definition uses DEFINE_AUTOAGGREGATE, be sure to use DECLARE_AUTOAGGREGATE in the corresponding automation declaration.

See Also
Automation Definition Macros
DECLARE_AUTOAGGREGATE macro
DEFINE_AUTOCLASS macro
END_AUTOAGGREGATE macro

DEFINE_AUTO CLASS Macro

See Also Automation Definition Macros

Header File
ocf/automacr.h

Syntax
```
DEFINE_AUTOCLASS(cls)
```

Description
Introduces a block of macros in an automation server. Each macro in the block defines an automatable member of *cls*, a user-defined C++ class. The block ends with the END_AUTOCLASS macro.

The block of definition macros appears in the implementation of the automatable C++ class. A corresponding block of automation declaration macros must appear in the definition of the same class.

See Also

Automation Declaration Macros

Automation Definition Macros

END_AUTOCLASS Macro

DEFINE_COMBASES*n* Macros

See Also

Header File
ocf/oleutil.h

Description
Implements the *IUnknown* interface for each of the OLE interfaces that your COM object supports.

Use the COMBASE macros to create C++ objects that conform to the OLE Component Object Model (COM). COM objects support OLE interfaces and let you derive classes that interact with OLE directly, not through ObjectComponents. These macros are meant for advanced users.

Automated objects can delegate to COM objects using the DEFINE_AUTOAGGREGATE and the DECLARE_AUTOAGGREGATEmacro.

To create a COM class:

1. Precede the declaration of the class with one of the DECLARE_COMBASES macros. Which you choose depends on how many interfaces (besides *IUnknown*) the COM class supports.
2. Precede the implementation of your COM class with the corresponding DEFINE_COMBASES macro.
3. Derive your final class multiply from your new COM class and from TUnknown.

COM objects can delegate to other COM objects using another set of related macros also defined in oleutil.h. For more information, look in the header file for the DEFINE_QI_xxxx macros.

Macro	**Meaning**
DEFINE_COMBASES1(name, i1)	Define *IUnknown* for the *i1* interface in the COM class *name*.
DEFINE_COMBASES2(name, i1, i2)	Define *IUnknown* for the *i1* and *i2* interfaces in the COM class *name*.
DEFINE_COMBASES3(name, i1, i2, i3)	Define *IUnknown* for the *i1*, *i2*, and *i3* interfaces in the COM class *name*.
DEFINE_COMBASES4(name, i1, i2, i3, i4)	Define *IUnknown* for the *i1*, *i2*, *i3*, and *i4* interfaces in the COM class *name*.

See Also
DECLARE_COMBASES*n* Macros

description Registration Key

See Also Registration Keys

Description

Registers a long descriptive name, up to 40 characters, meant for the user to see. The string appears in the Insert Object dialog box. It should describe the program and its object together--for example, "Quattro Pro 6.0 Notebook." This string is often just a concatenation of the *appname* and *menuname* strings.

The value of *description* should be localized.

A description string is required in document registration tables, and in the application table of automation servers. To register a description string, use the REGDATA macro, passing *description* as the first parameter and the descriptive string as the second parameter.

```
REGDATA(description, "DrawPad (Step15--Server) Drawing")
```

The *description* string is recorded in the system registry under the following keys:

```
CLSID\<clsid> = <MainUserTypeName>
<progid> = <MainUserTypeName>
```

See Also
Localizing Symbol Names
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)
version Registration Key

directory Registration Key

See Also Registration Keys

Description

Registers the default directory for document files. The document template class refers to this path when it invokes a File Open common dialog box. The directory path is not used by OLE.

The directory key is valid in any document registration table. It is always optional.

To register a directory, use the REGDATA macro, passing *directory* as the first parameter and a path name as the second parameter.

```
REGDATA(directory, "C:\\temp")
```

See Also
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)

docfilter Registration Key

See Also   Registration Keys

Description

Registers a file specification for listing files created by the application. This information is used by the document template class when it creates a File Open common dialog box. It is not used by OLE.

docfilter is valid in any document registration table. It is required unless the corresponding docflags key includes the *dtHidden* flag. If you register a document filter, you might also want to register the extension key.

To register a document filter, use the REGDATA macro, passing *docflags* as the first parameter and a filter string as the second parameter.

```
REGDATA(docfilter, "*.txt")
```

See Also
docflags Registration Key
dtxxxx Document Template Constants (OWL.HLP)
extension Registration Key
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)

docflags Registration Key

See Also		Registration Keys

Description

Registers document view option flags for the application's documents. This information is used by the document template class, not by OLE. The document template uses the flags to control its display of the File Open common dialog box. For a list of all the flags, see the description of *dtxxxx* Document View Constants.

The *docflags* key is valid in any document registration table. It is always optional.

To register document flags, use the REGDOCFLAGS macro.

```
REGDOCFLAGS(dtAutoOpen | dtAutoDelete | dtUpdateDir | dtCreatePrompt)
```

See Also
docfilter Registration Key
dtxxxxDocumentTemplateConstants (OWL.HLP)
REGDOCFLAGS Macro (OWL.HLP)
Registration Macros (OWL.HLP)
Registration Keys

DynamicCast function

See Also   Global Utility Functions

Header file
ocf/autodefs.h

Syntax
```
const void far* DynamicCast(const void far* obj, const typeinfo& src, const
  typeinfo& dst);
```

Description
Attempts to convert a pointer from one type to another. The attempt succeeds only if the old type and the new type are related through inheritance.

obj is the pointer you want to cast to a new type. *src* is type information about the source object. *dst* is information about the destination type.

If the conversion succeeds, *DynamicCast* returns a pointer to the new type. If it fails, the return is zero.

You can generate the *src* and *dst* parameters with the *typeid* parameter. (ObjectComponents requires the use of RTTI.)

See Also
MostDerived function
typeid (BCW.HLP)
typeinfo Class (CLASSLIB.HLP)

END_AUTOAGGREGATE Macro

See Also Automation Definition Macros

Header File
ocf/automacr.h

Syntax
```
END_AUTOAGGREGATE(cls, flags, name, doc, help)
```

Description

Terminates a block of macros that an automation server uses to define automatable methods in *cls*, a user-defined C++ class. The definition block begins with DEFINE_AUTOAGGREGATE Macro.

The related DEFINE_AUTOCLASS and END_AUTOCLASS macros also mark an automation definition block. Use the aggregation macros when *cls* is, inherits from, or delegates to a Component Object Model (COM) object.

name is the string that automation controllers use to identify objects of type *cls*. If the user asks OLE about the object name, the system returns the string in *doc*. If a .HLP file is registered for the object, then the context ID in *help* points to a screen that describes the object.

flags combines tf*xxxx* bit flags to provide type information for the automated class. The flags are documented with the END_AUTOCLASS macro.

See Also
Automation Definition Macros
DEFINE_AUTOAGGREGATE Macro
END_AUTOCLASS Macro

END_AUTOCLASS Macro

See Also Automation Definition Macros

Header File
ocf/automacr.h

Syntax
```
END_AUTOCLASS(cls, flags, name, doc, help)
```

Description
Terminates the block of macros an automation server uses to define automatable methods in *cls*, a user-defined C++ class. The definition block begins with DEFINE_AUTOCLASS.

name is the string that automation controllers use to identify objects of type *cls*. If the user asks OLE about the object name, the system returns the string in *doc*. If a .HLP file is registered for the object, then the context ID in *help* points to a screen that describes the object.

flags combines tf*xxxx* bit flags to provide type information for the automated class. Choose the flags that describe your class. The tf*xxxx* constants correspond to symbols defined by OLE and documented in OLE.HLP. The COCLASS, an OLE construct, holds type information for an OLE component, which might contain several smaller OLE objects. The COCLASS holds type information for all interfaces exposed at the component level.

Constant	Meaning
flags set on the COCLASS object	
tfAppObject	TYPEFLAG_FAPPOBJECT, set on COCLASS
tfCanCreate	TYPEFLAG_FCANCREATE, set on COCLASS
tfLicensed	TYPEFLAG_FLICENSED, set on COCLASS
tfPredeclared	TYPEFLAG_FPREDECLID, set on COCLASS
tfControl	TYPEFLAG_FCONTROL, set on COCLASS
tfCoClassXfer	tfAppObject | tfCanCreate | tfLicensed | tfControl | tfPredeclared
flags set on the COCLASS interfaces	
tfDefault	IMPLTYPEFLAG_FDEFAULT << 12
tfEventSource	IMPLTYPEFLAG_FSOURCE << 12
tfRestricted	IMPLTYPEFLAG_FRESTRICTED << 12
tfImplFlagXfer	tfDefault | tfEventSource | tfRestricted
flags set on individual automated classes	
tfHidden	TYPEFLAG_FHIDDEN
tfNonextensible	TYPEFLAG_FNONEXTENSIBLE
tfAutoClassMask	tfHidden | tfNonextensible
flags defined by OLE, but not applicable to IDispatch interfaces	
tfDual	TYPEFLAG_FDUAL
tfAutomation	TYPEFLAG_FOLEAUTOMATION
default flags for a class not exposed as part of the COCLASS	
tfNormal	automated classes not at application level

See Also

Automation Definition Macros

DEFINE_AUTOCLASS Macros

EXPOSE_APPLICATION Macro

See Also Automation Definition Macros

Header File
ocf/automacr.h

Syntax

EXPOSE_APPLICATION(cls, extName, doc, help)

Description
An automation server uses EXPOSE_APPLICATION in its automation definition (after DEFINE_AUTOCLASS macro) if it chooses to expose the application itself as a member of its automated object. OLE conventions suggest that each automation object should have this member.

cls is the class name of the application.

extName is the external, public name you assign to this member. Automation controllers use this string to refer to the application member. The string can be localized.

doc is a string that describes this member to the user. An automation controller can ask OLE for this string if the user requests help.

help is a context ID for an .HLP file. This parameter, which can be omitted, is useful only if you register an .HLP file to document your server. If you do, then when the user asks the controller for help, the controller passes this context ID to the Help system to display a screen describing the object member.

See also

Automation Definition Macros

Localizing Symbol Names

EXPOSE_DELEGATE Macro

See Also Automation Definition Macros

Header File
ocf/automacr.h

Syntax

EXPOSE_DELEGATE(cls, extName, locator)

Description

An automation server uses EXPOSE_DELEGATE in its automation definition (after DEFINE_AUTOCLASS macro) in order to combine two unrelated C++ classes into a single OLE automation object. In effect, the application's primary automated class delegates some tasks to another automated class. This macro tells ObjectComponents to search both classes to determine what commands the automated OLE object can perform.

cls is the name of the auxiliary class, which must also be automated. In other words, it must contain its own automation declaration and definition.

extName is an external, public name that an OLE controller uses to refer to this member of the object.

locator is a function that returns a pointer to an auxiliary object. In order to call members of that class, ObjectComponents needs a pointer to an object of that type. The conversion function should follow this prototype:

```
auxclass *locator( autoclass *this );
```

where *auxclass* is the name of the auxiliary class and *autoclass* is the name of the primary automated class. *locator* in effect converts a *this* pointer to a *that* pointer.

See Also
Automation Definition Macros
EXPOSE_INHERIT Macro
Localizing Symbol Names

EXPOSE_INHERIT Macro

See Also Automation Definition Macros

Header File
ocf/automacr.h

Syntax

```
EXPOSE_INHERIT(cls, extName);
```

Description
An automation server uses EXPOSE_INHERIT in its automation definition (after DEFINE_AUTOCLASS) in order to combine two related C++ classes into a single OLE automation object. In effect, the application's primary automated class delegates some tasks to its base class. This macro tells ObjectComponents to search both classes to determine what commands the automated OLE object can perform.

cls is the name of the base class, which must also be automated. In other words, it must contain its own automation declaration and definition.

extName is an external, public name that an OLE controller uses to refer to this member of the object. It can be localized.

See Also

Automation Definition Macros

EXPOSE_DELEGATE Macro

Localizing Symbol Names

EXPOSE_ITERATOR Macro

See Also Automation Definition Macros

Header File
ocf/automacr.h

Syntax

```
EXPOSE_ITERATOR(retType, doc, help);
```

Description
An automation server uses EXPOSE_ITERATOR in its automation definition (after DEFINE_AUTOCLASS) in order to expose an iterator object to enumerate objects in a collection. An iterator is useful only when the automated object itself represents a collection of other objects. The controller uses methods of the nested iterator object to retrieve and examine successive objects in a list.

retType is an automated data type that describes the type of the objects in the collection. For example, if the collection is an array of short integers, then *retType* should be TAutoShort. For more information, see Automation Data Types.

doc is a string that describes the iterator to the user. An automation controller can ask OLE for this string if the user requests help.

help is a context ID for an .HLP file. This parameter, which can be omitted, is useful only if you register an .HLP file to document your server. If you do, then when the user asks the controller for help, the controller passes this context ID to the Help system to display a screen describing the iterator.

See Also
Automation Data Types
Automation Definition Macros
AUTOITERATOR Macros

EXPOSE_METHOD Macros

See Also Automation Definition Macros

Header File
ocf/automacr.h

Syntax

```
EXPOSE_METHOD(intName, retType, extName, doc, help);
EXPOSE_METHOD_ID(id, intName, retType, extName, doc, help);
```

Description

An automation server uses EXPOSE_METHOD macros in its automation definition (after DEFINE_AUTOCLASS) to expose a member function of an object to OLE for automation.

intName is an internal name that you assign to identify the method. ObjectComponents uses the internal name to keep track of all the automated members. This name must match the name assigned to the method with the AUTOFUNC macro in the automation declaration.

retType is a data type that describes the type of value the method returns. For example, if the method returns a long integer, then *retType* should be TAutoLong. For more information, see Automation Data Types.

extName is the external, public name that a controller uses to specify this method. The string can be localized.

doc is a string that describes this method to the user. An automation controller can ask OLE for this string if the user requests help. This string can also be localized.

help is a context ID for an .HLP file. This parameter, which can be omitted, is useful only if you register an .HLP file to document your server. If you do, then when the user asks the controller for help, the controller passes this context ID to the Help system to display a screen describing the method.

id is a dispatch identifier that you can choose to assign explicitly by using one of the EXPOSE_METHOD_ID macros. The dispatch ID is what OLE passes to identify commands requested by a controller. The OLE system header files define several standard dispatch ID values. For example, -5 is the default evaluation method. Standard dispatch IDs are always negative numbers. By default, dispatch IDs are assigned low positive numbers incremented from 1. If you want to specify explicit dispatch IDs for your applications, choose high positive values in order not to collide with the low positive numbers ObjectComponents assigns to exposed members without explicit IDs.

0 is the ID of the default method or property. ObjectComponents never automatically assigns 0 as a dispatch ID. To have a default method, you need to assign 0 yourself.

An EXPOSE_METHOD or EXPOSE_METHOD_ID macro must always be followed immediately by one macro for each of the method's arguments.

Type of Argument	Declaration Macro
Required	REQUIRED_ARG Macro
Optional	OPTIONAL_ARG Macro

See Also
Automation Data Types
Automation Definition Macros
AUTOFUNC Macros
Localizing Symbol Names
OPTIONAL_ARG Macro
REQUIRED_ARG Macro

EXPOSE_PROPxxxx Macros

See Also Automation Definition Macros

Header File
ocf/automacr.h

Description
An automation server uses EXPOSE_PROPxxxx macros in its automation definition (after DEFINE_AUTOCLASS) to expose properties of an object to OLE for automation. A property is data that can be read or written only through a set of access functions (for example, *GetPosition* and *SetPosition*.)

Macro	Meaning
EXPOSE_PROPRW(intName, type, extName, doc, help)	The property can be read and written.
EXPOSE_PROPRW_ID(id, intName, type, extName, doc, help)	The property can be read and written. Its dispatch ID is *id*.
EXPOSE_PROPRO(intName, type, extName, doc, help)	The property is read-only.
EXPOSE_PROPRO_ID(id, intName, type, extName, doc, help)	The property is read-only. Its dispatch ID is *id*.
EXPOSE_PROPWO(intName, type, extName, doc, help)	The property is write-only and cannot be read (rarely used).

intName is an internal name that you assign to identify the property. ObjectComponents uses the internal name to keep track of all the automated members. This name must match the name assigned to the method with the AUTOPROP macro in the automation declaration.

type is a data type that describes the type of value the property holds. For example, if the property is a string, then *type* should be TAutoString. For more information, see Automation Data Types.

extName is the public name that a controller uses to refer to this property. The string can be localized.

doc is a string that describes this property to the user. An automation controller can ask OLE for this string if the user requests help. This string can also be localized.

help is a context ID for an .HLP file. This parameter, which can be omitted, is useful only if you register an .HLP file to document your server. If you do, then when the user asks the controller for help, the controller passes this context ID to the Help system to display a screen describing the property.

id is a dispatch identifier that you can choose to assign explicitly by using one of the EXPOSE_xxxx_ID macros. The dispatch ID is what OLE passes to identify commands requested by a controller. The OLE system header files define several standard dispatch ID values. For example, -5 is the default evaluation method. Standard dispatch IDs are always negative numbers. By default, dispatch IDs are assigned low positive numbers incremented from 1. If you want to specify explicit dispatch IDs for your applications, choose high positive values in order not to collide with the low positive numbers ObjectComponents assigns to exposed members without explicit IDs.

0 is the ID of the default method or property. ObjectComponents never automatically assigns 0 as a dispatch ID. To have a default property, you need to assign 0 yourself.

Note: A potential ambiguity arises in the *type* parameter when a read/write property contains an automated object. In the property setting function, *type* describes an argument, but in the property getting function, it describes a return value. According to the table of Automation Data Types, the value of the *type* parameter should be TAutoObject<> for a function argument and TAutoObjectDelete<> for a return value. In such cases, specify *TAutoObjectDelete*<> for *type*. ObjectComponents ignores the deletion information for the argument of the property setting function.

See Also

Automation Data Types

Automation Definition Macros

AUTOPROP Macros

Localizing Symbol Names

EXPOSE_QUIT Macro

See Also Automation Definition Macros

Header File
ocf/automacr.h

Syntax
```
EXPOSE_QUIT(extName, docString, helpContext)
```

Description
An automation server that exposes the application itself as an automated object uses the EXPOSE_QUIT macro to make a safe shutdown method available to the controller. The shutdown method implemented by this macro checks whether the application was originally invoked by OLE for automation. If so, it unregisters the active object and shuts down the server. If the user invoked the server before the controller connected to it, however, then the shutdown method does nothing because the application should continue to run.

Every automated application should include EXPOSE_QUIT in the application object's automation definition (after DEFINE_AUTOCLASS). EXPOSE_QUIT is not needed in the automation definition of other automated objects the server might create.

extName is the external, public name that a controller uses to call this method. The string can be localized.

doc is a string that describes the shutdown method to the user. An automation controller can ask OLE for this string if the user requests help. This string can also be localized.

help is a context ID for an .HLP file. This parameter, which can be omitted, is useful only if you register an .HLP file to document your server. If you do, then when the user asks the controller for help, the controller passes this context ID to the Help system to display a screen describing the shutdown method.

See Also

Automation Definition Macros

Localizing Symbol Names

extension Registration Key

See Also Registration Keys

Description

A file-name extension. This extension becomes the default extension assigned to file names in the File Open common dialog box. It is also recorded in the system registration database so that OLE can find the right server for a file based on the file's extension.

extension is valid in document registration tables of servers that support linking and embedding. It is always optional. If you register an extension, you might also want to register a docfilter.

To register a file extension, use the REGDATA macro, passing *extension* as the first parameter and the extension string as the second parameter. In 16-bit Windows, the extension is limited to three characters.

```
REGDATA(extension, "TXT")
```

See Also
docfilter Registration Key
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)

filefmt Registration Key

See Also Registration Keys

Description

Registers a name for a servers default file format. This string appears in dialog boxes where the user selects file types.

filefmt is valid in document registration tables for servers that support linking and embedding. It is always optional.

To register a file format, use the REGDATA macro, passing *filefmt* as the first parameter and the name string as the second parameter.

See Also
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)

FILETIME struct

See Also

Header File
ocf/ocstorag.h

Description
This structure holds a 64-bit value that indicates a date and time. One field holds the low half of the value and the other holds the high half. The complete value represents the number of 100-nanosecond intervals since January 1, 1601.

OLE uses the FILETIME structure to record the creation and access times for file elements.

To manipulate FILETIME values, call any of the following OLE commands (described in OLE.HLP): *CoDosDateTimeToFileTime*, *CoFileTimeToDosDateTime*, and *CoFileTimeNow*.

Public Data Members
```
uint32 dwLowDateTime;
uint32 dwHighDateTime;
```

See Also

STATSTG struct

TOcStorage::SetElementTimes

TOcStorage::SetTimes

FILETIME::dwLowDateTime

FILETIME

Syntax

```
uint32 dwLowDateTime;
```

Description

Contains the low-order half of a 64-bit time counter value.

FILETIME::dwHighDateTime

FILETIME

Syntax

```
uint32 dwHighDateTime;
```

Description

Contains the high-order half of a 64-bit time counter value.

format*n* Registration Key

See Also Registration Keys

Description

Registers a Clipboard format the application supports. An application registers the formats that it can put on or take from the Clipboard. A server can register different sets of formats for different document types.

Clipboard format keys are valid in any document registration table. Any application that supports linking and embedding should register at least some Clipboard formats. ObjectComponents supports up to eight formats using the keys *format0* through *format7.* The ocrFormatLimit constant, defined in ocf/ocreg.h, represents the maximum number of formats allowed (8).

To register a Clipboard format, use the REGFORMAT macro. The first parameter assigns a priority to the format. Give your preferred format highest priority. Programs that support OLE usually prefer to export their data as OLE objects, and so they make *ocrEmbedSource* priority 0. The second parameter identifies a particular data format. For explanations of the other parameters, see the description of the REGFORMAT macro.

```
REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict, ocrGet)
```

See Also
ocrxxxx Aspect Constants
ocrxxxx Clipboard Constants
ocrxxxx Direction Constants
ocrxxxx Limit Constants
ocrxxxx Medium Constants
REGFORMAT Macro (OWL.HLP)
Registration Keys

handler Registration Key

See Also Registration Keys

Description

A full path pointing to a library that can draw objects created by the server.

A path for the library that OLE can call to draw objects without having to launch the server as a separate process. By default this value is OLE2.DLL. OLE itself can render cached formats such as metafiles and bitmaps.

The *handler* key is valid in document registration tables for servers that support linking and embedding. It is always optional, but if you omit it OLE cannot use your handler library.

To register a handler, use the REGDATA macro passing *handler* as the first parameter and the path of the handler DLL in the second parameter. If the path does not begin with a drive or root directory, ObjectComponents determines the full path by starting at the place where the server itself is installed. For example, if the server is at C:\MYDIR and the handler path is HELPERS\MYHANDLR, then the full path is assumed to be C:\MYDIR\HELPERS\MYHANDLR.DLL.

See Also
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)

helpdir Registration Key

See Also Registration Keys

Description

Directory where online Help for the type library resides. (The name of the Help file is registered separately with the typehelp key.)

This key matters only in the application registration table of an automation controller. It is always optional. If you register a type library Help file without registering a Help directory, ObjectComponents automatically assumes the same directory registered for *path*.

To register a Help directory, use the REGDATA macro passing *helpdir* as the first parameter and the path string as the second parameter.

If the path does not begin with a drive or root directory, ObjectComponents determines the full path by starting at the place where the server itself is installed. For example, if the server is at C:\MYDIR and the Help path is HELP, then the Help file is assumed to be at C:\MYDIR\HELP.

See Also
path Registration Key
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)
typehelp Registration Key

HR_xxxx Return Constants

General OLE Classes, Macros, and Type Definitions

Header File
ocf/ocdefs.h

Description
OLE system calls return HRESULT values that sometimes encode detailed information about the result of the call. ObjectComponents sometimes passes HRESULT values back to you. These macros simplify the task of testing for some common results. Each represents a possible return value. For a complete listing of HRESULT return values, see the scode.h header file.

Constant	Meaning
HR_ABORT	The operation was aborted.
HR_FAIL	An unspecified error occurred.
HR_FALSE	An action did not complete in the usual way but no error occurred. For example, an enumeration reached the end of its list.
HR_HANDLE	A handle is invalid.
HR_INVALIDARG	One or more arguments are invalid.
HR_NOERROR	No error occurred.
HR_NOINTERFACE	The requested interface is not supported.
HR_NOTIMPL	The requested service is not implemented.
HR_OK	Same as HR_NOERROR.
HR_OUTOFMEMORY	Not enough memory is available to complete the operation.
HR_POINTER	A pointer is invalid.

_ICLASS Macro

General OLE Classes, Macros, and Type Definitions

Header File
ocf/oleutil.h

Description
Modifies the declaration of an interface class, one that defines or implements an interface for OLE or for the BOCOLE support library.

iconindex Registration Key

See Also Registration Keys

Description

A zero-based index telling which of the icons in the server's resources represents the type of objects the server produces.

Use *iconindex* in the document registration tables of a linking and embedding server. It is always optional.

To register an icon index, use the REGICON macro, passing the index value as the parameter.

```
REGICON(1)
```

See Also
REGICON Macro (OWL.HLP)
Registration Macros (OWL.HLP)
Registration Keys

_IFUNC Macro

General OLE Classes, Macros, and Type Definitions

Header File
ocf/oleutil.h

Description
Modifies the declaration of an OLE function.

The _IFUNC macro controls function calling conventions and export declarations. Placing these macros in a keyword allows the compiler to choose the right combination of modifiers for a particular platform.

ObjectComponents uses the macro to declare OLE and BOCOLE functions as well as member functions that wrap direct OLE and BOCOLE calls.

_IFUNC serves the same purpose in Borland headers that the STDMETHODCALLTYPE serves in OLE system headers.

insertable Registration Key

See Also		Registration Keys

Description

Indicates the application is a server and allows its document to be linked or embedded in other applications. Registering this key makes the document type show up in dialog boxes listing objects that can be inserted. The value assigned to this key is ignored.

insertable is valid in the document registration tables of a linking and embedding server. An application must register *insertable* for at least one document type in order to be a linking and embedding server. A server need not make all its document types insertable, however. (Be sure to register a *progid* for insertable document types, as well.)

To register the *insertable* key, use the REGDATA macro, passing *insertable* as the first parameter and 0 as the second parameter.

```
REGDATA(insertable, 0)
```

See Also
progid Registration Key
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)

language Registration Key

See Also		Registration Keys

Description

Overrides the locale ID currently in effect. By default, the *language* key takes its value from the system's default language setting for the current user. Registering a language setting directs ObjectComponents to choose a particular language for registration strings you have localized.

During automation this value is reset internally at the request of the automation controller.

See Also

Localizing Symbol Names

Registration Keys

menuname Registration Key

See Also		Registration Keys

Description

A short name for the server's objects, such as *Chart*, *Notebook*, or *Drawing*. The name appears on the Edit menu in container programs. For consistency in the user interface, the suggested maximum length is 15 characters.

menuname is required in the document registration tables of a linking and embedding server. In other places it is irrelevant. The *menuname* string can be localized.

To register the *menuname* key, use the REGDATA macro, passing *menuname* as the first parameter and a name string as the second parameter.

```
REGDATA(menuname, "Drawing")
```

The *menuname* string is recorded in the system registry under the following key:

```
CLSID\<clsid>\AuxUserType\2 = <ShortName>
```

See Also
Localizing Symbol Names
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)

MostDerived function

See Also Global Utility Functions

Header file
ocf/autodefs.h

Syntax
```
const void far* MostDerived(const void far* obj, const typeinfo& src);
```

Description
Returns a pointer to the most derived class type that fits the given object. This is useful when dealing with polymorphic objects. Use it to obtain a consistent pointer to an object, regardless of the type of pointer used to reach the object.

obj is the pointer whose most derived type you want to determine. *src* holds type information about the *obj* object. The return value points to the most derived class that can be made out of *obj*. If *obj* is already the object's most derived type, then the return value is *obj*.

Use *typeid* to generate the *src* parameter.

See Also
DynamicCast function
typeid (BCW.HLP)
typeinfo class (CLASSLIB.HLP)

ObjectPtr typedef

Automation Enumerated Types and Type Definitions

Header file
ocf/autodefs.h

Syntax
```
typedef void* ObjectPtr;
```

Description
ObjectPtr is a **void*** that points to a C++ object.

OC_APPxxxx Messages

See Also   Linking and Embedding Messages

Header File
ocf/ocapp.h

Description
These messages are sent from ObjectComponents to the application's main window. They notify the application of signals and events that come from the OLE system. The actual message sent is WM_OCEVENT. The constants in the table below are carried in the message's *wParam* and identify particular events. To find out what each message carries in its *lParam*, see Event Handlers to look up the corresponding event handlers (such as EvOcAppBorderSpaceSet and EvOcAppMenus) in the ObjectWindows Help (OWL.HLP).

Applications that use the ObjectWindows Library can set up event handlers in their response tables using the EV_OC_xxxx macros defined in ocfevent.h. For more information about the data each message carries, see the descriptions of the corresponding event handlers.

The constants beginning OC_APP indicate events typically handled in the main frame window. Another set of constants beginning OC_VIEW indicate events typically handled in the view object.

Messages	Meaning
OC_APPDIALOGHELP	The user pressed the Help button in one of the standard OLE dialog boxes.
OC_APPBORDERSPACEREQ	Asks the container whether it can give the server border space in its frame window.
OC_APPBORDERSPACESET	Asks the container to give the server border space in its frame window.
OC_APPFRAMERECT	Requests coordinates for the inner rectangle of the container's main window.
OC_APPINSMENUS	Asks the container to merge its menu into the shared menu bar.
OC_APPMENUS	Asks the container to install the merged menu bar.
OC_APPPROCESSMSG	Asks the container to process accelerators and other messages from the server's message queue.
OC_APPRESTOREUI	Tells the container to restore its normal menu and borders because in-place editing has ended.
OC_APPSHUTDOWN	Tells the server when its last linked or embedded object closes down. If the user did not launch the server directly, the server can terminate.
OC_APPSTATUSTEXT	Passes text for the status bar from the server to the container during in-place editing.

See Also
Event Handlers (OWL.HLP)
Messages and Windows
ObjectComponents Messages
OC_VIEWxxxx Messages
TOleFrame_EvOcAppBorderSpaceSet (OWL.HLP)
TOleFrame_EvOcAppMenus (OWL.HLP)
WM_OCEVENT Message

_OCFxxxx Macros

See Also General OLE Classes, Macros, and Type Definitions

Header File
ocf/ocfdefs.h

Description
These macros are used internally to declare classes, functions, and data members in ObjectComponents classes. Their definitions vary depending on whether you build a 16- or 32-bit EXE or DLL. Some of them also force the declaration to__huge.

These macros closely match the corresponding _OWLxxxx macros.

Constant	Meaning
_OCFCLASS	Exports or imports classes for DLLs.
_OCFDATA	Exports or imports data members for DLLs.
_OCFFUNC	Exports or imports member functions for DLLs.

See Also
__huge (BCW.HLP)
_OWLxxxx macros (OWL.HLP)

OC_VIEWxxxx Messages

See Also   Linking and Embedding Messages

Header File
ocf/ocview.h

Description
These messages are sent from ObjectComponents to the application's window procedure. They notify the application of signals and events that come from the OLE system. The actual message sent is WM_OCEVENT. The constants in the table below are carried in the message's *wParam* and identify particular events. To find out what each message carries in its *lParam*, see Event Handlers to look up the corresponding event handlers (such as EvOcViewBorderSpaceSet and EvOcViewDrag) in the ObjectWindows Help (OWL.HLP).

Applications that use the ObjectWindows Library can set up event handlers in their response tables using the EV_OC_xxxx macros defined in ocfevent.h. For more information about the data each message carries, see the descriptions of the corresponding event handlers.

The constants beginning OC_VIEW indicate events typically handled in the view object. Another set of constants beginning OC_APP indicate events that typically concern the application object.

Constant	Meaning
OC_VIEWATTACHWINDOW	Asks the server window to attach to its own frame window or the container's window.
OC_VIEWBORDERSPACEREQ	Asks whether the server can have space in the container's window for in-place editing tools.
OC_VIEWBORDERSPACESET	Requests border space for in-place editing tools in the container's view.
OC_VIEWBREAKLINK	Asks server to break a link to the currently selected data.
OC_VIEWCLIPDATA	Asks the server for Clipboard data in a particular format.
OC_VIEWCLOSE	Tells server to close this remote view.
OC_VIEWDRAG	Requests visual feedback during a drag operation.
OC_VIEWDROP	Accepts a dropped object.
OC_VIEWGETITEMNAME	Asks the server for a moniker identifying the currently selected data.
OC_VIEWGETPALETTE	Asks the server for the palette it uses to paint its object.
OC_VIEWGETSCALE	Asks the container to give scaling information.
OC_VIEWGETSITERECT	Asks container for the site rectangle.
OC_VIEWINSMENUS	Asks the server to insert its menus in the menu bar for in-place editing.
OC_VIEWLOADPART	Asks the server to load its document. (The server's document contains one part.)
OC_VIEWOPENDOC	Asks the server for the extents of its open document.
OC_VIEWPAINT	Asks the server to paint a remote view of its document.
OC_VIEWPARTACTIVATE	Indicates that an embedded part has become active.
OC_VIEWPARTINVALID	Indicates that a part needs repainting.
OC_VIEWPARTSIZE	Asks the server for the extents of its object.
OC_VIEWSAVEPART	Asks the server to save its document. (The server's document

	contains one part.)
OC_VIEWSCROLL	Asks the client to scroll its view because the user is trying to drag something off the edge.
OC_VIEWSETSCALE	Asks the server to handle scaling.
OC_VIEWSETLINK	Asks server to create a link to the currently selected data.
OC_VIEWSETSITERECT	Asks the container to set the site rectangle.
OC_VIEWSETTITLE	Tells the container to append the name of the server to its window caption. Sent when beginning in-place editing.
OC_VIEWSHOWTOOLS	Asks the server to display its tool bars in the container's window for in-place editing.
OC_VIEWTITLE	Gets the title displayed in the view's window.

See Also
Event Handlers (OWL.HLP)
Messages and Windows
ObjectComponents Messages
OC_APPxxxx Messages
TOleWindow_EvOcViewBorderSpaceSet (OWL.HLP)
TOleWindow_EvOcViewDrag (OWL.HLP)

ocrxxxx Aspect Constants

See Also ocrxxxx Constants

Header File
ocf/ocreg.h

Description
These constants identify modes of presenting data. A server might be able to draw the same object several different ways, such as displaying its full content, creating a miniature representation of the content, or representing the type of object with an icon.

When a server registers a data format, it also registers the aspects it supports for each format. The values of these constants are flags and can be combined with the bitwise OR operator (|).

Constant	OLE Equivalent	Meaning
ocrContent	DVASPECT_CONTENT	Show the full content of the object at its normal size.
ocrThumbnail	DVASPECT_THUMBNAIL	Show the content of the object shrunk to fit in a smaller space.
ocrIcon	DVASPECT_ICON	Show an icon representing the type of object.
ocrDocPrint	DVASPECT_DOCPRINT	Show the object as it would look if sent to the printer.

See Also
ocrxxxx Constants
Registration Macros (OWL.HLP)
TOcAspect enum

ocrxxxx Clipboard Constants

See Also ocrxxxx Constants

Header File
ocf/ocreg.h

Description
These constants identify standard data formats for data that applications might share with each other. Use them in the REGFORMAT macro to describe the formats that your documents can import and export.

Constant	Windows Format Name	Meaning
ocrText	CF_TEXT	Array of text characters
ocrBitmap	CF_BITMAP	Device-dependent bitmap
ocrMetafilePict	CF_METAFILEPICT	A Windows metafile wrapped in a METAFILEPICT structure
ocrSylk	CF_SYLK	Symbolic Link Format
ocrDif	CF_DIF	Data Interchange Format
ocrTiff	CF_TIFF	Tag Image File Format
ocrOemText	CF_OEMTEXT	Text containing characters in the original equipment manufacturer's character set (usually ASCII)
ocrDib	CF_DIB	Device-independent bitmap
ocrPalette	CF_PALETTE	GDI palette object
ocrPenData	CF_PENDATA	Data for pen extensions to the operating system
ocrRiff	CF_RIFF	Resource Interchange File Format (often used for multimedia)
ocrWave	CF_WAVE	A sound wave file (uses a subset of the RIFF format)
ocrUnicodeText	CF_UNICODETEXT	Wide-character Unicode text (32-bit only)
ocrEnhMetafile	CF_ENHMETAFILE	Enhanced metafile (32-bit only)
ocrRichText	"Rich Text Format"	RTF tagged text format
ocrEmbedSource	"Embed Source"	OLE object that can be embedded
ocrEmbeddedObject	"Embedded Object"	OLE object that is already embedded
ocrLinkSource	"Link Source"	OLE object that can be linked
ocrObjectDescriptor	"Object Descriptor"	Descriptive information about an OLE object that can be embedded
ocrLinkSrcDescriptor	"Link Source Descriptor"	Descriptive information about an OLE object that can be linked

See Also
ocrxxxx Constants

Registration Macros (OWL.HLP)

ocrxxxx Direction Constants

See Also ocrxxxx Constants

Header File
ocf/ocreg.h

Description
These constants identify directions for passing data. For example, a server might be able to export and import bitmaps but only import metafiles. In that case, it uses *ocrGetSet* for the bitmap format and *ocrGet* for metafiles.

When a server registers a data format, it also specifies whether it can get or set each format.

Constant	Meaning
ocrGet	Imports data in the given format
ocrSet	Exports data in the given format
ocrGetSet	Both exports and imports data in the given format

See Also
ocrxxxx Constants

Registration Macros (OWL.HLP)

ocrxxxx Limit Constants

See Also ocrxxxx Constants

Header File
ocf/ocreg.h

Description
These constants set the maximum number of verbs and data formats that an application is allowed to register for any one document type. Currently these limits are both set to 8.

Constant	Meaning
ocrVerbLimit	Maximum number of verbs a server can register for a document
ocrFormatLimit	Maximum number of data formats an application can register for a document

See Also
ocrxxxx Constants

ocrxxxx Medium Constants

See Also ocrxxxx Constants

Header File
ocf/ocreg.h

Description
These constants identify channels for passing data. A server might be able to pass a particular kind of object as a global memory handle, as a disk file handle, or through a data stream, for example.

When a server registers a data format, it also registers the transfer channels it supports for each format. The values of these constants are flags and can be combined with the bitwise OR operator (|).

Constant	OLE Equivalent	Meaning
ocrHGlobal	TYMED_HGLOBAL	Handle to global memory object
ocrFile	TYMED_FILE	Handle to disk file
ocrIStream	TYMED_ISTREAM	Stream object in a compound file
ocrIStorage	TYMED_ISTORAGE	Storage object in a compound file
ocrGDI	TYMED_GDI	GDI object (such as a bitmap)
ocrMfPict	TYMED_MFPICT	METAFILEPICT structure

See Also
ocrxxxx Constants

Registration Macros (OWL.HLP)

ocrxxxx Object Status Constants

See Also ocrxxxx Constants

Header File
ocf/ocreg.h

Description
These constants describe how an object behaves when presented in particular aspects. Register these options for documents using the REGSTATUS macro.

The values of these constants are flags and can be combined with the bitwise OR operator (|).

Constant	Meaning
ocrActivateWhenVisible	Applies only if *ocrInsideOut* is set. Indicates that the object prefers to be active whenever it is visible. The container is not obliged to comply.
ocrCanLinkByOle1	Used only in OBJECTDESCRIPTOR. Indicates that an OLE 1 container can link to the object.
ocrCantLinkInside	This object, when embedded, should not be made the source of a link.
ocrInsertNotReplace	This object, when placed in a document, should not replace the current selection but be inserted next to it.
ocrInsideOut	The object can be activated and edited without having to install menus or toolbars. Objects of this type can be active concurrently.
ocrIsLinkObject	Set by an OLE 2 link for OLE 1 compatibility. The system sets this bit automatically.
ocrNoSpecialRendering	Same as *ocrRenderingIsDeviceIndependent*.
ocrOnlyIconic	The only useful way the server can draw this object is as an icon. The content view looks like the icon.
ocrRecomposeOnResize	When container site changes size, the server would like to redraw its object. (Presumably the server wants to do something other than scale.)
ocrRenderingIsDeviceIndependent	The object makes no presentation decisions based on the target device. Its presentation data is always the same.
ocrStatic	The object is an OLE static object and cannot be edited.

See Also

ocrxxxx Constants

Registration Macros (OWL.HLP)

ocrxxxx Usage Constants

See Also   ocrxxxx Constants

Header File
ocf/ocreg.h

Description
These constants tell how a server supports concurrent clients. Use them to register the usage key for a server.

Constant	Meaning
ocrSingleUse	One client per application instance
ocrMultipleUse	Multiple clients per application instance
ocrMultipleLocal	Multiple clients supported by separate in-proc server

See Also

ocrxxxx Constants

usage Registration Key

ocrxxxx Verb Attributes Constants

See Also ocrxxxx Constants

Header File
ocf/ocreg.h

Syntax
```
enum ocrVerbAttributes
```

Description
These constants give the container hints about how a verb is used. Register these options for documents using the REGVERBOPT macro.

The values of these constants are flags and can be combined with the bitwise OR operator (|).

Constant	Meaning
ocrNeverDirties	The verb never modifies the object in such a way that it needs to be saved again.
ocrOnContainerMenu	The verb should be displayed on the container's menu of object verbs when the object is active. The standard verbs Hide, Show, and Open should not have this flag set.

See Also
ocrxxxx Constants

Registration Macros (OWL.HLP)

ocrxxxx Verb Menu Flags

See Also ocrxxxx Constants

Header

ocf/ocreg.h

Description

These constants describe how a server's verbs should appear on the container's menu. Register these options for documents using the REGVERBOPT macro.

The values of these constants are flags and can be combined with the bitwise OR operator (|).

Constant	Windows Equivalent	Meaning
ocrGrayed	MF_GRAYED	Make the verb appear gray on the menu. This also disables the verb.
ocrDisabled	MF_DISABLED	Disable the verb so the user cannot choose it.
ocrChecked	MF_CHECKED	Place a check by the verb.
ocrMenuBarBreak	MF_MENUBARBREAK	Places the verb in a new column and adds a vertical line to separate the columns.
ocrMenuBreak	MF_MENUBREAK	Places the verb in a new column without separating the columns.

See Also
ocrxxxx Constants

Registration Macros (OWL.HLP)

Oc*xxxx* Global Functions

See Also

Header File

ocf/ocreg.h

Description

ObjectComponents uses these global functions to register applications in the system registration database. They are called during construction of the application's registrar object. Because ObjectComponents performs all the usual registration chores for you, normally you don't need to call these functions directly. They are global, however, so if you need them you can call them without the overhead of creating the registrar and its related internal objects.

If they fail, these functions throw TXRegistry exceptions.

Function	Purpose
OcRegisterClass	Writes all the information from one registration table to an output stream.
OcRegistryUpdate	Merges all the information from an input stream into the registration database.
OcRegistryValidate	Checks whether information in a registration table matches the corresponding information already recorded in the registration database.
OcSetupDebugReg	Takes the information from a registration table and composes a modified registration table for registering a debugging version.
OcUnRegisterClass	Removes entries from the system's registration database.

The following pseudocode describes the sequence of events ObjectComponents normally follows to register an application.

```
for table = 1 to NumTables {
  validate table's registration                // OcRegistryValidate
  if invalid
    write reg entries to output stream         // OcRegisterClass
}
merge stream into registration database        // OcRegistryUpdate
```

See Also
TXRegistry

OcRegisterClass Function

See Also Oc*xxxx* Global Functions

Syntax

```
long OcRegisterClass(TRegList& regInfo, HINSTANCE hInst, ostream& out,
  TLangId lang, char* filter = 0, TRegItem* defaults = 0, TRegItem* extra =
  0);
```

Description

Writes all the information from a registration table to an output stream. After writing all registration tables to the same output stream, call OcRegistryUpdate to merge the stream with the system registration database. This sequence makes it possible to cancel registration before modifying the registration database if any of the output operations fails.

regInfo is the registration table you want to record. (Registration tables are built with registration macros and conventionally have names like *AppReg* and *DocReg*.)

hInst identifies the application module. ObjectComponents uses it to discover the full path to the application's executable in order to provide a default value for the *path* key.

out is the stream where *OcRegisterClass* writes the registration information it produces. If you set -RegServer on the application command line and specify a file name, the output stream is sent to the file you specify. Otherwise, it is a temporary buffer in memory.

lang is the locale ID for retrieving localized strings from XLAT resources for registration values.

filter lists registration key templates to process. When filter is zero, the function processes all information in *regInfo*. A nonzero value restricts processing to just the templates named. For example, the following command limits processing to three templates.

```
::OcRegisterClass(RegInfo, AppInstance, vstrm, lang, "\001\002\006");
```

The three templates are the command line, the *progid*, and the *clsid*. To find the numbers for particular templates, look for the initialization of the *OcRegTemplates[]* array in OCREG.CPP.

The example command is used in validating an application's existing registration entries. It writes only three entries to the output stream. ObjectComponents then passes the same stream to OcRegistryValidate determines whether those three entries match those already in the system.

defaults points to an array of TRegItem structures, each holding a key/value pair. If the *regInfo* table omits any of the keys supplied in *defaults*, *OcRegisterClass* automatically writes the missing keys with their default values to the output stream.

extra points to an array of *TRegItem* structures, each holding a key/value pair. The *extra* array overrides any *regInfo* entries for the same keys.

The return value is the mask of document template flags (dtxxxx constants) written to the output stream for the *docflags* key.

See Also
dtxxxx Document Template Constants (OWL.HLP)
Localizing Symbol Names
OcRegistryUpdate
OcRegistryValidate
ostream (CLASSLIB.HLP)
path Registration Key
RegistrationMacros (OWL.HLP)
TLangId (OWL.HLP)
TRegList (OWL.HLP)
TRegItem (OWL.HLP)

OcRegistryUpdate Function

See Also Oc*xxxx* Global Functions

Syntax

```
void OcRegistryUpdate(istream& in);
```

Description

Merges the information from *in* with the system registration database. To create the input stream, call OcRegisterClass for each table.

See Also
istream (CLASSLIB.HLP)
OcRegisterClass

OcRegistryValidate Function

See Also		Oc*xxxx* Global Functions

Syntax

```
int OcRegistryValidate(istream& in);
```

Description

Searches the registration database for all the registration entries from *in*. If it finds all the entries, and if all the entries match exactly, the return value is 0. Otherwise, the return value tells how many *in* entries do not have exact matches in the registration database.

To create the input stream, call OcRegisterClass.

See Also
istream (CLASSLIB.HLP)
OcRegisterClass

OcSetupDebugReg Function

See Also   Oc*xxxx* Global Functions

Syntax

```
int OcSetupDebugReg(TRegList& regInfo, TRegItem* regDebug, TLangId lang,
  char* clsid);
```

Description

Reads registration values from the *regInfo* structure and builds a new structure containing just the debug entries.

A registration table can contain keys such as *debugger* and *debugprogid* that are used only to register a debugging version of the same application--one that launches the server directly into a debugger. The presence of the *debugprogid* key causes ObjectComponents to register the same table twice, using different *clsid* values. *OcSetupDebugReg* creates the *TRegList* structure for the second entry by replacing the *progid* and adding the -Debug switch to the command line.

regInfo is the original structure containing debug keys.

regDebug points to a new array that the function creates. The size of the array is *DebugRegCount*.

lang specifies the language to use when retrieving localized strings from XLAT resources.

clsid is the GUID to assign for the debugging version of the structure. ObjectComponents supplies this value for you. This string overrides the clsid in *regInfo*.

The return value is nonzero for success. 1 indicates that the value in the *clsid* parameter was used. -1 indicates that no *clsid* was passed and *regDebug* has the same *clsid* as *regInfo*.

After creating the debugging registration table, call OcRegisterClass to write the entries to an output stream and OcRegistryUpdate to merge them into the registration database.

See Also
debugdesc Registration Key
debugger Registration Key
debugprogid Registration Key
OcRegisterClass
TLangId
TRegList (OWL.HLP)
TRegItem (OWL.HLP)

OcUnregisterClass Function

See Also Oc*xxxx* Global Functions

Syntax

```
int OcUnregisterClass(TRegList& regInfo, TRegItem* extra=0);
```

Description

Removes entries from the system registration database.

regInfo is the table whose entries are to be expunged.

extra points to an array of TRegItem entries supplying default values for keys that might not be present in *regInfo*.

OcUnregisterClass unregisters only the keys stored in *OcUnregParams[]*. They are *debugclsid*, *debugprogid*, *clsid*, *progid*, *extension*, and *permid*. With those entries gone, Windows automatically deletes any remaining related entries.

See Also
Registration Keys
TRegList (OWL.HLP)
TRegItem (OWL.HLP)

OPTIONAL_ARG Macro

See Also

Header File
ocf/automacr.h

Syntax
```
OPTIONAL_ARG(cls, extName, default)
```

Description
An automation server uses this macro in its automation definition (after DEFINE_AUTOCLASS) in order to describe one argument in an exposed method.

After an EXPOSE_METHOD or EXPOSE_METHOD_ID macro, you need to add a list of argument macros, one for each parameter in the method. If the argument has a default value, then use the OPTIONAL_ARG macro.

type is an automation class that describes the argument's data type. For example, if the argument is Boolean value, then *type* should be TAutoBool. For more information, see Automation Data Types.

extName is the public name that a controller uses to refer to this argument. The string can be localized.

default is the default value assigned if the caller chooses to omit the argument.

See Also

Automation Data Types

Automation Definition Macros

EXPOSE_METHOD Macros

Localizing Symbol Names

path Registration Key

See Also Registration Keys

Description

The path where OLE looks to find and load the server.

The path is optional for any server's application registration table. Usually you can omit the path because by default ObjectComponents records the actual path and file name of the server when it registers itself.

To register a path, use the REGDATA macro, passing *path* as the first parameter and the full path string, including .EXE name, as the second parameter.

See Also

REGDATA macro (OWL.HLP)

Registration Keys

Registration Macros (OWL.HLP)

permid Registration Key

See Also		Registration Keys

Description

A string that names the application without indicating any version. The *permid* is just like the *progid* but without a version number. It always represents the latest installed version of a class.

The *permid* key is valid in any registration table. It is always optional. If you register *permid*, you should also register permname. Like the progid, the *permid* cannot be localized.

To register *permid*, use the REGDATA macro, passing *permid* as the first parameter and the ID string as the second parameter.

See Also
permname Registration Key
progid Registration Key
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)
version Registration Key

permname Registration Key

See Also Registration Keys

Description

A string that describes the application without indicating any version. The *permname* is just like the *description* but without a version number. It always represents the latest installed version of a class. A *permname* value can contain up to 40 characters.

The *permname* key is valid in any registration table. It is always optional. If you register *permname*, you should also register *permid*. The *permname* string should be localized.

To register *permname*, use the REGDATA macro, passing *permname* as the first parameter and the descriptive string as the second parameter.

See Also
description Registration Key
Localizing Symbol Names
permid Registration Key
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)
version Registration Key

progid Registration Key

See Also Registration Keys

Description

Registers a string which uniquely identifies the class.

The string can contain up to 39 characters. The first character must be a letter. Subsequent characters can be letters, digits, or periods (no spaces or other delimiters). Conventionally, the *progid* value has three parts separated by periods. They are the program name, an object name, and a version number. The value of the *progid* cannot be localized.

To be a linking and embedding server, an application must register a *progid* and insertable for at least one document type.

A *progid* string is required in every application registration table. To register a *progid*, use the REGDATA macro, passing *progid* as the first parameter and the identifier string as the second parameter.

```
REGDATA(progid, "DrawingPad.Application.2")
```

See Also
insertable Registration Key
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)

REQUIRED_ARG Macro

See Also

Header File
ocf/automacr.h

Syntax
```
REQUIRED_ARG(type, extName);
```

Description
An automation server uses this macro in its automation definition (after DEFINE_AUTOCLASS) in order to describe one argument in an exposed method.

After an EXPOSE_METHOD or EXPOSE_METHOD_ID macro, you need to add a list of argument macros, one for each parameter in the method. If the argument does not have a default value and is not an object, then use the REQUIRED_ARG macro.

type is an automation class that describes the argument's data type. For example, if the argument is Boolean value, then *type* should be TAutoBool. For more information, see Automation Data Types.

extName is the external, public name that a controller uses to refer to this argument. The string can be localized.

See Also
Automation Data Types
Automation Definition Macros
EXPOSE_METHOD Macros
Localizing Symbol Names

STATSTG Struct

See Also Compound File I/O Enumerated Types and Structs

Header File

ocf/ocstorag.h

Description

This structure holds information describing a stream or storage object (one that supports the *IStream* or *IStorage* interface.) The information includes, for example, the size of the file element, its creation and access times, its operating modes, and its identifiers.

Public Data Members

```
FILETIME atime;
ULARGE_INTEGER cbSize;
IID clsid;
FILETIME ctime;
uint32 grfLocksSupported;
uint32 grfMode;
uint32 grfStateBits;
FILETIME mtime;
char far* pwcsName;
uint32 reserved;
uint32 type;
```

See Also

TOcStorage::Stat

TOcStream::Stat

STATSTG::atime

See Also STATSTG

Syntax

```
FILETIME atime;
```

Description

atime stores the time when the last access operation was performed on the file element. It is defined for storages only. Streams do not have time stamps.

See Also
FILETIME struct

STATSTG::cbSize

STATSTG

Syntax

```
ULARGE_INTEGER cbSize;
```

Description

cbSize stores the number of bytes in a stream (or a lockbytes array). For storages, this value is not defined.

STATSTG::clsid

See Also STATSTG

Syntax

```
IID clsid;
```

Description

Holds a globally unique identifier (GUID) assigned as the *clsid* of a storage object. The *clsid* of any newly created storage is CLSID_NULL. Streams do not have class identifiers.

See Also
TOcStorage::SetClass

STATSTG::ctime

See Also STATSTG

Syntax

```
FILETIME ctime;
```

Description

ctime stores the time when the file element was created. It is defined for storages only. Streams do not have time stamps.

See Also
FILETIME struct

STATSTG::grfLocksSupported

See Also STATSTG

Syntax

```
uint32 grfLocksSupported;
```

Description

Contains bit flags indicating whether a stream or lockbyte object supports particular kinds of locks. The possible flags are LOCK_WRITE, LOCK_EXCLUSIVE, and LOCK_ONLYONCE. They can be combined with the bitwise OR operator.

The value of *grfLocksSupported* is undefined for storage objects because they do not support locks at all.

See Also

TOcStream::LockRegion

TOcStream::UnlockRegion

STATSTG::grfMode

See Also STATSTG

Syntax

```
uint32 grfMode;
```

Description

Contains bit flags indicating the access mode in which the file element was opened. The possible values are STGM_*xxxx* flags.

See Also
STGM_xxxx constants
TOcStorage Public Constructors and Destructor
TOcStorage::CreateStorage

STATSTG::grfStateBits

See Also STATSTG

Syntax

```
uint32 grfStateBits;
```

Description

Holds information about the current state of a storage. When first created, a storage's state is 0. Currently no state bits are defined, but all 32 are reserved for system use and applications should not use them privately.

See Also
TOcStorage::SetStateBits

STATSTG::mtime

See Also STATSTG

Syntax

```
FILETIME mtime;
```

Description

mtime stores the time when the file element was last modified. It is defined for storages only. Streams do not have time stamps.

See Also
FILETIME struct

STATSTG::pwcsName

See Also STATSTG

Syntax

```
char far* pwcsName;
```

Description

Points to the name assigned at creation to identify the file element. The *Stat* function allocates space for this string, but the caller is responsible for freeing it.

See Also

TOcStorage::Stat

TOcStream::Stat

STATSTG::reserved

STATSTG

Syntax

```
uint32 reserved;
```

Description

This field is reserved and should be ignored.

STATSTG::type

STATSTG

Syntax

```
uint32 type;
```

Description

The value in this field indicates the type of object the STATSTG structure describes. The possible values are STGTY_STORAGE, STGTY_STREAM, and STGTY_LOCKBYTES.

STGC enum

See Also Compound File I/O Enumerated Types and Structs

Header File

ocf/ocstorag.h

Description

These flags are used in *Commit* commands for compound files to control how changes are committed. They are bit values and can be combined with the bitwise OR operator (|).

Constant	Meaning
STGC_DEFAULT	Commits changes normally.
STGC_OVERWRITE	Allows new data to overwrite old data. Without this flag, *Commit* writes to a new area and erases the old data only after the *Write* operation succeeds.
STGC_ONLYIFCURRENT	Commits changes only if doing so does not overwrite changes already made by another process.
STGC_DANGEROUSLYCOMMIT-MERELYTODISKCACHE	Commits changes without flushing the disk caching buffer. Sacrifices security for speed.

For general use, STGC_ONLYIFCURRENT is recommended; STGC_OVERWRITE is not.

See Also

TOcStorage::Commit

TOcStream::Commit

STGMxxxx Constants

See Also

Header File

ocf/ocstorag.h

Description

These constants specify access modes for file elements: streams, storages, and lockbyte arrays. The constants are bit flags and can be combined with the bitwise OR operator (|).

Constant	Meaning
STGM_DIRECT	Uses direct access mode (commits each change immediately)
STGM_TRANSACTED	Uses transacted mode (buffers changes until committed explicitly)
STGM_READ	Allows read operations
STGM_WRITE	Allows write operations
STGM_READWRITE	Allows read and write operations
STGM_SHARE_EXCLUSIVE	Denies all access to other tasks
STGM_SHARE_DENY_WRITE	Denies write access to other tasks
STGM_SHARE_DENY_READ	Denies read access to other tasks
STGM_SHARE_DENY_NONE	Allows all access to other tasks
STGM_CREATE	Creates the file element if it does not already exsit
STGM_CONVERT	If the a file element with the given name already exists, copies its contents to a stream called CONTENTS and creates a new element with the given name
STGM_PRIORITY	Gives the opener exclusive access to the committed version of the file to reduce the cost of a subsequent copy operation
STGM_DELETEONRELEASE	Deletes the file element from the disk as soon as its reference count reaches 0

For more information about each option, see OLE.HLP.

See Also

TOcStorage Public Constructors and Destructor

TOcStorage::CreateStorage

TOcStorage::CreateStream

TOcStorage::OpenStorage

TOcStorage::OpenStream

TOcStream Public Constructors and Destructor

TAutoBase Class

See Also Automation Classes

Header File

ocf/autodefs.h

Description

TAutoBase is a base class for deriving automatable objects. The class does only one thing: whenever an object of *TAutoBase* is destroyed, the destructor notifies OLE that the object is no longer available.

Automated objects are not required to derive from *TAutoBase*. Doing so is simply a safeguard and matters only if the logic of the program makes it possible for the automated object to be destroyed by non-automated means while still connected to an OLE controller.

If you are using *TAutoBase* to derive classes with explicit class specifiers that do not match the default specifiers for the application's model, then be sure to define the _AUTOCLASS macro.

Public Destructor

```
virtual ~TAutoBase();
```

See Also

_AUTOCLASS Macro

Telling OLE When the Object Goes Away

TAutoBase Public Destructor

TAutoBase

Syntax
```
virtual ~TAutoBase();
```

Description

The virtual destructor--the only member of class *TAutoBase*--sends OLE an obituary when the object is destroyed. The notification matters in cases where the object might be destroyed by non-automated means, without the knowledge of OLE, while still connected to an automation controller. Sending the obituary prevents a crash if OLE subsequently sends a command to the nonexistent object.

TAutoBool struct

See Also   Automation Data Types

Header File

ocf/autodefs.h

Description

Use *TAutoBool* in an automation definition to describe the parameters and return values of automated methods.

Public Data Member

```
static TAutoType ClassInfo;
```

See Also

Automation Data Types

Automation Definition Macros

TAutoBool::ClassInfo

TAutoBool

Syntax

```
static TAutoType ClassInfo;
```

Description

The *ClassInfo* member of *TAutoBool* holds information that identifies the Boolean data type.

TAutoCommand Class

Automation Classes

Header File
ocf/autodefs.h

Description
TAutoCommand is an abstract base class for automation command objects. An automation server constructs a command object whenever it receives a command from an automation controller. The command object receives all parameters as VARIANT unions from OLE. The compiler generates calls to command object conversion functions in order to extract the proper C++ data type from the union.

All this happens internally. Normally you should not have to construct or manipulate *TAutoCommand* objects directly.

Public Destructor
```
TAutoCommand(int attr);
virtual ~TAutoCommand();
```

Type DefinitionS
```
typedef bool (*TCommandHook)(TAutoCommand& cmdObj);
typedef const char* (*TErrorMsgHook)(long errCode);
```

Public Member Functions
```
void ClearFlag(int mask);
virtual void Execute();
void Fail(TXAuto::TError);
TAutoSymbol* GetSymbol();
virtual TAutoCommand& Invoke();
bool IsPropSet();
static const char* LookupError(long errCode);
virtual int Record(TAutoStack& q);
virtual long Report();
virtual void Return(TAutoVal& v);
void SetFlag(int mask);
void SetSymbol(TAutoSymbol* sym);
static TErrorMsgHook SetErrorMsgHook(TErrorMsgHook callback);
static TCommandHook SetCommandHook(TCommandHook callback);
bool TestFlag(int mask);
virtual TAutoCommand* Undo();
virtual bool Validate();
```

Protected Data Members
```
int Attr;
TAutoSymbol* Symbol;
```

TAutoCommand Public Constructor and Destructor

See Also TAutoCommand

Syntax

Constructor

```
TAutoCommand(int attr);
```

Destructor

```
virtual ~TAutoCommand();
```

Description

Creates a command having the attributes set in the *attr* flag mask. The flags are defined in the AutoSymFlag **enum**.

Destructor

Destroys the *TAutoCommand* object.

See Also
AutoSymFlag enum

TAutoCommand::ClearFlag

See Also TAutoCommand

Syntax

```
void ClearFlag(int mask);
```

Description

Clears all the flags in *mask*. The flags are defined in the AutoSymFlag **enum**.

See Also
AutoSymFlag enum

TAutoCommand::Execute

TAutoCommand

Syntax

```
virtual void Execute();
```

Description

Executes the automation command by invoking the internal C++ member of the automated class to which the command belongs.

TAutoCommand::Fail

See Also TAutoCommand

Syntax

```
void Fail(TXAuto::TError);
```

Description

Throws whatever exception is indicated by the parameter.

See Also
TXAuto::TError enum

TAutoCommand::GetSymbol

TAutoCommand

Syntax

```
TAutoSymbol* GetSymbol();
```

Description

Retrieves the symbol that generates this command.

TAutoCommand::Invoke

See Also TAutoCommand

Syntax

```
virtual TAutoCommand& Invoke();
```

Description

Initiates the process of executing a command. The user can override the usual process by supplying a hook with the AUTOINVOKE macro.

See Also

AUTOINVOKE macro

TAutoCommand::SetCommandHook

TAutoCommand::IsPropSet

TAutoCommand

Syntax

```
bool IsPropSet();
```

Description

Returns true if the *asSet* property flag is set. This flag indicates that the command assigns a value to some property of the automated class and does not return a value.

TAutoCommand::LookupError

See Also TAutoCommand

Syntax

```
static const char* LookupError(long errCode);
```

Description

Translates an error code from a function into a message string for the user. *errCode* is a function status value sent by *Report*. *LookupError* works by calling a function you have installed with SetErrorMsgHook. You do not have to call *LookupError* directly. If you have installed an error message hook, *LookupError* is called for you at the right time.

See Also
TAutoCommand::Report

TAutoCommand::Record

See Also TAutoCommand

Syntax

```
virtual int Record(TAutoStack& q);
```

Description

Records the command and its arguments by calling any hook the programmer might have supplied in the automation declaration with the AUTORECORD macro.

Recording is not supported in the current version of ObjectComponents.

See Also

AUTORECORD macro

TAutoCommand::Report

See Also TAutoCommand

Syntax

```
virtual long Report();
```

Description

The AUTOREPORT macro invokes this function to translate the status code a command returns into an error code.

See Also

AUTOREPORT Macro

TAutoCommand::SetErrorMsgHook

TAutoCommand::Return

TAutoCommand

Syntax

```
virtual void Return(TAutoVal& v);
```

Description

Converts whatever value the internal C++ command returned into a VARIANT union. The converted value is passed to OLE. This is what the automation controller receives as its return value.

TAutoCommand::SetErrorMsgHook

See Also TAutoCommand

Syntax

```
static TErrorMsgHook SetErrorMsgHook(TErrorMsgHook callback);
```

Description

Installs a user-defined callback function of type TErrorMsgHook to be called if the command returns an error code.

See Also

TAutoCommand::LookupError

TAutoCommand::Report

TAutoCommand::SetCommandHook

TAutoCommand::TErrorMsgHook typedef

TAutoCommand::SetCommandHook

See Also TAutoCommand

Syntax

```
static TCommandHook SetCommandHook(TCommandHook callback);
```

Description

Installs a user-defined callback function of type TCommandHook to be called whenever the command is executed. The command hook is useful for monitoring automation calls.

See Also

TAutoCommand::Invoke

TAutoCommand::SetErrorMsgHook

TAutoCommand::TCommandHook typedef

TAutoCommand::SetFlag

See Also TAutoCommand

Syntax
```
void SetFlag(int mask);
```

Description
Sets all the flags in *mask*. The flags are defined in the TAutoSymFlag **enum**.

See Also
AutoSymFlag enum

TAutoCommand::SetSymbol

TAutoCommand

Syntax

```
void SetSymbol(TAutoSymbol* sym);
```

Description

Assigns a symbol to the command object. The symbol is set internally. It is taken from the tables built by the automation definition and declaration of the automated class.

TAutoCommand::TestFlag

See Also TAutoCommand

Syntax

```
bool TestFlag(int mask);
```

Description

Returns **true** if any of the flags in *mask* are set for this command. The flags are defined in the TAutoSymFlag **enum**.

See Also
AutoSymFlag enum

TAutoCommand::Undo

See Also TAutoCommand

Syntax

```
virtual TAutoCommand* Undo();
```

Description

Generates a command for the undo stack by calling any hook the programmer might have supplied in the automation declaration with the AUTOUNDO macro.

Undoing commands is not supported in the current version of ObjectComponents.

See Also
AUTOUNDO macro

TAutoCommand::TCommandHook typedef

See Also TAutoCommand

Syntax

```
typedef bool (*TCommandHook)(TAutoCommand& cmdObj);
```

Description

Describes the prototype for a user-defined callback function called during Invoke, before executing the automation command object. *cmdObj* is the object about to be executed. If the callback returns **false**, *Invoke* does not execute the command.

See Also

TAutoCommand::Invoke

TAutoCommand::SetCommandHook

TAutoCommand::TErrorMsgHook typedef

See Also TAutoCommand

Syntax

```
typedef const char* (*TErrorMsgHook)(long errCode);
```

Description

Describes the prototype for a user-defined callback function called after Invoke to process any error code the command might return. *errCode* is the status result. The callback is expected to return a string describing the error for the user.

See Also

TAutoCommand::LookupError

TAutoCommand::SetErrorMsgHook

TAutoCommand::Validate

See Also TAutoCommand

Syntax

```
virtual bool Validate();
```

Description

Tests the validity of the command's parameters by executing whatever validation function or expression the programmer supplied in the automation declaration with the AUTOVALIDATE macro.

See Also
AUTOVALIDATE macro

TAutoCommand::Attr

See Also TAutoCommand

Syntax

```
int Attr;
```

Description

Attribute and state flags. The flags are defined in the AutoSymFlag **enum**.

See Also
AutoSymFlag enum

TAutoCommand::Symbol

TAutoCommand

Syntax

```
TAutoSymbol* Symbol;
```

Description

The symbol entry that generates this command. OLE passes this symbol to make this command execute.

TAutoCurrency struct

See Also   Automation Data Types

Header File
ocf/autodefs.h

Description
TAutoCurrency is an automation data type that helps ObjectComponents provide type checking for members of an automated class exposed to OLE. Use *TAutoCurrency* in an automation definition to identify currency values.

Public Data Member
```
static TAutoType ClassInfo;
```

See Also

Automation Data Types

Automation Definition Macros

TAutoCurrency::ClassInfo

TAutoCurrency

Syntax

```
static TAutoType ClassInfo;
```

Description

The *ClassInfo* member of *TAutoCurrency* holds information that identifies data as a currency value.

TAutoCurrencyRef struct

See Also Automation Data Types

Header File
ocf/autodefs.h

Description
TAutoCurrencyRef is an automation data type that helps ObjectComponents provide type checking for members of an automated class exposed to OLE. Use *TAutoCurrencyRef* in an automation definition to identify currency values passed by pointer, not by value.

Public Data Member
```
static TAutoType ClassInfo;
```

See Also

Automation Data Types

Automation Definition Macros

TAutoCurrencyRef::ClassInfo

TAutoCurrencyRef

Syntax

```
static TAutoType ClassInfo;
```

Description

The *ClassInfo* member of *TAutoCurrencyRef* holds information that identifies the TAutoCurrency* data type.

TAutoDate struct

See Also Automation Data Types

Header File
ocf/autodefs.h

Description
TAutoDate is an automation data type that helps ObjectComponents provide type checking for members of an automated class exposed to OLE. Use *TAutoDate* in an automation definition to identify date values stored as type **double**.

Public Data Members
```
static TAutoType ClassInfo;
double Date;
```

Public Constructors
```
TAutoDate(double d);
TAutoDate();
```

Public Member Function
```
operator double();
```

See Also

Automation Data Types

Automation Definition Macros

TAutoDate Public Constructor

TAutoDate

Form 1

```
TAutoDate();
```

Form 2

```
TAutoDate(double d);
```

Description

Form 1: Creates an empty *TAutoDate*.

Form 2: Creates a *TAutoData* that initially holds the value in *d*, assumed to be a date.

TAutoDate::ClassInfo

TAutoDate

Syntax

```
static TAutoType ClassInfo;
```

Description

The *ClassInfo* member of *TAutoDate* holds information that identifies the date data type.

TAutoDate::Date

TAutoDate

Syntax
```
double Date;
```

Description
Stores a date as a 32-bit value.

TAutoDate::operator double

TAutoDate

Syntax

```
operator double();
```

Description

Returns the value stored in the Date field of *TAutoDate*.

TAutoDateRef struct

See Also Automation Data Types

Header File
ocf/autodefs.h

Description
TAutoDateRef is an automation data type that helps ObjectComponents provide type checking for members of an automated class exposed to OLE. Use *TAutoDateRef* in an automation definition to identify **double** date values passed by pointer, not by value.

Public Data Members

```
static TAutoType ClassInfo;
```

See Also

Automation Data Types

Automation Definition Macros

TAutoDateRef::ClassInfo

TAutoDateRef

Syntax

```
static TAutoType ClassInfo;
```

Description

The *ClassInfo* member of *TAutoDateRef* holds information that identifies the TAutoDate* data type.

TAutoDouble struct

See Also Automation Data Types

Header File
ocf/autodefs.h

Description
TAutoDouble is an automation data type that helps ObjectComponents provide type checking for members of an automated class exposed to OLE. Use *TAutoDouble* in an automation definition to identify **double** values.

Public Data Member
```
static TAutoType ClassInfo;
```

See Also

Automation Data Types

Automation Definition Macros

TAutoDouble::ClassInfo

TAutoDouble

Syntax

```
static TAutoType ClassInfo;
```

Description

The *ClassInfo* member of *TAutoDouble* holds information that identifies the **double** data type.

TAutoDoubleRef struct

See Also Automation Data Types

Header File
ocf/autodefs.h

Description
TAutoDoubleRef is an automation data type that helps ObjectComponents provide type checking for members of an automated class exposed to OLE. Use *TAutoDoubleRef* in an automation definition to identify **double** values passed by pointer, not by value.

Public Data Member
```
static TAutoType ClassInfo;
```

See Also

Automation Data Types

Automation Definition Macros

TAutoDoubleRef::ClassInfo

TAutoDoubleRef

Syntax

```
static TAutoType ClassInfo;
```

Description

The *ClassInfo* member of *TAutoDoubleRef* holds information that identifies the **double*** data type.

TAutoEnumerator<> class

See Also   Automation Classes

Header File
ocf/autodefs.h

Description
An automation controller creates a *TAutoEnumerator* object to enumerate items in a collection held by an automation server. A collection can contain any set of similar values or objects that the server chooses to expose as a group. The items in the collection might be numbers in an array, for example, or each one might be an automated object.

The type you pass to the template is the type of value the collection holds. If the collection is a set of integer values, pass **int**. If the collection holds automated objects, pass the controller's proxy class (derived from TAutoProxy).

At first, a newly created enumerator is empty. After you call *Step*, the enumerator holds the first value in the collection. To see the value, call *Value* if the collection contains values of intrinsic data types, such as int or **float**, or call *Object* if the collection contains automated objects. *Step*, *Value*, and *Object* are the most important methods of *TAutoEnumerator*. The others are generally called for you at the right time.

Public Constructors and Destructor
```
~TAutoEnumerator();
TAutoEnumerator(const TAutoEnumerator& copy);
TAutoEnumerator();
```

Public Member Functions
```
void Bind(TAutoVal& val);
void Clear();
void Object(TAutoProxy& prx);
bool Step();
void Unbind();
void Value(T& v);
```

See Also
Enumerating Automated Collections

TAutoEnumerator Public Constructor and Destructor

See Also TAutoEnumerator

Form 1
```
TAutoEnumerator();
```

Form 2
```
TAutoEnumerator(const TAutoEnumerator& copy);
```

Destructor
```
~TAutoEnumerator();
```

Description

However it is constructed, a newly created TAutoEnumerator object does not yet hold any value. Always call Step to get the first item before calling Value or Object to see the item.

Form 1: Constructs an enumerator object but does not attach it to any automated collection.

Form 2: Constructs a new enumerator object by copying an existing one. Both enumerators are attached to the same collection of objects.

Destructor

Detaches the enumerator from its collection before allowing the enumerator to be destroyed.

See Also
TAutoEnumerator::Step

TAutoEnumerator::Bind

See Also		TAutoEnumerator

Syntax

```
void Bind(TAutoVal& val);
```

Description

Connects the enumerator to the collection object, *val*. *Bind* is called internally when the controller passes the enumerator object to a method that returns a collection.

See Also
TAutoEnumerator::Unbind

TAutoEnumerator::Clear

See Also TAutoEnumerator

Syntax

```
void Clear();
```

Description

Empties the enumerator so that it no longer points to any item in the collection. This method is called internally during Step.

See Also
TAutoEnumerator::Step

TAutoEnumerator::Object

See Also TAutoEnumerator

Syntax

```
void Object(TAutoProxy& prx);
```

Description

Returns in *prx* the current object from the collection. Use *Object* if the items in the collection are automated objects. If the collection contains data values, then call Value instead.

To advance the enumerator so that *Object* returns the next object, call Step.

See Also

TAutoEnumerator::Step

TAutoEnumerator::Value

TAutoEnumerator::Step

See Also TAutoEnumerator

Syntax

```
bool Step();
```

Description

Advances the enumerator object one step so that Value returns the next item in the collection. *Step* returns **false** when called after the enumerator has reached the last item in the collection.

See Also

TAutoEnumerator::Object

TAutoEnumerator::Value

TAutoEnumerator::Unbind

See Also TAutoEnumerator

Syntax

```
void Unbind();
```

Description

Disconnects the enumerator object from the collection it currently enumerates.

See Also
TAutoEnumerator::Bind

TAutoEnumerator::Value

See Also TAutoEnumerator

Syntax

```
void Value(T& v);
```

Description

Returns in *v* the current item from the collection. Use *Value* if the items in the collection are data values. If the collection contains objects, then call Object instead.

To advance the enumerator so that *Value* returns the next item, call Step.

See Also

TAutoEnumerator::Object

TAutoEnumerator::Step

TAutoFloat struct

See Also Automation Data Types

Header File

ocf/autodefs.h

Description

TAutoFloat is an automation data type that helps ObjectComponents provide type checking for members of an automated class exposed to OLE. Use *TAutoFloat* in an automation definition to identify **float** values.

Public Data Member

```
static TAutoType ClassInfo;
```

See Also

Automation Data Types

Automation Definition Macros

TAutoFloat::ClassInfo

TAutoFloat

Syntax
```
static TAutoType ClassInfo;
```

Description
The *ClassInfo* member of *TAutoFloat* holds information that identifies the **float** data type.

TAutoFloatRef struct

See Also		Automation Data Types

Header File

ocf/autodefs.h

Description

TAutoFloatRef is an automation data type that helps ObjectComponents provide type checking for members of an automated class exposed to OLE. Use *TAutoFloatRef* in an automation definition to identify **float** values passed by pointer, not by value.

Public Data Member

```
static TAutoType ClassInfo;
```

See Also

Automation Data Types

Automation Definition Macros

TAutoFloat::ClassInfo

TAutoFloatRef

Syntax

```
static TAutoType ClassInfo;
```

Description

The *ClassInfo* member of *TAutoFloatRef* holds information that identifies the **float*** data type.

TAutoInt typedef

See Also Automation Data Types

Header File

ocf/autodefs.h

Syntax

```
#if (MAXINT==MAXSHORT)
typedef TAutoShort TAutoInt;
#else
typedef TAutoLong TAutoInt;
#endif
```

Description

TAutoInt is an automation data type that helps ObjectComponents provide type checking for members of an automated class exposed to OLE. Use *TAutoInt* in an automation definition to identify **int** values. As with **int**, whether *TAutoInt* refers to a 16-bit or a 32-bit value depends on the compiler.

See Also

Automation Data Types

TAutoLong

TAutoShort

TAutoIterator Class

See Also Automation Classes

Header File

ocf/autodefs.h

Description

TAutoIterator is a pure virtual base class for iterator objects. An iterator is used to enumerate a collection of other objects. The iterator's methods let the caller step through a list of objects and examine each one in turn.

An automation server needs to create an iterator in any automated object that represents a collection of other objects. To create an iterator, the server usually inserts an AUTOITERATOR macro in the automation definition of the collection class (after DEFINE_AUTOCLASS).

In most cases, you do not need to work with the iterator class directly because the AUTOITERATOR macro implements the object for you. In cases where the iterator requires a more complex implementation, however, you might need to define the class directly yourself.

You can still declare the class using AUTOITERATOR_DECLARE instead of AUTOITERATOR. This is just a shortcut for writing out all the standard members of an iterator object by hand.

TAutoIterator has five pure virtual members that any derived class must implement. These five functions compose a standard interface for iterators in automated collection objects. They are *Init*, *Test*, *Step*, *Return*, and *Copy*. The first four correspond to steps in a **for** loop that steps through the collection. (See AUTOITERATOR for a description of the correspondence.) *Copy* creates a duplicate iterator.

The constructors are protected because *TAutoIterator* should be constructed only by a derived class.

Besides implementing the inherited virtual functions, a class derived from *TAutoIterator* also typically declares one or more data members that record the iterator's current state. Usually the state variable remembers a position in the sequence of enumerated objects.

TAutoIterator is a COM object and implements the *IUnknown* interface.

Public Member Functions

```
virtual TAutoIterator* Copy()=0;
TAutoSymbol* GetSymbol();
virtual void Init()=0;
operator IUnknown;
virtual void Return(TAutoVal& v)=0;
void SetSymbol(TAutoSymbol* sym);
virtual void Step()=0;
virtual bool Test()=0;
```

Protected Constructors

```
TAutoIterator(TAutoIterator& copy);
TAutoIterator(TServedObject& owner);
```

Protected Data Member

```
TServedObject& Owner;
```

See Also

Automation Definition Macros

AUTOITERATOR Macros

Implementing an Iterator for the Collection

TAutoIterator Protected Constructors

TAutoIterator

Form 1
```
TAutoIterator (TServedObject& owner);
```

Form 2
```
TAutoIterator (TAutoIterator& copy);
```

Description

The constructors are protected because only a derived class should construct a *TAutoIterator*.

Form 1: Constructs an iterator to enumerate items held in the *owner* class. *owner* can be any automated class.

Form 2: Constructs an iterator by creating a copy of another iterator. Both iterators enumerate the same collection of objects.

TAutoIterator::Copy

See Also TAutoIterator

Syntax

```
virtual TAutoIterator* Copy()=0;
```

Description

Returns a copy of the iterator object. Your implementation should copy the iterator's state variables.

See Also

TAutoIterator::Init

TAutoIterator::Return

TAutoIterator::Step

TAutoIterator::Test

TAutoIterator::GetSymbol

See Also TAutoIterator

Syntax

```
TAutoSymbol* GetSymbol();
```

Description

Retrieves the automation symbol associated with the iterator. Usually you do not need to call this function.

See Also
TAutoIterator::SetSymbol

TAutoIterator::Init

See Also TAutoIterator

Syntax

```
virtual void Init()=0;
```

Description

Initializes any state variables in the iterator. The primary task of an iterator is to loop through a list of objects enumerating them one by one. *Init* tells the iterator to prepare for beginning a new pass through the loop. For example, if the iterator's state variable is called *index*, *Init* might say

```
index = 0;
```

See Also

TAutoIterator::Copy

TAutoIterator::Return

TAutoIterator::Step

TAutoIterator::Test

TAutoIterator::operator IUnknown*()

See Also TAutoIterator

Syntax

```
operator IUnknown*();
```

Description

Returns a pointer to the iterator's *IUnknown* OLE interface and calls AddRef on the interface pointer. This operator is called internally to return the iterator to OLE. Usually you do not need to call it directly yourself.

See Also
TocStorage::AddRef

TAutoIterator::Return

See Also TAutoIterator

Syntax

```
virtual void Return(TAutoVal& value)=0;
```

Description

Extracts one item from a collection and returns a reference to it in the value parameter. The primary task of an iterator is to loop through a list of objects enumerating them one by one. *Return* is the command that retrieves a different item from the collection on each pass through the loop. For example, *Return* might look like this:

```
value = (Collection->Array)[Index]
```

where *value* is the function's parameter, *Collection* points to the enclosing collection object, *Array* is a member of *Collection*, and *Index* is the iterator's state variable.

value is type TAutoVal and represents a VARIANT union, which is the format in which OLE passes values. *TAutoVal* defines conversion operators to handle standard C++ data types as well as C++ strings, TAutoCurrency, TAutoData, and automated C++ objects. The items in a collection can be any of these types.

See Also

TAutoIterator::Copy

TAutoIterator::Init

TAutoIterator::Step

TAutoIterator::Test

TAutoIterator::SetSymbol

See Also TAutoIterator

Syntax

```
void SetSymbol(TAutoSymbol* sym);
```

Description

Associates an automation symbol with the iterator. *SetSymbol* is called internally during the construction of the iterator. Usually you do not need to call it directly yourself.

See Also
TAutoIterator::GetSymbol

TAutoIterator::Step

See Also TAutoIterator

Syntax

```
virtual void Step()=0;
```

Description

Advances the iterator to point to the next item in a collection. The primary task of an iterator is to loop through a list of objects enumerating them one by one. *Step* is like the *i*++ statement in a **for** loop. It changes the state of the iterator to focus on the next item. For example, if the iterator's state variable is called *index*, *Step* might simply say:

```
index++;
```

See Also

TAutoIterator::Copy

TAutoIterator::Init

TAutoIterator::Return

TAutoIterator::Test

TAutoIterator::Test

See Also TAutoIterator

Syntax

```
virtual bool Test()=0;
```

Description

Tests whether all items have been enumerated. The primary task of an iterator is to loop through a list of objects, enumerating them one by one. *Test* returns **true** if more objects remain to be enumerated and **false** when it reaches the end of the list. For example, if the iterator's state variable is called *index*, *Test* might say

```
return (index >= NUM_ITEMS);
```

See Also
TAutoIterator::Copy
TAutoIterator::Init
TAutoIterator::Return
TAutoIterator::Step

TAutoIterator::Owner

TAutoIterator

Syntax

```
TServedObject& Owner;
```

Description

Holds a reference to the collection object that encloses the iterator. *Owner* is initialized by the constructor. The undocumented *TServedObject* class implements the interfaces that a client expects to find on an OLE object. ObjectComponents uses this class internally. *Owner* can be any automated object.

TAutoLong struct

See Also Automation Data Types

Header File
ocf/autodefs.h

Description
TAutoLong is an automation data type that helps ObjectComponents provide type information for members of an automated class exposed to OLE. Use *TAutoLong* in an automation definition to identify **long** values.

Public Data Member
```
static TAutoType ClassInfo;
```

See Also

Automation Data Types

Automation Definition Macros

TAutoLong::ClassInfo

TAutoLong

Syntax

```
static TAutoType ClassInfo;
```

Description

The *ClassInfo* member of *TAutoLong* holds information that identifies the **long** data type.

TAutoLongRef struct

See Also Automation Data Types

Header File

ocf/autodefs.h

Description

TAutoLongRef is an automation data type that helps ObjectComponents provide type information for members of an automated class exposed to OLE. Use *TAutoLongRef* in an automation definition to identify **long** values passed as pointers, not by value.

Public Data Member

```
static TAutoType ClassInfo;
```

See Also
Automation Data Types
Automation Definition Macros

TAutoLongRef::ClassInfo

TAutoLong

Syntax

```
static TAutoType ClassInfo;
```

Description

The *ClassInfo* member of *TAutoLongRef* holds information that identifies the **long*** data type.

TAutoObject <> Class

Automation Classes

Header File

ocf/autodefs.h

Description

TAutoObject holds a pointer to a C++ object. *TAutoObject* casts the pointer to different data types appropriately when an automation operation requires conversion. It also retrieves type information about the object when needed during automation. Think of *TAutoObject* as a smart pointer.

ObjectComponents often creates smart pointers for you. Usually you do not need to manipulate *TAutoObject* objects directly.

Public Constructors

```
TAutoObject(IDispatch* dispatch);
TAutoObject(T& ref);
TAutoObject(T* point);
TAutoObject();
```

Public Member Functions

```
T& operator *();
void operator =(T& ref);
void operator =(T* point);
void operator =(IDispatch* dispatch);
operator T&();
operator T*();
TObjectDescriptor();
```

Protected Data Member

```
T* P;
```

TAutoObject::operator *()

TAutoObject

Syntax

```
T& operator*();
```

Description

The dereference operator returns a reference to the object whose pointer *TAutoObject* holds.

TAutoObject::operator =

See Also TAutoObject

Syntax

Form 1
```
void operator=(T* point);
```

Form 2
```
void operator=(T& ref);
```

Form 3
```
void operator=(IDispatch* dispatch);
```

Description

The assignment operators place a pointer to a C++ object in the *TAutoObject*. They are usually used to initialize the *TAutoObject* after creating it with the default constructor.

Form 1:
Places the *point* pointer in the *TAutoObject*.

Form 2:
Places a pointer to the object *ref* in the *TAutoObject*.

Form 3:
Attempts to read type information from the object that owns the *IDispatch* interface. If it succeeds, the operator places in the *TAutoObject* a pointer to the C++ object. If it fails, the constructor throws a TXAuto::xTypeMismatch exception.

See Also
TXAuto::TError enum

TAutoObject::operator T&()

TAutoObject

Syntax

```
operator T&();
```

Description

Returns a reference to the object whose pointer *TAutoObject* holds.

TAutoObject::P

TAutoObject

Syntax

```
T* P;
```

Description

Returns the pointer that *TAutoObject* holds.

TAutoObject Public Constructors

See Also TAutoObject

Form 1
```
TAutoObject();
```

Form 2
```
TAutoObject(T* point);
```

Form 3
```
TAutoObject(T& ref);
```

Form 4
```
TAutoObject(IDispatch* dispatch);
```

Description

Form 1: Constructs an empty *TAutoObject* that contains no pointer.

Form 2: Constructs a *TAutoObject* that holds the pointer *point*.

Form 3: Constructs a *TAutoObject* that holds a pointer to the object *ref*.

Form 4: Attempts to read type information from the object that owns the *IDispatch* interface. If it succeeds, the constructor builds a *TAutoObject* around a pointer to the C++ object. If it fails, the constructor throws a TXAuto::xTypeMismatch exception.

See Also
TXAuto::TError enum

TAutoObject::operator TObjectDescriptor()

TAutoObject

Syntax

```
operator TObjectDescriptor();
```

Description

Constructs and returns a new object descriptor object based on the pointer that *TAutoObject* holds. This operator is called internally to obtain type information for constructing an automation object.

TAutoObject::operator T*

TAutoObject

Syntax

```
operator T*();
```

Description

Returns a pointer to the object *TAutoObject* holds.

TAutoObjectByVal<> Class

See Also Automation Classes

Header File
ocf/autodefs.h

Base Class
TAutoObjectDelete

Description
An automation server uses this class when an automated method needs to return a copy of an object. Usually you do not have use the class directly because the automation macros make the proper declarations for you.

To return an object, *TAutoObjectByVal* clones the object by calling its copy constructor. The clone is passed to the automation controller as the return value from some automation command. *TAutoObjectByVal* holds on to the cloned object until the controller releases it. Then it destroys the object by calling its destructor.

In other respects, *TAutoObjectByVal* closely resembles its parent class, TAutoObjectDelete.

Public Data Member
```
void operator =(T obj);
```

Public Constructors
```
TAutoObjectByVal(T obj);
TAutoObjectByVal();
```

See Also

TAutoObjectDelete

Constructing and Exposing a Collection Class

TAutoObjectByVal <> Public Constructors

TAutoObjectByVal

Form 1
```
TAutoObjectByVal();
```

Form 2
```
TAutoObjectByVal(T obj);
```

Description

Form 1: Creates an empty *TAutoObjectByVal*.

Form 2: Creates a *TAutoObjectByVal* that holds a copy of the object *obj*. *T* is the data type passed into the template.

TAutoObjectByVal::<>operator =

TAutoObjectByVal

Syntax

```
void operator =(T obj);
```

Description

This operator creates a new object of type *T* by copying the original object, *obj*. The copy is passed to an automation controller as the return value from an automated method. *T* is the data type passed into the template.

TAutoObjectDelete <> Class

See Also Automation Classes

Header File
ocf/autodefs.h

Base Class
TAutoObject

Description
An automation server uses this class when an automated method needs to return an object to an automation controller. Usually you do not have use the class directly because the automation macros make the proper declarations for you.

Like its parent class TAutoObject, *TAutoObjectDelete* exists in order to hold a pointer to an object and convert it as necessary when the object is passed from server to client through automation calls. The difference between the two classes is that when the automation controller is through with the automated object, *TAutoObjectDelete* informs the connector object that it can let the automated C++ object call its destructor.

Public Constructors
```
TAutoObjectDelete(T& r);
TAutoObjectDelete(T* p);
TAutoObjectDelete();
```

Public Member Functions
```
void operator =(T& r);
void operator =(T* p);
operator TObjectDescriptor();
```

See Also
TAutoObject

TAutoObjectDelete <> Public Constructors

See Also TAutoObjectDelete

Form 1
```
TAutoObjectDelete();
```

Form 2
```
TAutoObjectDelete(T* p);
```

Form 3
```
TAutoObjectDelete(T& r);
```

Description

The *TAutoObjectDelete* constructors do nothing but pass their parameters back to the parent class, TAutoObject.

Form 1: Creates an empty *TAutoObjectDelete* object.

Form 2: Creates a *TAutoObjectDelete* object from a pointer to another object.

Form 3: Creates a *TAutoObjectDelete* object from a reference to another object.

See Also
TAutoObject

TAutoObjectDelete::operator =

TAutoObjectDelete

Syntax

Form 1

```
void operator =(T& r);
```

Form 2

```
void operator =(T* p);
```

Description

Form 1: Tells *TAutoObjectDelete* to hold a pointer to the object referred to by *r*.

Form 2: Tells *TAutoObjectDelete* to hold the pointer *p*.

TAutoObjectDelete::operator TObjectDescriptor()

TAutoObjectDelete

Syntax

```
operator TObjectDescriptor();
```

Description

Returns type information describing the object.

TAutoProxy Class

See Also Automation Classes

Header File
ocf/autodefs.h

Description
An automation controller derives classes from *TAutoProxy* to represent automated OLE objects that it wants to command. To send commands to an automated object, the controller invokes methods on the proxy that represents the object. ObjectComponents connects the proxy to the original so that invoking members of the proxy also invokes members of the automated object.

A proxy object must inherit from *TAutoProxy*. In the derived class, the controller declares one method for each command it wants to send. The declared methods must match the prototypes of the desired commands. To implement these proxy methods, the controller uses three macros: AUTONAMES, AUTOARGS, and AUTOCALL. The macros insert code that calls down to the base class. *TAutoProxy* passes the commands to OLE.

Usually you do not have to call anything in *TAutoProxy* directly. All you have to do is derive your proxy class from *TAutoProxy* and implement the methods with the proxy macros.

To generate proxy classes quickly and easily, use the AUTOGEN.EXE tool in the OCTOOLS directory. AUTOGEN reads the automation server's type library and writes all the necessary headers and source files for your proxy objects.

Public Destructor
```
~TAutoProxy();
```

Public Member Functions
```
void Bind(IUnknown* obj);
void Bind(IUnknown& obj);
void Bind(const GUID& guid);
void Bind(char far* progid);
void Bind(TAutoVal& val);
void Bind(IDispatch* obj);
void Bind(IDispatch& obj);
operator IDispatch&;
operator IDispatch*;
bool IsBound();
long Lookup(char far* name);
long Lookup(const long id);
void Lookup(const char* names, long* ids, unsigned count);
void MustBeBound();
void SetLang(TLangId lang);
void Unbind();
```

Protected Constructor
```
TAutoProxy(TLangId lang);
```

Protected Member Function
```
TAutoVal& Invoke(int attr, TAutoProxyArgs& args, long* ids, unsigned
  named=0);
```

See Also

Automation Proxy Macros

Declaring Proxy Classes

TAutoProxy Public Destructor

TAutoProxy

Destructor

```
~TAutoProxy();
```

Description

Destroys the *TAutoProxy* object.

The constructors are protected because only derived proxy classes should call them.

TAutoProxy::Protected Constructor

See Also TAutoProxy

Syntax

```
TAutoProxy(TLangId lang);
```

Description

Constructs a *TAutoProxy* object and sets the object to use the language identified by the *lang* locale ID.

See Also
TLangID typedef (OWL.HLP)

TAutoProxy::Bind

TAutoProxy

Syntax

Form 1
```
void Bind(IUnknown* obj);
```

Form 2
```
void Bind(IUnknown& obj);
```

Form 3
```
void Bind(const GUID& guid);
```

Form 4
```
void Bind(char far* progid);
```

Form 5
```
void Bind(TAutoVal& val);
```

Form 6
```
void Bind(IDispatch* obj);
```

Form 7
```
void Bind(IDispatch& obj);
```

Description

The *Bind* function attempts to open a channel of communication to the automation server in order to send commands. More specifically, *Bind* requests a pointer to the server's *IDispatch* interface.

Bind is called internally when the object is passed as the return object for another proxy method. Which form of *Bind* is used depends on what information available to identify the server.

Form 1: Binds the proxy object to a server identified by a pointer to its *IUnknown* interface. Throws a *TXOle* exception for failure.

Form 2: Binds the proxy object to a server identified by a reference to its *IUnknown* interface. Throws a *TXOle* exception for failure.

Form 3: Binds the proxy object to a server identified by its globally unique ID (GUID). This is the *clsid* that the server registered for objects of the type you want to control. Throws a *TXOle* exception for failure.

Form 4: Binds the proxy object to a server identified by its *progid*. (This is the GUID that the server registered to identify the application itself.) Throws a *TXOle* exception for failure.

Form 5: Attempts to intrepret the value in the *TAutoVal* union as a reference to an *IDispatch* object and bind to the *IDispatch* directly. Throws a *TXAuto* exception if the object does not support *IDispatch*.

Form 6: Accepts *obj* as the proxy object's server.

Form 7: Accepts *obj* as the proxy object's server.

TAutoProxy::operator IDispatch&()

TAutoProxy

Syntax

```
operator IDispatch&();
```

Description

Returns a reference to the *IDispatch* interface of the proxy object's server.

TAutoProxy::operator IDispatch*()

See Also TAutoProxy

Syntax

```
operator IDispatch*();
```

Description

Returns a pointer to the *IDispatch* interface of the proxy object's server and calls AddRef on the interface pointer.

See Also
TocStorage::AddRef

TAutoProxy::IsBound

TAutoProxy

Syntax

```
bool IsBound();
```

Description

Returns **true** if the server already has a pointer to the *IDispatch* interface of its server and **false** if it does not.

TAutoProxy::Lookup

TAutoProxy

Syntax

Form 1
```
long Lookup(char far* name);
```

Form 2
```
long Lookup(const long id);
```

Form 3
```
void Lookup(const char* names, long* ids, unsigned count);
```

Description

Given the name of a command or an argument, *Lookup* calls the server to ask for the corresponding ID values. Although commands and arguments have names for the convenience of programmers, OLE actually identifies them by numbers. A server must find out the ID number in order to execute the command.

Form 1: Calls the server to get the ID that matches the name.

Form 2: Returns the value passed in as *id*.

Form 3: Looks up a series of names and returns all their IDs at once. *names* and *ids* point to two parallel arrays. *count* gives the number of elements in both arrays. With a single call to OLE, *Lookup* fills the *ids* array with numbers to identify all the names.

TAutoProxy::MustBeBound

TAutoProxy

Syntax

```
void MustBeBound();
```

Description

Throws a TXAuto exception if the *TAutoProxy* object does not have an *IDispatch* interface for its server. *TAutoProxy* calls this method internally before performing actions that assume the object is already bound to the server.

TAutoProxy::SetLang

See Also TAutoProxy

Syntax

```
void SetLang(TLangId lang);
```

Description

Sets the locale ID that the controller will pass to the server with each command. The locale ID tells the server what language the controller is using.

See Also

TlangId typedef (OWL.HLP)

TAutoStack

TAutoProxy::Unbind

See Also TAutoProxy

Syntax

```
void Unbind();
```

Description

Decrements the reference count of the proxy object's server and erases internal references to the server.

See Also:
TAutoProxy::Bind

TAutoProxy::Invoke

See Also TAutoProxy

Syntax

```
TAutoVal& Invoke(int attr, TAutoProxyArgs& args, long* ids, unsigned
  named=0);
```

Description

Sends a command to the automation server. *Invoke* is called by the AUTOCALL macros.

attr describes the type of command being issued and can be a combination of the AutoCallFlag enum values.

The *args* object contains all the values passed as arguments to the command.

ids points to an array of ID values identifying the command and the arguments. There should be one ID value for each element in the *args* array.

names tells how many arguments in *args* are identified by name.

See Also

AutoCallFlag enum

AUTOCALL_xxxx Macros

TAutoShort struct

See Also Automation Data Types

Header File
ocf/autodefs.h

Description
TAutoShort is an automation data type that helps ObjectComponents provide type checking for members of an automated class exposed to OLE. Use *TAutoShort* in an automation definition to identify **short** values.

Public Data Member
```
static TAutoType ClassInfo;
```

See Also
Automation Data Types
Automation Definition Macros

TAutoShort::ClassInfo

TAutoShort

Syntax
```
static TAutoType ClassInfo;
```

Description
The *ClassInfo* member of *TAutoShort* holds information that identifies the **short** data type.

TAutoShortRef struct

See Also Automation Data Types

Header File
ocf/autodefs.h

Description
TAutoShortRef is an automation data type that helps ObjectComponents provide type checking for members of an automated class exposed to OLE. Use *TAutoShortRef* in an automation definition to identify **short** values passed by pointer, not by value.

Public Data Member
```
static TAutoType ClassInfo;
```

See Also

Automation Data Types

Automation Definition Macros

TAutoShortRef::ClassInfo

TAutoShortRef

Syntax

```
static TAutoType ClassInfo;
```

Description

The *ClassInfo* member of *TAutoShortRef* holds information that identifies the data type **short***.

TAutoStack Class

See Also   Automation Classes

Header File
ocf/autodefs.h

Description
TAutoStack processes the command stack that an automation controller sends to an automation server through OLE. The command stack contains a dispatch ID identifying a particular command and a set of VARIANT unions containing all the arguments needed to execute the command.

ObjectComponents interprets the dispatch ID and extracts the proper C++ value from each union. It builds a command object (TAutoCommand) and calls the command's *Execute* method. *TAutoCommand* in turn invokes the methods you have exposed by declaring them and defining them in your automated classes.

The stack also carries a locale ID identifying the language used in the command. ObjectComponents takes the locale into account when interpreting strings it extracts from the stack. If you have provided localization resources, then ObjectComponents translates to the requested language for you.

Usually you do not have to work with *TAutoStack* directly. ObjectComponents automatically passes a stack in to the proper command object for you. The command objects are created by the automation declaration macros.

Public Constructor and Destructor
```
TAutoStack(TServedObject& owner, VARIANT far* stack, TLocaleId locale,int
  argcount, int namedcount, long far* map);
~TAutoStackTAutoStack();
```

Public Member Function
```
TAutoVal& operator [](int index);
```

Public Data Members
```
const int ArgCountTLangId LangId;
int ArgSymbolCount;
int CurrentArg;
TServedObject& Owner;
TAutoSymbol* Symbol;
```

Constant
```
SetValue
```

See Also
Automation Declaration Macros
Localizing Symbol Names
TAutoCommand

TAutoStack Public Constructor and Destructor

See Also TAutoStack

Constructor

```
TAutoStack(TServedObject& owner, VARIANT far* stack, TLocaleId locale,int
  argcount, int namedcount, long far* map);
```

Destructor

```
~TAutoStack();
```

Description

The constructor is called only internally. You should not need to construct your own stack.

owner is the automated object to which the command is directed.

stack points to a series of contiguous unions of type VARIANT. The unions contain values or object references passed in automation commands.

locale is a locale ID describing the language the controller is using.

argcount tells how many arguments follow the dispatch ID in the stack.

namedcount tells how many of the arguments were passed with their names. A controller can pass arguments in any order, and even omit optional arguments, if it identifies the arguments it does pass explicitly by the name the server gives them.

if *namedcount* is greater than zero, then *map* points to an array of ID values corresponding to the argument names passed by the constructor.

map is a table for translating named argument IDs to argument positions.

Destructor

Destroys the *TAutoStack* object.

See Also

Localizing Symbol Names

TAutoVal

TLocaleId typedef

TAutoStack::operator []

See Also TAutoStack

Syntax

```
TAutoVal& operator[](int index);
```

Description

Extracts individual arguments from the command stack for use as C++ function arguments. *index* is a zero-based index into the command's argument list, which follows the order established in the corresponding EXPOSE macro of the automation definition. This operator is called by the command objects generated in the automation declaration.

If index is out of range, the operator throws a TXAuto::xNoArgSymbol exception.

See Also

Automation Declaration Macros

Automation Definition Macros

TXAuto::TError enum

TAutoStack::ArgCount

TAutoStack

Syntax

```
const int ArgCount;
```

Description

Holds the number of arguments passed on the command stack (named or unnamed).

TAutoStack::LangId

See Also TAutoStack

Syntax

```
TLangId LangId;
```

Description

Holds a number that identifies the language the controller is using to send commands.

See Also
TlangID typedef (OWL.HLP)

TAutoStack::ArgSymbolCount

TAutoStack

Syntax

```
int ArgSymbolCount;
```

Description

Holds the number of command arguments exposed to automation.

TAutoStack::CurrentArg

TAutoStack

Syntax

```
int CurrentArg;
```

Description

As ObjectComponents processes the arguments on the stack one by one, this member indexes the current argument. When *CurrentArg* reaches ArgCount, all the arguments have been processed.

TAutoStack::Owner

TAutoStack

Syntax

```
TServedObject& Owner;
```

Description

Refers to the automated object that is processing the command on the stack.

TAutoStack::Symbol

See Also TAutoStack

Syntax

```
TAutoSymbol* Symbol;
```

Description

Points to the command symbol defined by the corresponding EXPOSE macro in the automation definition.

See Also
Automation Definition Macros

TAutoStack::SetValue Constant

TAutoStack

Syntax

```
TAutoStack::SetValue
```

Description

SetValue is a predefined standard dispatch ID. The dispatch ID is a number that identifies a particular command that an automated object can execute. The only two standard dispatch IDs used in ObjectComponents are 0 for an object's default action and -3 for a command that sets the value of a property. *SetValue* is -3.

TAutoString struct

See Also Automation Data Types

Header File
ocf/autodefs.h

Description
An automation server uses *TAutoString* to describe C string types in an automation definition. The member functions of the *TAutoString* structure facilitate copying and assigning string values with minimal memory reallocations when strings are passed back and forth between servers and controllers.

You do not need to use *TAutoString* with C++ *string* objects. For more information, see Automation Data Types.

TAutoString works best with **const** string values. When passed a non-constant string, *TAutoString* must make an internal copy. When the string is **const**, *TAutoString* knows the value will not change and can skip the copying step. The performance improvement is significant.

Public Constructors and Destructor
```
TAutoString(const string& s);
TAutoString(const TAutoString& copy);
TAutoString(TAutoVal& val);
TAutoString(const char far* s = 0);
TAutoString(BSTR s, bool loan);
~TAutoString();
```

Public Member Functions
```
operator int()
TAutoString& operator =(char* s);
TAutoString& operator =(const char far* str);
TAutoString& operator =(const TAutoString& copy);
operator char*();
operator const char far*();
```

Public Data Member
```
static TAutoType ClassInfo;
```

See Also
Automation Data Types

TAutoString::operator int()

TAutoString

Syntax
```
operator int()
```

Description
Returns the length of the string value (as *strlen* would calculate the length).

TAutoString::operator char*()

TAutoString

Syntax

```
operator char*();
```

Description

Returns the object's string value in the form of a non-**const** C-style string. To do this, *TAutoString* must create a new copy of the string. It is faster to assign to a **const** char* where possible.

TAutoString::operator =

TAutoString

Form 1

```
TAutoString& operator =(const char far* str);
```

Form 2

```
TAutoString& operator =(char* s);
```

Form 3

```
TAutoString& operator =(const TAutoString& copy)
```

Description

Form 1: Accepts a C-style **const** string as the new value of the *TAutoString*.

Form 2: Accepts a C-style non-**const** string as the new value of *TAutoString*. Because the string is not constant, *TAutoString* must create a new copy of the string for itself. This makes Form 3 significantly slower than Form 1. Try to pass **const** strings where possible.

Form 3: Sets the value of the *TAutoString* object to be a string copied from another *TAutoString* object.

TAutoString::ClassInfo

TAutoString

Syntax

```
static TAutoType ClassInfo;
```

Description

This static structure holds a number that identifies the data type as a string. All the automation data types hold a similar static identifier so that ObjectComponents can query any of them to determine what they are.

TAutoString::operator const char far*()

TAutoString

Syntax

```
operator const char far*();
```

Description

Returns the object's string value in the form of a **const** C-style string.

TAutoString Public Constructors and Destructor

TAutoString

Form 1
```
TAutoString(const string& s);
```

Form 2
```
TAutoString(const TAutoString& copy);
```

Form 3
```
TAutoString(TAutoVal& val);
```

Form 4
```
TAutoString(const char far* str);
```

Form 5
```
TAutoString(BSTR s, bool loan)
```

Destructor
```
~TAutoString();
```

Description

Form 1: Creates a *TAutoString* and assigns it the string held in a C++ *string* object.

Form 2: Creates a new *TAutoString* that holds the same string value as the *copy* object.

Form 3: Initializes the new object with the value in a TAutoVal union. *TAutoVal* represents the VARIANT data type OLE uses to pass values between two applications. It is a union of many types. This constructor extracts the value from the union as a string.

Form 4: Initializes the new object with the value in a **const** C string.

Form 5: Initializes the new object with the value in a BASIC-style string, one preceded by its length and not terminated by null. That is the format OLE uses for passing strings. Set *loan* to **true** if the *TAutoString* object owns the BSTR and **false** if it only references the BSTR.

Destructor

TAutoString maintains a reference count on the string object it contains. The destructor decrements the reference count.

TAutoType struct

See Also

Header File
ocf/autodefs.h

Description
The *TAutoType* structure is a static data member of all the automation data type classes, such as TAutoBool and TAutoString. *TAutoType* makes all these data types self-describing. This is an essential quality for dealing with the VARIANT unions that OLE uses to pass values during automation. Because all the automation types derive from *TAutoType*, ObjectComponents can process values of any type with the same code. Because *TAutoType* is self-describing, ObjectComponents can always determine the actual type of any particular item.

Usually you do not have to work with *TAutoType* directly, just with the automation types that derive from it.

Public Member Function
```
short GetType();
```

See Also
Automation Data Types

TAutoType::GetType

See Also TAutoType

Syntax
```
short GetType();
```

Description
Returns an integer that identifies a particular data type. The identifiers are defined in the AutoDataType **enum**.

See Also
AutoDataType enum

TAutoVal Class

See Also Automation Classes

Header File
ocf/autodefs.h

Description
TAutoVal duplicates the VARIANT type that OLE uses to pass values between an automation server and controller. It also adds access methods to retrieve the value in the VARIANT. A VARIANT can be cast to type *TAutoVal,* and *TAutoVal* can be cast to a VARIANT.

A VARIANT is a large union with fields of many different data types. A large set of overloaded assignment operators allow many different kinds of values to be stored in a *TAutoVal* object. Each assignment operator also records internally a number that identifies the type of value just received. A similar set of conversion operators allows the value in the object to be cast to different types of values. Whether a particular conversion succeeds depends on the type of value in the object. A *string* cannot be cast to some other object, for example. If the conversion fails, *TAutoVal* throws an exception of type TXAuto::xConversionFailure.

ObjectComponents treats the data passed between an automation server and controller as a stack of unions. The stack is TAutoStack, and the items on the stack are *TAutoVal*. Because the server and controller are built separately and can use different programming languages, data passed between them cannot retain an intrinsic type. Command identifiers and argument values are passed as VARIANTs. The recipient of a VARIANT value must rely on the item's context in order to determine what type the value is supposed to be. For example, when it sees a dispatch ID for a command that expects two integer arguments, the application extracts integers from the next two VARIANTs.

Public Member Functions
```
void operator =(TObjectDescriptor od);
void operator =(IUnknown* ifc);
void operator =(IDispatch* ifc);
void operator =(TAutoVoid);
void operator =(TAutoDate far* i);
void operator =(TAutoDate i);
void operator =(TAutoCurrency i);
void operator =(TAutoCurrency far* p);
void operator =(TAutoString s);
void operator =(string s);
void operator =(const char far* s);
void operator =(TBool far* p);
void operator =(TBool i);
void operator =(double far* p);
void operator =(double i);
void operator =(float far* p);
void operator =(float i);
void operator =(short far* p);
void operator =(short i);
void operator =(unsigned long far* p);
void operator =(unsigned long i);
void operator =(long far* p);
void operator =(long i);
void operator =(int far* p);
void operator =(int i);
void Clear();
void Copy(const TAutoVal& copy);
operator doublefar*();
operator double();
operator floatfar*();
operator float();
int GetDataType();
IDispatch&();
IDispatch*();
operator intfar*();
operator int();
bool IsRef();
operator IUnknown&();
operator IUnknown*();
operator longfar*();
operator long();
operator shortfar*();
operator short();
string();
operator TAutoCurrency();
operator TAutoCurrencyfar*();
operator TAutoDatefar*();
operator TAutoDate();
operator TBoolfar*();
operator TBool();
operator TUString*();
operator unsignedlong();
operator unsignedlongfar;
```

See Also

TAutoStack

TXAuto::TError enum

TAutoVal::operator int()

TAutoVal

Syntax

```
operator int();
```

Description

Returns the value in the object as an integer.

TAutoVal::operator int far*()

TAutoVal

Syntax

```
operator int far*();
```

Description

Returns a pointer to an **int**.

TAutoVal::operator short()

TAutoVal

Syntax

```
operator short();
```

Description

Returns the value in the object as a **short** integer.

TAutoVal::operator short far*()

TAutoVal

Syntax

```
operator short far*();
```

Description

Returns a pointer to a **short** integer.

TAutoVal::operator long()

TAutoVal

Syntax

```
operator long();
```

Description

Returns the value in the object as a **long** integer.

TAutoVal::operator long far*()

TAutoVal

Syntax

```
operator long far*();
```

Description

Returns a pointer to a **long** integer.

TAutoVal::operator unsigned long()

TAutoVal

Syntax

```
operator unsigned long();
```

Description

Returns the value in the object as an **unsigned long** integer.

TAutoVal::operator unsigned long far*()

TAutoVal

Syntax

```
operator unsigned long far*();
```

Description

Returns a pointer to a **long** integer. (*TAutoVal* does not distinguish **long** from **unsigned long**.)

TAutoVal::operator float()

TAutoVal

Syntax

```
operator float();
```

Description

Returns the value in the object as a floating-point value.

TAutoVal::operator float far*()

TAutoVal

Syntax

```
operator float far*();
```

Description

Returns a pointer to a floating-point value.

TAutoVal::operator double()

TAutoVal

Syntax

```
operator double();
```

Description

Returns the value in the object as a **double** value.

TAutoVal::operator double far*()

TAutoVal

Syntax

```
operator double far*();
```

Description

Returns a pointer to a **double** value.

TAutoVal::operator TBool()

TAutoVal

Syntax

```
operator TBool();
```

Description

Returns the value in the object as a Boolean value.

TAutoVal::operator TBool far*()

TAutoVal

Syntax

```
operator TBool far*();
```

Description

Returns a pointer to a Boolean value.

TAutoVal::operator TAutoCurrency()

See Also TAutoVal

Syntax

```
operator TAutoCurrency();
```

Description

Returns the value in the object as a currency value.

See Also
TAutoCurrency

TAutoVal::operator TAutoCurrency far*()

See Also TAutoVal

Syntax

```
operator TAutoCurrency far*();
```

Description

Returns a pointer to a currency value.

See Also
TAutoCurrency

TAutoVal::operator TAutoDate()

See Also TAutoVal

Syntax

```
operator TAutoDate();
```

Description

Returns the value in the object as a date value.

See Also
TAutoDate

TAutoVal::operator TAutoDate far*()

See Also TAutoVal

Syntax

```
operator TAutoDate far*();
```

Description

Returns a pointer to a date value.

See Also
TAutoDate

TAutoVal::operator string()

TAutoVal

Syntax

```
operator string();
```

Description

Returns the value in the object as a C++ *string* object.

TAutoVal::operator TUString*()

TAutoVal

Syntax

```
operator TUString*();
```

Description

Returns the value in the object as a *TUString* object. *TUString* is a reference-counted union of various string representations. It is used internally by ObjectComponents for implementing TAutoString.

TAutoVal::operator IDispatch*()

See Also TAutoVal

Syntax

```
operator IDispatch*();
```

Description

Extracts an *IDispatch* interface from the value in the *TAutoVal* object. *IDispatch* is the standard OLE interface supported by automatable objects. This function also calls the interface's AddRef method.

See Also

TAutoVal::IDispatch&

TocStorage::AddRef

TAutoVal::operator IDispatch&()

See Also		TAutoVal

Syntax

```
operator IDispatch&();
```

Description

Extracts an *IDispatch* interface from the value in the *TAutoVal* object. *IDispatch* is the standard OLE interface supported by automatable objects. This method does not call AddRef on the *IDispatch* interface.

See Also

TAutoVal::IDispatch*

TocStorage::AddRef

TAutoVal::operator IUnknown*()

See Also TAutoVal

Syntax

```
operator IUnknown*();
```

Description

Extracts an *IUnknown* interface from the value in the *TAutoVal* object. *IUnknown* is the standard OLE interface supported by all objects. This method calls *AddRef* on the *IUnknown* interface.

See Also
TAutoVal::IUnknown&

TAutoVal::operator IUnknown&()

See Also TAutoVal

Syntax

```
operator IUnknown&();
```

Description

Extracts an *IUnknown* interface from the value in the *TAutoVal* object. *IUnknown* is the standard OLE interface supported by all objects. This method does not call AddRef on the *IUnknown* interface.

See Also

TAutoVal::IUnknown*

TocStorage::AddRef

TAutoVal::operator =

See Also TAutoVal

Syntax

```
void operator=(int i);
void operator=(int far* p);
void operator=(long i);
void operator=(long far* p);
void operator=(unsigned long i);
void operator=(unsigned long far* p);
void operator=(short i);
void operator=(short far* p);
void operator=(float i);
void operator=(float far* p);
void operator=(double i);
void operator=(double far* p);
void operator=(TBool i);
void operator=(TBool far* p);
void operator=(const char far* s);
void operator=(string s);
void operator=(TAutoString s);
void operator=(TAutoCurrency i);
void far* operator=(TAutoCurrency far* p);
void operator=(TAutoDate i);
void far* operator=(TAutoDate far* i);
void operator=(TAutoVoid);
void operator=(IDispatch* ifc);
void operator=(IUnknown* ifc);
void operator=(TObjectDescriptor od);
```

Description

Assignment operators initialize *TAutoVal* by placing in the object both the assigned value and an ID to show the type of the assigned value.

This table describes those data types that are not standard C types.

Type	Description
IDispatch	A class ObjectComponents uses internally to implement the standard OLE interface called *IDispatch*, supported by automatable objects
IUnknown	A class that ObjectComponents uses internally to implement the standard *IUnknown* OLE interface, supported by all OLE objects
string	C++ *string* object
TAutoCurrency	An automation data type that holds a currency value
TAutoString	An automation data type that holds a C-style string value
TAutoVoid	An automation data type that represents a **void** return
TObjectDescriptor	A class that ObjectComponents uses internally to hold information about an OLE object

See Also

AutomationDataTypes

string (CLASSLIB.HLP)

TAutoVal::GetDataType

TAutoVal

Syntax

```
int GetDataType();
```

Description

Returns an integer identifying the type of value that was assigned to the union.

TAutoVal::Clear

See Also TAutoVal

Syntax

```
void Clear();
```

Description

Clears the value stored in the object, leaving it empty. This method cannot be called on the objects managed by TAutoStack.

See Also
TAutoStack

TAutoVal::Copy

See Also		TAutoVal

Syntax

```
void Copy(const TAutoVal& copy);
```

Description

Copies the *TAutoVal* object into *copy*. Intelligently allocates space for a string, if needed, and calls AddRef if the value in the union is an OLE object.

See Also
TocStorage::AddRef

TAutoVal::IsRef

TAutoVal

Syntax

```
bool IsRef();
```

Description

Returns **true** if the value assigned to the union is a reference to a value.

TAutoVoid struct

See Also Automation Data Types

Header File
ocf/autodefs.h

Description
TAutoVoid is an automation data type like TAutoShort and TAutoBool. Use it in an automation definition to describe functions that return no value.

The purpose of the structure is to implement the assignment of **void** to a TAutoVal.

Public Data Member
```
static TAutoType ClassInfo;
```

See Also

Automation Data Types

TAutoVal Struct

TAutoVoid::ClassInfo

See Also TAutoVoid

Syntax

```
static TAutoType ClassInfo;
```

Description

As with any automation data type, the *ClassInfo* member holds a value that identifies a data type, in this case **void**.

See Also
TAutoType Struct

TComponentFactory Type Definition

See Also General OLE Classes, Macros, and Type Definitions

Header file
ocf/ocreg.h

Syntax
```
typedef IUnknown* (*TComponentFactory)(IUnknown* outer, uint32 options,
  uint32 id = 0);
```

Description
TComponentFactory is a type definition for a callback function.

outer points to the *IUnknown* interface of an external OLE object under which the application is asked to aggregate. If *outer* is 0, then either the new object is independent or it will become the outer object in an aggregation.

options contains bit flags indicating the application's running state. To test the flags, use the TOcAppMode **enum** constants.

id is a number ObjectComponents assigns to identify a particular type of object the application can create. If *id* is 0, the application is asked to create itself. To request particular document types, ObjectComponents passes the document template ID.

The return value is a pointer to the *IUnknown* interface of whatever object the callback function creates, either the application itself or one of its objects. During aggregation, the return value becomes the inner *IUnknown* pointer in some other object. (*IUnknown* is a standard OLE type declared in compobj.h.)

A callback of type *TComponentFactory* is passed to the constructor of an application's registrar object (either TOcRegistrar for a linking and embedding application or TRegistrar for an application that supports automation only). The easiest way to create a factory callback is with a factory template, such as *TOleFactory*<>.

See Also

Factory Template Classes (OWL.HLP)

TOcAppMode enum

TOcRegistrar

TRegistrar

TLocaleId Type Definition

See Also General OLE Classes, Macros, and Type Definitions

Header file
ocf/autodefs.h

Syntax
```
typedef unsigned long TLocaleId;
```

Description
A locale ID is a 32-bit value that identifies a language. The low half of the value is a 16-bit language ID. In the current OLE definition, the upper word is reserved, so in effect a locale ID is a 32-bit language ID.

Windows uses locale IDs to set the system's default language. ObjectComponents uses locale IDs in automation. An automation controller passes a locale ID to the server with every command. The server is expected to interpret the commands it receives as strings in the given language.

There are two predefined system locale settings in the olenls.h header.

Constant	Meaning
LOCALE_SYSTEM_DEFAULT	The default locale set for the system.
LOCALE_USER_DEFAULT	The default locale set for a particular user (which can differ from the system setting on multiuser systems)

See Also
Langxxxx Language ID Constants (OWL.HLP)
Localizing Symbol Names
TLangId typedef (OWL.HLP)

TOcApp Class

See Also   Linking and Embedding Classes

Header File
ocf/ocapp.h

Base Class
TUnknown

Description
TOcApp is an ObjectComponents connector object for a linking and embedding application. It implements the interfaces an application needs for communicating with OLE. Any ObjectComponents application that supports linking and embedding needs to have a *TOcApp* object. Usually the registrar creates it for you when you call *TOcRegistrar->Run*.

Applications that support automation but do not support linking and embedding do not need a *TOcApp* object. They create a TRegistrar instead of a TOcRegistrar.

TOcApp is a COM object and implements the *IUnknown* interface.

Public Member Functions
```
void AddUserFormatName(char far* name, char far* resultName, char far* id =
  0);
bool Browse(TOcInitInfo& initInfo);
bool BrowseClipboard(TOcInitInfo& initInfo);
bool CanClose();
bool Clip(IBPart far* part, bool link, bool embed, bool delay = false);
bool Convert(TOcPart* ocPart, bool b);
bool Drag(IBPart far* part, TOcDropAction inAction, TOcDropAction&
  outAction);
uint EnableEditMenu(TOcMenuEnable enable, IBDataConsumer far* ocview);
void EvActivate(bool active);
void EvResize();
bool EvSetFocus(bool set);
string GetName() const;
TOcNameList& GetNameList();
TOcRegistrar& GetRegistrar();
bool IsOptionSet(uint32 option) const;
bool Paste(TOcInitInfo& initInfo);
bool RegisterClass(const string& progid, BCID classId, bool multiUse);
void RegisterClasses(const TDocTemplate* tplHead =
  ::DocTemplateStaticHead);
virtual void ReleaseObject();
void SetOption(uint32 bit, bool state);
void SetupWindow(HWND frameWnd);
bool TranslateAccel(MSG far* msg);
bool UnregisterClass(const string& progid);
void UnregisterClasses(const TDocTemplate*
  tplHead=::DocTemplateStaticHead);
```

Protected Constructor and Destructor
```
TOcApp(TOcRegistrar& registrar, uint32 options = ULONG_MAX, IUnknown* outer
  = 0, const TDocTemplate* tplHead = ::DocTemplateStaticHead);
~TOcApp();
```

Protected Member Functions
```
uint32 ForwardEvent(int eventId, const void far* param);
```

```
uint32 ForwardEvent(int eventId, uint32 param = 0);
```

See Also

Connector Objects

TOcModule::OcInit

TOcRegistrar::CreateOcApp

TOleFactoryBase (OWL.HLP)

TRegistrar

TUnknown

TOcApp Protected Constructor and Destructor

See Also TOcApp

Syntax

Constructor

```
TOcApp(TOcRegistrar& registrar, uint32 options = ULONG_MAX, IUnknown* outer
  = 0, const TDocTemplate* tplHead = ::DocTemplateStaticHead);
```

Destructor

```
~TOcApp();
```

Description

Usually the creation and destruction of an application's *TOcApp* object are managed by the TOcRegistrar object.

Constructor

The constructor for a *TOcApp* object expands the application's message queue if necessary to accommodate OLE message traffic and builds the application's list of supported Clipboard formats.

registrar is a registration object that processes the command line. Create the registrar first.

options is a set of application mode bit flags. The *TOcApp* object is usually created in the TComponentFactory callback function. The constructor's *options* parameter is the same as the callback's *options* parameter.

outer points to the *IUnknown* interface of the outer object inside which the new application is asked to aggregate itself.

tplHead points to the head of an application's list of document templates. The ObjectWindows Library stores an application's document template list in the global variable *DocTemplateStaticHead*.

Destructor

The *TOcApp* destructor notifies OLE that the application is no longer available.

See Also

TComponentFactory typedef

TOcRegistrar

TOcApp::ReleaseObject

TDocTemplateT<D,V>

TOcApp::AddUserFormatName

See Also TOcApp

Syntax

```
void AddUserFormatName(char far* name, char far* resultName, char far* id =
  0);
```

Description

Call this function to associate a result name with a Clipboard format. The *resultName* parameter describes the data format to users and appears in Help text of the Paste Special dialog box. Use one of the other two parameters to identify the associated Clipboard format. This method is used only if you have a non-standard, private Clipboard format that you want to associate with names used in the Paste Special dialog box.

A custom format must first be entered in the application's registration tables using the REGFORMAT macro. For example:

```
REGFORMAT(0, "DrawingClip", ocrContent, ocrIStorage, ocrGet);
```

"DrawingClip" becomes the ID string that Windows uses internally to identify the custom format. To associate more descriptive strings with the custom format, call *AddUserFormatName*:

```
AddUserFormatName("DrawingPad", "a freehand drawing", "DrawingClip");
```

The name of the "DrawingClip" format is now "DrawingPad". If the user chooses Paste Special when data of this type is on the Clipboard, the name in the dialog box is "DrawingPad". It is perfectly legal for the ID and the name to be the same string.

The result string, "a freehand drawing", typically appears in the Help text during a Paste Special operation.

See Also
Registration Macros (OWL.HLP)

TOcApp::Browse

See Also TOcApp

Syntax

```
bool Browse(TOcInitInfo& initInfo);
```

Description

Displays the Insert Object dialog box allowing the user to choose from available servers to create a new object in the open document. Returns **true** if the user inserts an object and **false** if the user cancels.

Create *initInfo* first by passing to its constructor the view object where the new object will be inserted. *Browse* fills *initInfo* with information about the object. Then use *initInfo* to create a new TOcPart.

See Also

TOcInitInfo

TOcPart

TOcView

TOcApp::BrowseClipboard

See Also TOcApp

Syntax

```
bool BrowseClipboard(TOcInitInfo& initInfo);
```

Description

Displays the Paste Special dialog box showing the available formats for the data currently on the Clipboard, allowing the user to choose what format to paste. Returns **true** if the user pastes data and **false** if the user cancels.

Create *initInfo* first by passing to its constructor the view object where the new object will be inserted. *Browse* fills *initInfo* with information about the object. Then use *initInfo* to create a new TOcPart.

This function is called by TOcView::BrowseClipboard.

See Also

TOcInitInfo

TOcPart

TOcView

TOcApp::CanClose

TOcApp

Syntax

```
bool CanClose();
```

Description

A container calls this function to determine whether it can shut down. *CanClose* polls all the connected servers and attempts to close them. It returns **true** if it is safe to close the application.

TOcApp::Clip

See Also TOcApp

Syntax

```
bool Clip(IBPart far* part, bool link, bool embed, bool delay = false);
```

Description

Copies the currently selected object to the Clipboard. Usually you do not have to call *Clip* directly because TOcView::Copy does it for you.

part points to the linked or embedded object. You can pass an object of type TOcPart for this parameter. (*TOcPart* supports the *IBPart* interface, which is defined in the BOCOLE library.) If *link* and *embed* are both **true**, then other applications can either link or embed the object when they paste it from the Clipboard. Make delay **true** to have ObjectComponents provide delayed rendering of alternate data formats. (Delayed rendering saves memory. For more information, refer to the Clipboard Overview in the API Help file. Look for the topic "Clipboard Operations.")

See Also

TOcPart

TOcView::Copy

TOcApp::Convert

See Also TOcApp

Syntax

```
bool Convert(TOcPart* ocPart, bool activate);
```

Description

Displays the Convert dialog box where the user can alter the aspect or format of a linked or embedded object. *ocPart* points to the object the user wants to modify.

Make *activate* **true** if you want ObjectComponents to activate the object after converting it. Generally *activate* should be **false** if the user has chosen Links from the Edit menu. If the user tries to activate an object whose server is not present, you can offer the option of converting the object to another server, and in that case *activate* should be **true**.

See Also
TOcPart

TOcApp::Drag

See Also TOcApp

Syntax

```
bool Drag(IBPart far* part, TOcDropAction inAction, TOcDropAction&
  outAction);
```

Description

A container calls this function when the user wants to drag one of the container's objects. The first parameter, *part*, is the object the user is trying to drag. Usually this is an object of type TOcPart. (*TOcPart* supports the *IBPart* interface, which is defined in the BOCOLE library.)

inAction combines bit flags indicating possible drag actions the application supports. The flags indicate whether the user can move, copy, or link the object. The value returned in *outAction* contains just one of the action flags indicating what actually did happen.

See Also

TOcDropAction enum

TOcPart

TOcApp::EnableEditMenu

See Also TOcApp

Syntax

```
uint EnableEditMenu(TOcMenuEnable enable, IBDataConsumer far* ocview);
```

Description

An application calls *EnableEditMenu* to find out which of the OLE-related commands on its Edit menu should currently be enabled. The flags combined in *enable* indicate the commands to be tested, and the return value uses the same bit flags to indicate which commands to enable. *ocview* is usually an object of type TOcView. (*TOcView* supports the *IBDataConsumer* interface, which is defined in the BOCOLE library.)

TOleWindow and TOleView call TOcMenuEnable enum in the command enabler functions for the Edit menu.

See Also

TOcMenuEnable enum

TOcView

TOleWindow (OWL.HLP)

TOleView (OWL.HLP)

TOcApp::EvActivate

See Also TOcApp

Syntax

```
void EvActivate(bool active);
```

Description

A container calls this function to tell OLE when its frame window becomes active or inactive. Make *active* **true** if the window was activated or **false** if it was deactivated.

See Also

TOcApp::EvResize

TOcApp::EvSetFocus

TOcApp::EvResize

See Also TOcApp

Syntax

```
void EvResize();
```

Description

A container calls this function to tell OLE when the size of its frame window (the main window) has changed. OLE might need this information to let a server modify its tool bar during in-place editing.

See Also

TOcApp::EvActivate

TOcApp::EvSetFocus

TOcApp::EvSetFocus

See Also TOcApp

Syntax

```
bool EvSetFocus(bool set);
```

Description

A container calls this function to tell OLE that its frame window has either received or yielded the input focus. Make *set* **true** if the window gained the focus or **false** if it lost the focus.

See Also

TOcApp::EvActivate

TOcApp::EvResize

TOcApp::GetName

TOcApp

Syntax

```
string GetName() const;
```

Description

Returns a string object containing the application's name.

TOcApp::GetNameList

See Also TOcApp

Syntax

```
TOcNameList& GetNameList();
```

Description

Returns an array of TOcNameList objects containing the names of all the Clipboard formats the application supports. The TOcView class uses this list when executing the Paste Special command. The list provides the names and Help strings associated with the formats.

See Also
TOcNameList

TOcApp::GetRegistrar()

See Also TOcApp

Syntax

```
TOcRegistrar& GetRegistrar();
```

Description

Returns the application's registrar object. This is the same object passed into the *TOcApp* constructor.

See Also

TOcApp Protected Constructor and Destructor

TOcRegistrar Class

TOcApp::IsOptionSet

See Also TOcApp

Syntax

```
bool IsOptionSet(uint32 option) const;
```

Description

Tests the application mode flags and returns **true** if those set in *option* are set for the application. The application mode flags are defined in the TOcAppMode **enum**.

Registrar objects also have an *IsOptionSet* method. In most cases the two return the same results. In a DLL server, however, the registrar remembers the set of options that the server originally started with, while *TOcApp::IsOptionSet* queries the options for the currently active instance of the DLL.

To check options in a linking and embedding application, you should call the *IsOptionSet* member your application object inherits from TOcModule. TOcModule::IsOptionSet calls *TOcApp::IsOptionSet*.

A automation DLL server that does not support linking and embedding does not have a *TOcApp* object. It can, however, still find its per-instance options by examining the option flags passed to its factory callback.

See Also

TOcApp::SetOption

TOcAppMode enum

TOcModule::IsOptionSet

TRegistrar::IsOptionSet

TOcApp::Paste

See Also TOcApp

Syntax

```
bool Paste(TOcInitInfo& initInfo);
```

Description

Fills *initInfo* with information about the object on the Clipboard. Returns **true** if it succeeds in gathering information and **false** if it fails. The information is needed to create a new TOcPart object.

TOcApp::Paste is called by TOcView::Paste. Usually you don't need to call *TOcApp::Paste* directly yourself.

See Also

TOcInitInfo

TOcView::Paste

TOcApp::RegisterClass

See Also TOcApp

Syntax

```
bool RegisterClass(const string& progid, BCID classId, bool multiUse);
```

Description

Tells OLE that the application is capable of producing objects of a certain type. What objects a server can produce depend on the types of documents it registers.

progid is the registered string that identifies a type of object.

RegisterClasses loops through the application's document templates and calls *RegisterClass* once for each type. The call is made internally and usually you do not need to invoke either function directly.

See Also

TOcApp::RegisterClasses

TOcApp::UnregisterClass

TOcApp::UnregisterClasses

TOcApp::RegisterClasses

See Also TOcApp

Syntax

```
void RegisterClasses(const TDocTemplate* tplHead
  = ::DocTemplateStaticHead);
```

Description

Announces to OLE that the application is running and tells OLE about each type of document the application has registered. The document types are exposed to OLE as kinds of objects the application can produce. *RegisterClasses* tells OLE who you are and what you can make.

tplHead points to the beginning of the application's list of document templates. ObjectWindows stores this list in the global variable *DocTemplateStaticHead*. UnregisterClasses loops through the list of document types and calls UnregisterClass for each one that has a registered *progid*.

RegisterClasses loops through the document structures in *tplHead* and calls RegisterClass once for each type that has a *progid*. The call is made internally, and usually you do not need to invoke either function directly.

See Also
progid Registration Key
TOcApp::RegisterClass
TOcApp::UnregisterClass
TOcApp::UnregisterClasses

TOcApp::ReleaseObject

See Also TOcApp

Syntax

```
virtual void ReleaseObject();
```

Description

ReleaseObject notifies the object that the application's main window is gone. If the application is not serving a client, *ReleaseObject* also decrements the TOcApp object's internal reference count. The object will destroy itself when the count reaches zero. The destructor of TOcModule calls this function.

See Also
TOcModule

TOcApp::SetOption

See Also TOcApp

Syntax

```
void SetOption(uint32 bit, bool state);
```

Description

Modifies the application's running mode flags. *bit* contains bit flags from the TOcAppMode **enum**. If *state* is **true**, *SetOption* turns the flags on. If *state* is **false**, it turns the flags off. You should never have to call this function because ObjectComponents always maintains the mode flags.

See Also

TOcApp::IsOptionSet

TOcAppMode enum

TOcApp::SetupWindow

TOcApp

Syntax

```
void SetupWindow(HWND frameWnd);
```

Description

Tells the *TOcApp* object what window to associate with the application. Usually *frameWnd* is the application's main window. Usually this function is called from the *SetupWindow* function associated with the application's main window.

TOcApp::TranslateAccel

TOcApp

Syntax
```
bool TranslateAccel(MSG far* msg);
```

Description

A container application adds *TranslateAccel* to its Windows message loop if it wants to make a DLL server's accelerator keystrokes available to the user during in-place editing. DLL servers require this cooperation because they do not have message loops of their own, as an .EXE server does.

If you call *TranslateAccel* after the usual call to the Windows API *TranslateAccelerator*, then your own accelerators will have priority if they happen to conflict with the server's.

msg holds a Windows message structure. The return value is **true** if the server translates the accelerator and **false** if it does not.

TOcApp::UnregisterClass

See Also TOcApp

Syntax

```
bool UnregisterClass(const string& progid);
```

Description

Notifies OLE when the application is no longer available to produce objects of a certain type. *progid* is the registered string that identifies a type of object.

UnregisterClasses loops through all the documents the application registered and calls *UnregisterClass* for each one. The destructor of *TOcApp* calls *UnregisterClasses.*

See Also

TOcApp::RegisterClass

TOcApp::RegisterClasses

TOcApp::UnregisterClasses

TOcApp::UnregisterClasses

See Also TOcApp

Syntax

```
void UnregisterClasses(const TDocTemplate* tplHead
  = ::DocTemplateStaticHead);
```

Description

Announces to the system that the application is no longer available for OLE interactions. *tplHead* points to the beginning of the application's list of document templates. ObjectWindows stores this list in the global variable *DocTemplateStaticHead*.

UnregisterClasses loops through the list of document types and calls UnregisterClass for each one that has a registered **progid**. *UnregisterClasses* is called from the *TOcApp* destructor.

See Also
progid Registration Key
TOcApp::RegisterClass
TOcApp::RegisterClasses
TOcApp::UnregisterClass

TOcApp::ForwardEvent

See Also TOcApp

Syntax

Form 1

```
uint32 ForwardEvent(int eventId, const void far* param);
```

Form 2

```
uint32 ForwardEvent(int eventId, uint32 param = 0);
```

Description

Both forms send a WM_OCEVENT message to the application's main window. The *eventId* parameter becomes the message's *wParam* and should be one of the OC_APPxxxx or OC_VIEWxxxx constants. The second parameter becomes the message's *lParam* and can be either a pointer (Form 1) or an integer (Form 2). Which form you use depends on the information a particular event needs to send in its *lParam*.

See Also
WM_OCEVENT message
OC_APPxxxx messages
OC_VIEWxxxx messages
TOcRegistrar class

TOcAppMode enum

See Also Linking and Embedding Enums

Header File
ocf/ocreg.h

Description
The enumerated values of *TOcAppMode* represent flags that ObjectComponents sets to indicate an application's running modes. Some flags are set in response to command-line switches that OLE places on a server's command line. Others are set as the application registers itself.

To determine whether a particular mode flag is set, call TOcApp::IsOptionSet or TOcModule::IsOptionSet. The TOcApp object holds the mode flags for each instance of the application. TOcModule simply queries the *TOcApp*.

The enumerated values are bit flags and can be combined with the bitwise OR operator (|). Flags marked with an asterisk can differ for each instance of an application.

Constant	What the Server Should Do
amAnyRegOption	Combine the *RegServer*, *UnregServer*, and *TypeLib* bits.
amAutomation	Register itself as single-use (one client only). Always accompanied by Embedding.
amDebug	Enter a debugging session.
amExeMode	*Nothing. This flag is set to indicate that the server is running as an .EXE. Either the server was built as an .EXE, or it is a DLL that was launched by an .EXE stub and is running as an executable program.
amExeModule	Nothing. This flag is set to indicate that the server was built as a .EXE program.
amEmbedding	*Consider remaining hidden because it is running for a client, not for itself.
amLangId	Use the locale ID that follows this switch when creating registration and type libraries. (Useless without the *-RegServer* or *-TypeLib* switch.)
amNoRegValidate	Omit the usual validation check comparing the server's *progid*, *clsid*, and *path* to those registered with the system. The registrar object responds to this flag.
amRegServer	Register itself in the system registration database and quit.
amRun	Run its message loop. This is used by the factory callback function.
amServedApp	*Avoid deleting itself (a client is using the application and holds a reference to it).
amShutdown	*When the TComponentFactory callback sees this flag, it should terminate the application.
amSingleUse	*Register itself as a single-use (one client only) application.
amTypeLib	Create and register a type library.
amUnregServer	Remove all its entries from the system registration database and quit.

See Also

TOcModule::IsOptionSet

TOcApp::IsOptionSet

TOcAspect enum

See Also Linking and Embedding Enums

Header File
ocf/ocobject.h

Syntax
```
enum TOcAspect
```

Description
A container uses these values to request that objects in its documents be presented in particular ways. An object might be asked to show all its content, to show a miniature representation of its content, or an icon that represents the type of object it is. A server is not obliged to support all the possible aspects.

The values are flags and can be combined with the bitwise OR operator (|).

Constant	Meaning
asContent	Show the full content of the object at its normal size.
asThumbnail	Show the content of the object shrunk to fit in a smaller space.
asIcon	Show an icon representing the type of object.
asDocPrint	Show the object as it would look if sent to the printer.
asDefault	Continue to use the last aspect specified.
asMaintain	Preserve the object's original aspect ratio. Do not alter the aspect ratio to fit the rectangle where the client chooses to show the object.

See Also

TOcPart::Draw

ocrxxxx Aspect Constants

TOcPartChangeInfo class

See Also

Header File
ocf/ocpart.h

Decsription
The OC_VIEWPARTINVALID event uses this class to carry information to the container's view window when a part needs repainting. The class's two data members tell what part has changed and whether the change affects just the object's data, just its appearance, or both. The member functions set and query the data member values.

Public Constructor
```
TOcPartChangeInfo(TOcPart* part, TOcInvalidate type = invView)
```

Public Member Functions
```
TOcPart* GetPart();
bool IsDataChange();
bool IsViewChange();
void SetDataChange();
void SetPart(TOcPart* part);
void SetViewChange();
```

Protected Data Member
```
TOcPart* Part;
int Type;
```

See Also
OC_VIEWxxxx Messages

TOcPartChangeInfo Public Constructor

See Also TOcPartChangeInfo

Syntax

```
TOcPartChangeInfo(TOcPart* part, TOcInvalidate type = invView)
```

Description

part points to the object that has become invalid. *type* is a bit flag indicating whether the part's data, its appearance, or both are out of date. The possible values are *invView* and *invData*.

See Also

TOcInvalidate enum

TOcPart

TOcPartChangeInfo::GetPart

TOcPartChangeInfo

Syntax

```
TOcPart* GetPart();
```

Description

Returns a pointer to the part object whose data or appearance are out of date.

TOcPartChangeInfo::IsDataChange

See Also TOcPartChangeInfo

Syntax

```
bool IsDataChange();
```

Description

Returns **true** if the part's data has changed and **false** if it has not. The result is determined by testing the *Type* member for the presence of the *invData* flag.

See Also

TOcInvalidate enum

TOcPartChangeInfo::Type

TOcPartChangeInfo::IsViewChange

See Also TOcPartChangeInfo

Syntax

```
bool IsViewChange();
```

Description

Returns **true** if the part's appearance has changed and **false** if it has not. The result is determined by testing the *Type* member for the presence of the *invView* flag.

See Also

TOcInvalidate enum

TOcPartChangeInfo::Type

TOcPartChangeInfo::SetDataChange

See Also		TOcPartChangeInfo

Syntax

```
SetDataChange();
```

Description

Call this member to indicate that the data in the part has changed and needs updating. *SetDataChange* turns on the *invData* bit in the Type member.

See Also

TOcInvalidate enum

TOcPartChangeInfo::Type

TOcPartChangeInfo::SetPart

See Also   TOcPartChangeInfo

Syntax

```
void SetPart(TOcPart* part);
```

Description

Call this function to specify the part that has become invalid

See Also
TOcPartChangeInfo::Part

TOcPartChangeInfo::SetViewChange

See Also TOcPartChangeInfo

Syntax

```
void SetViewChange();
```

Description

Call this member to indicate that the appearance of the part has changed and needs updating. SetDataChange turns on the *invView* bit in the Type member.

See Also

TOcInvalidate enum

TOcPartChangeInfo::Type

TOcPartChangeInfo::Part

See Also TOcPartChangeInfo

Syntax

```
TOcPart* Part;
```

Description

Part holds a pointer to the part object whose view or data has become invalid. *Part* is set by the constructor or by the SetPart member function.

See Also

TOcPartChangeInfo Public constructor

TOcPartChangeInfo::SetPart

TOcPartChangeInfo::Type

See Also TOcPartChangeInfo

Syntax

```
int Type;
```

Description

Type holds bit flags indicating whether the part's data, its appearance, or both have become invalid. *Type* is set by the constructor SetViewChange or the SetDataChange members. The bit flags, defined in the TOcInvalidate **enum**, are *invView* and *invData*.

See Also

TOcInvalidate enum

TOcPartChangeInfo Public constructor

TOcPartChangeInfo::SetDataChange

TOcPartChangeInfo::SetViewChange

TOcDataProvider Class

See Also   Linking and Embedding Classes

Header File
ocdata.h

Description
Along with *TOcLinkView*, *TOcDataProvider* provides support for serving a portion of a document. With the functionality of *TOcDataProvider*, a container can embed or link to a selection in the server. T*OcDataProvider* creates a site for this data provider and connects the part and the site. A typical cut and copy transaction using *TOcDataProvider* is illustrated in the tutorial program, Step17dv.cpp, on your distribution disk.

A *TOcDataProvider* object is created whenever a selection is copied to the clipboard. If a server supports linking to selection,   a moniker (a name an application assigns to the item for linking purposes) is created for the selection. Therefore, in the constructor of TOcDataProvider, the *Rename* function is called.

Public Constructors and Destructors
```
TOcDataProvider(TOcView& ocView, TRegList* regList, IUnknown* outer = 0,
  void* userData, TDeleteUserData callBack);
```

Public Member Functions
```
ulong _IFUNC AddRef();
void Disconnect();
void* GetUserData();
HRESULT _IFUNC QueryInterface(const GUID far& iid, void far*far* iface);
ulong _IFUNC Release();
void Rename();
void SetUserData(void* userData);
```

Protected Members Functions
```
UINT  _IFUNC CountFormats();
HRESULT _IFUNC Draw(HDC dc, const RECTL far* pos, const RECTL far* clip,
  TOcAspect aspect, TOcDraw bd);
HRESULT _IFUNC GetFormat(uint index, TOcFormatInfo far* fmt);
HANDLE _IFUNC GetFormatData(TOcFormatInfo far* fmt);
HRESULT _IFUNC GetPartSize(TSize far* size);
HRESULT _IFUNC Save(IStorage*, BOOL sameAsLoad, BOOL remember);
```

See Also

TOcDocument

TOcView

TOcPart

TOcDataProvider Public Constructor

See Also TOcDataProvider

Syntax

```
TOcDataProvider(TOcView& ocView, TRegList* regList, IUnknown* outer = 0,
  void* userData, TDeleteUserData callBack);
```

Description

Constructs a *TOcDataProvider* object associated with the given server view and creates a site for this remote view. *view* refers to the TOcView object associated with the data provider object. Calls *Rename* to create a moniker, a name an application assigns to the selection for linking purposes.

regList is the registration structure for a particular document. Use the BEGIN_REGISTRATION and END_REGISTRATION macros to create an object of type TRegList.

outer is the root interface of an outer object inside which the new linked view is asked to aggregate itself.

userData contains information the application passes in so that it can remember the selection that is copied. Because *TOcDataProvider* performs delayed rendering of the selection, the application is asked to render the selection at paste, rather than at copy time. However, the selection can change between the time it's copied and then pasted. Therefore, *TOcDataProvider* needs to remember the selection at the time the copying took place.

When its time to render or paint the selection, *TOcDataProvider* gives the *userData* back to the server application. It is the responsibility of the application to reestablish the selection through the use of *userData*.

callBack is a callback function of type *TDeleteUserData*. When the *TOcDataProvider* object is destroyed, it calls this callback function in the server application to give it a chance to clean up the *userData* it passes in.

The *TDeleteUserData* callback function is defined as

```
typedef void (*TDeleteUserData)(void* userData);
```

See Also

TOcDataProvider::Rename

Registration Macros (OWL.HLP)

TOcDocument

TOcView

TOcDataProvider::AddRef

See Also		TOcDataProvider

Syntax

```
ulong _IFUNC AddRef();
```

Description

Increases the reference count on the linked view object. Initializes the reference count to 1. If **this** is not aggregated, *AddRef* returns the *TOcDataProvider* object's reference count. The reference count indicates how many client applications hold pointers to this object.

See Also
TUnknown::GetRefCount

TOcDataProvider::Disconnect

TOcDataProvider

Syntax

```
void Disconnect();
```

Description

Called when the user wants the server to render its data right away. This happens when a cut operation takes place (because the selection no longer exists after the cut is performed) and when a server application shuts down while it has a selection copied on the clipboard.

TOcDataProvider::GetUserData

See Also TOcDataProvider

Syntax

```
void* GetUserData();
```

Description

Returns the user data, which is the information the application uses to remind itself of the attributes of the selection. It is important for *TOcDataProvider* to have this information because the selection might change between the time it is copied and pasted. When it's time to render or paint the selection, *TOcDataProvider* gives the *userData* back to the server application.

See Also
TOcDataProvider::SetUserData

TOcDataProvider::QueryInterface

See Also TOcDataProvider

Syntax

```
HRESULT _IFUNC QueryInterface(const GUID far& iid, void far*far* iface);
```

Description

Asks for the interface identified by *iid* and returns the supported interface through *iface*. If the given interface is not supported, *iface* returns 0.

See Also
TOcLinkView::QueryInterface

TOcDataProvider::Release

TOcDataProvider

Syntax

```
ulong _IFUNC Release();
```

Description

Deletes the object when the reference count reaches zero. Releases the *TOcDataProvider* object, which means that the clipboard becomes the owner of this object. The *TOcDataProvider* object is destroyed whenever the clipboard releases it when something else is copied to the clipboard.

TOcDataProvider::Rename

TOcDataProvider

Syntax

```
void Rename();
```

Description

Establishes the moniker for the server document and then sends an OC_VIEWGETITEMNAME message to TOleWindow. It is the server applications responsibility to return the moniker for the selection or the whole document.

TOcDataProvider::SetUserData

See Also TOcDataProvider

Syntax

```
void SetUserData(void* userData);
```

Description

Sets the *userData*, which is the information the application uses to remind itself of the attributes of the selection. It is important for *TOcDataProvider* to have this information because the selection might change between the time it is copied and pasted. When it's time to render or paint the selection, *TOcDataProvider* gives the *userData* back to the server application. It's the application's responsibility to reestablish the selection through the use of *userData*.

See Also
TOcDataProvider::GetUserData

TOcDataProvider::CountFormats

TOcDataProvider

Syntax

```
UINT _IFUNC CountFormats();
```

Description

CountFormats returns the number of clipboard formats the server application supports.

TOcDataProvider::Draw

See Also		TOcDataProvider

Syntax

```
HRESULT _IFUNC Draw(HDC dc, const RECTL far* pos, const RECTL far* clip,
  TOcAspect aspect, TOcDraw bd);
```

Description

Paints the metafile, sets up the window origin and extent with the proper values,   and sends an OC_VIEWPAINT message to *TOleWindow* to actually paint the selection into the metafile DC. *TOleWindow* uses two data members of the *TOcViewPaint* structure to determine how to paint the metafile: *PaintSelection* and *Moniker*. *PaintSelection* is **true,** only the selection is painted. If *Moniker* is not null, the link view corresponding to this *Moniker* is painted.

See Also
TOleWindowOWL.HLP)

TOcDataProvider::GetFormat

See Also TOcDataProvider

Syntax

```
HRESULT _IFUNC GetFormat(uint index, TOcFormatInfo far* fmt);
```

Description

Returns the data format identified by *index*. This function is primarily used internally to return the supported clipboard formats from the registration table.

See Also
TOcDataProvider::GetFormatData

TOcDataProvider::GetFormatData

See Also TOcDataProvider

Syntax

```
HANDLE _IFUNC GetFormatData(TOcFormatInfo far* fmt);
```

Description

When the user does a paste into the container, this function is called to render native data formats in a memory handle. (The server application needs to render the selection into a handle.) *GetFormatData* also sends an OC_VIEWCLIPDATA message to *TOleWindow*.

See Also

TOcDataProvider::Save

TOcDataProvider::GetFormat

TOleWindowOWL.HLP)

TOcDataProvider::GetPartSize

TOcDataProvider

Syntax

```
HRESULT _IFUNC GetPartSize(TSize far* size);
```

Description

Returns the size of the server document. When a paste operation occurs in a container, the server is asked to render its data through *TOcDataProvider* object on the clipboard. To do this, the container calls *GetPartSize* to get the size of the server document selection. Once the container has the size of the document selection, the *TOcDataProvider* draws the served selection of the document.

TOcDataProvider::Save

See Also TOcDataProvider

Syntax

```
HRESULT _IFUNC Save(IStorage*, BOOL sameAsLoad, BOOL remember);
```

Description

For an embedded source, *Save* renders the selection in *IStorage*. When the user pastes data into the container, *TOcDataProvider* renders the data format in the storage media selected by the user.

The server renders its native data in a handle. To save a selection to the *IStorage* passed in to *TOcDataProvider*, the server needs to send a message: OC_VIEWSAVEPART to *TOleView.* (*TOleView::EvOcViewSavePart* is the handler for this message.) *TOcSaveLoad* lets *TOleView* know whether a selection or whole document needs to be saved.

For native data, *GetFormatData* is called.

See Also
TOcDataProvider::GetFormatData
TOleViewOWL.HLP)

TOcDialogHelp enum

See Also Linking and Embedding Enums

Header File
ocf/ocobject.h

Syntax
```
enum TOcDialogHelp
```

Description
The TOC_APPDIALOGHELP event tells the container when the user clicks the Help button in a standard OLE dialog box. The *lParam* of the WM_OCEVENT message carries one of these values to indicate which dialog box the user has open.

Constant	Dialog Box	Purpose
dhBrowse	Insert Object dialog box	Choose an object to insert.
dhBrowseClipboard	Paste Special dialog box	Choose the data format for pasting an object.
dhConvert	Convert dialog box	Convert an object to work with a different server.
dhBrowseLinks	Links dialog box	Update links to objects.
dhChangeIcon	Change Icon dialog box	Used internally by Insert Object and Paste Special dialog boxes.
dhFileOpen	File Open dialog box	Choose a file to open.
dhSourceSet	Change Source dialog box	Assign a new link source to a linked object.
dhIconFileOpen	File Open dialog box	Confirm that the chosen file contains an icon resource.

See Also
ObjectComponents Messages (OWL.HLP)
OC_APPxxxx messages
TOleFrame::EvOcAppDialogHelp (OWL.HLP)
WM_OCEVENT message

TOcDocument Class

See Also Linking and Embedding Classes

Header File
ocf/ocdoc.h

Description
The primary responsibility of a *TOcDocument* is to save and load data in a compound file using hierarchically ordered storages. (A storage is a compartment within a file, just as a directory is a compartment on a disk.) By default the application's native data always goes in the document's root storage, but the application is free to create its own storages in the same file. *TOcDocument* creates new storages below the root as necessary for OLE objects that the user inserts into the compound document. The new storages take their names from the names of the objects they store. TOcView automatically assigns a unique string identifier to each new object.

Both servers and containers can create objects of type *TOcDocument*. In the container, this object represents an entire compound document. In the server, it represents the data for a single OLE object. (The server's single OLE object can have other OLE objects linked or embedded in it.)

A *TOcDocument* object manages the collection of TOcPart objects that are deposited in one of the container's documents. It does not draw the data on the screen. To do that, every *TOcDocument* needs a corresponding *TOcView* or TOcRemView object. An application can possess multiple pairs of associated document and view objects, one for each open document.

A container creates a *TOcView* object to draw its compound document in the container's own window. Because the window where the server draws belongs to the container (it is a child of the container's window), the server must create a remote view object (*TOcRemView*) for each document.

In spite of the similar names, *TOcDocument* and *TOcView* are not part of the ObjectWindows Doc/View model. The nature of OLE makes it useful to separate data from its graphical representation, and the terms *document* and *view* express that separation even outside of ObjectWindows.

To execute its tasks, a *TOcDocument* must use the standard OLE interfaces *IStorage* and *IStream*. Usually it is not necessary to use these interfaces directly because ObjectComponents implements them for you in its *TOcStorage* and *TOcStream* classes. These classes are thin wrappers around standard OLE interfaces. The implementation of *TOcDocument* makes use of both objects.

Public Constructors and Destructor
```
TOcDocument(TOcApp& app, const char far* fileName = 0);
TOcDocument(TOcApp& app, const char far* fileName, IStorage far* storageI);
~TOcDocument();
```

Public Member Functions
```
void Close();
TOcView* GetActiveView();
string GetName() const;
TOcPartCollection& GetParts();
TOcStorage* GetStorage();
bool LoadParts();
void RenameParts(IBRootLinkable far* BLDocumentI);
bool SaveParts(IStorage* storage = 0, bool sameAsLoaded = true);
bool SaveToFile(const char far* newName);
void SetActiveView(TOcView* view);
void SetName(const string& newName);
void SetStorage(IStorage* storage);
void SetStorage(const char far* path);
```

See Also

Creating ObjectComponents View and Document Objects

TOcPart

TOcRemView

TOcStorage

TOcStream

TOcView

TOcDocument Public Constructors and Destructor

See Also TOcDocument

Form 1
```
TOcDocument(TOcApp& app, const char far* fileName = 0);
```

Form 2
```
TOcDocument(TOcApp& app, const char far* fileName, IStorage far* storageI);
```

Destructor
```
~TOcDocument();
```

Description

Form 1: Creates a new document object for the application and optionally assigns a file name for storing the document. A container uses this constructor for each document the user opens.

Form 2: Creates a new document object for the application and assigns a particular file and storage object to hold the document. The container calls this constructor when opening an existing file. The server and the container each create their own *TOcDocument* object for the object they share, but both their objects point to the same file for storing the object.

IStorage is the standard OLE storage interface. ObjectComponents implements this interface in the *TOcStorage* class. It is usually not necessary to manipulate the *IStorage* interface or the *TOcStorage* class directly in an ObjectComponents application.

Destructor: Destroys the document object.

See Also
TOcStorage

TOcDocument::Close

See Also TOcDocument

Syntax

```
void Close();
```

Description

A container calls TOcPart::Close for each object in the compound document to release its servers. TOleDocument calls this function automatically when asked to close down.

See Also

TOcPart::Close

TOleDocument (OWL.HLP)

TOcDocument::GetActiveView

See Also TOcDocument

Syntax

```
TOcView* GetActiveView();
```

Description

Returns a pointer to the active view. *TOcPart* calls this method to coordinate changing focus among active parts.

See Also

TOcDocument::SetActiveView

TOcDocument::GetName

See Also TOcDocument

Syntax

```
string GetName() const;
```

Description

Returns the name of the file where the document will be stored. ObjectComponents keeps track of the name in order to create links correctly.

See Also
TOcDocument::SetName

TOcDocument::GetParts

See Also TOcDocument

Syntax

```
TOcPartCollection& GetParts();
```

Description

Returns an object with information about all the parts in the document. Each part corresponds to a linked or embedded object. Create an iterator of type *TOcPartCollectionIter* to loop through the collection and extract information about individual parts.

See Also

TOcPart

TOcPartCollection

TOcPartCollectionIter

TOcDocument::GetStorage

See Also TOcDocument

Syntax

```
TOcStorage* GetStorage();
```

Description

Returns the document file's root storage.

See Also
TOcDocument::SetStorage

TOcDocument::LoadParts

See Also TOcDocument

Syntax

```
bool LoadParts();
```

Description

Reads all the linked and embedded parts saved in a compound file. *LoadParts* does not necessarily load all the data from all the parts into memory immediately. The data is needed only if the object is visible.

LoadParts returns **true** if all the parts are read successfully. If no file has yet been assigned to the document, then there is nothing to load and the function still returns **true**. (A document can acquire a file from its constructor, from *SaveToFile*, or from *SetStorage*.)

See Also

TOcDocument Public Constructors and Destructors

TOcDocument::SaveParts

TOcDocument::SaveToFile

TOcDocument::SetStorage

TOcDocument::RenameParts

See Also TOcDocument

Syntax

```
void RenameParts(IBRootLinkable far* BLDocumentI);
```

Description

Call this whenever the name of the document file changes. *RenameParts* updates the internal name stored with each part so that other applications can still link to them correctly.

IBRootLinkable is a custom OLE interface defined in the BOCOLE support library. Objects of type *TOcView* implement this interface, so it is usually not necessary to implement it yourself. Simply pass the document's view object to *RenameParts*.

TOcView calls this function automatically if the view is renamed.

See Also
TOcDocument::SetName
TOcView::Rename

TOcDocument::SaveParts

See Also TOcDocument

Syntax

```
bool SaveParts(IStorage* storage = 0, bool sameAsLoaded = true);
```

Description

Writes all the document's linked and embedded objects to the document's file. *storage* is the root storage in the file. A container's *TOcDocument* creates the storage object when the document is created or the first time it is saved. Find the object by calling *GetStorage*. A server gets the storage object from the container. It is usually not necessary to manipulate the *storage* object directly.

sameAsLoaded should be **true** unless the name of the document file has changed since the last time the document was loaded or saved.

SaveParts returns **true** if all the objects are successfully written to the file.

LoadParts and *SaveParts* are called by the *Open* and *Commit* methods in TOleDocument.

See Also
TOcDocument::GetStorage
TOcDocument::LoadParts
TOcDocument::SaveToFile
TOleDocument (OWL.HLP)

TOcDocument::SaveToFile

See Also TOcDocument

Syntax

```
bool SaveToFile(const char far* newName);
```

Description

Saves the document in the file named by *newName*. Usually a container calls this function when the user chooses File|Save for an unnamed document or File|Save As for any document. *SaveToFile* creates a new storage object and then calls *SaveParts*. It returns **true** if all the linked and embedded parts are successfully saved

See Also
TOcDocument::SaveParts

TOcDocument::SetActiveView

See Also TOcDocument

Syntax

```
void SetActiveView(TOcView* view);
```

Description

A *TOcView* object calls this method when it is activated so that the document can locate the active view. *TOcDocument* communicates only with the active view. The active view sends messages to the corresponding window, perhaps a *TOleView* window. This window is responsible for telling other windows about changes.

See Also

TOcDocument::GetActiveView

TOcDocument::SetName

See Also TOcDocument

Syntax

```
void SetName(const string& newName);
```

Description

Tells the document the name of the file where it will be stored. ObjectComponents needs to know the name in order to create links correctly. More specifically, *SetName* causes ObjectComponents to update the OLE moniker that a link server must provide.

See Also
TOcDocument::GetName

TOcDocument::SetStorage

See Also TOcDocument

Syntax

Form 1
```
void SetStorage(const char far* path);
```

Form 2
```
void SetStorage(IStorage* storage);
```

Description

Assigns the document a storage for writing its data. *storage* becomes the document's root storage. Each linked or embedded object gets its own substorage under the root storage.

Form 1: Creates a compound file using the name in *path* and assigns the root storage of the new file to be the root storage of the document. Usually a container calls this function when the user chooses File|Save for an unnamed document or File|Save As for any document.

Form 2: Assigns *storage* to be the document's root storage. Usually a server calls this function when the container passes it an *IStorage* object. (An *IStorage* object implements the standard OLE interface *IStorage*. Usually it is not necessary to manipulate this object directly.)

See Also
TOcDocument::GetStorage

TOcDragDrop struct

See Also Linking and Embedding Structs

Header File
ocf/ocview.h

Description
Holds information that a view or a window needs in order to accept a drag and drop object. The OC_VIEWDRAG and OC_VIEWDROP messages carry a reference to this structure in their *lParam*s. TOleView and TOleWindow process these messages for you, so you should not need to use *TOcDragDrop* directly unless you are programming without ObjectWindows. For examples of how to process OC_VIEWDRAG and OC_VIEWDROP messages, look at the source code for the EvOcViewDrag and EvOcViewDrop methods in *TOleWindow.*

Public Data Members
```
TOcInitInfo far* InitInfo();
TRect Pos;
TPoint Where;
```

See Also
OC_VIEWxxxx Messages
TOleWindow::EvOcViewDrag (OWL.HLP)
TOleWindow::EvOcViewDrop (OWL.HLP)

TOcDragDrop::InitInfo

See Also TOcDragDrop

Syntax

```
TOcInitInfo far* InitInfo;
```

Description

When carried in an OC_VIEWDROP message, this field describes an object about to be dropped on the view. When carried in an OC_VIEWDRAG message, this field is zero.

See Also

OC_VIEWxxxx Messages

TOcInitInfo

TOcDragDrop::Pos

See Also TOcDragDrop

Syntax

```
TRect Pos;
```

Description

The coordinates in *Pos* indicate the area of the view where the user has dropped an object. The position is given in device coordinates relative to the client area.

See Also
TRect (OWL.HLP)

TOcDragDrop::Where

See Also TOcDragDrop

Syntax

```
TPoint Where;
```

Description

The coordinates in *Where* indicate the point on the view where the mouse released the object. The position is given in client area coordinates.

See Also
TPoint (OWL.HLP)

TOcDropAction enum

See Also Linking and Embedding Enums

Header File
ocf/ocobject.h

Syntax
```
enum TOcDropAction
```

Description
TOcApp::Drag uses these values to describe what actions are allowed and what actions actually occur during a drag and drop operation. The values are flags and can be combined with the bitwise OR operator (|).

Constant	Meaning
daDropCopy	Copy the object to the drop site.
daDropMove	Move the object to the drop site.
daDropLink	Create a link to the object at the drop site.
daDropNone	No action occurred.

See Also
TOcApp::Drag

TOcFormat class

See Also   Linking and Embedding Classes

Header File
ocf/ocview.h

Description
Holds information about one Clipboard format that a particular view supports.

TOcFormat, TOcFormatList, and TOcFormatListIter all work together to manage a list of formats. *TOcFormatList* adds and deletes *TOcFormat* objects from the list. *TOcFormatListIter* enumerates the items in the list whenever the view needs to examine them one by one. Because TOcView creates and maintains this list internally, you do not need to use these classes directly.

When ObjectComponents receives your document registration table, it sees entries for each Clipboard format that the document receives or produces. From these entries, *TOcView* creates a list of objects of type *TOcFormat*, each object representing one format. The view needs this list to know when a Clipboard command or drag and drop operation can succeed. For example, if the user drags a bitmap object over a view that accepts only text, *TOcView* refuses the object and the server adjusts the cursor accordingly.

Public Constructors
```
TOcFormat();
TOcFormat(uint fmtId, char far* fmtName, char far* fmtResultName, uint
  fmtMedium, bool fmtIsLinkable, uint aspect = 1, uint direction = 1);
```

Public Member Functions
```
void operator =(const TOcFormatInfo&);
bool operator ==(const TOcFormat& other);
void Disable(bool disable = true);
uint GetAspect() const;
uint GetDirection() const;
uint GetFormatId() const;
void GetFormatInfo(TOcFormatInfo far& f);
char far* GetFormatName();
uint GetMedium() const;
bool IsDisabled() const;
void SetAspect(uint aspect);
void SetDirection(uint direction);
void SetFormatId(uint id);
void SetFormatName(uint id, TOcApp& ocApp);
void SetFormatName(char far* name, TOcApp& ocApp);
void SetMedium(uint medium);
```

See Also

TOcFormatList

TOcFormatListIter

TOcView

TOcFormat Public Constructors

Sea Also TOcFormat

Constructor

Form 1

```
TOcFormat();
```

Form 2

```
TOcFormat(uint fmtId, char far* fmtName, char far* fmtResultName, uint
  fmtMedium, bool fmtIsLinkable, uint aspect = 1, uint direction = 1);
```

Description

Form 1: Creates an empty *TOcFormat* object (default constructor).

Form 2: Creates a *TOcFormat* object and initializes it with information about a particular Clipboard format.

fmtId is a number (such as CF_TEXT or CF_SYLK) identifying a particular Clipboard format.

fmtName is the name string identifying a Clipboard format.

fmtResultName is a string that describes this format to the user. For standard Clipboard formats, this string comes from the OLEVIEW.RC file. For user-defined formats, it comes from the string supplied in the TOcApp::AddUserFormatName.

fmtMedium is one of the ocrxxxx medium constants specifying the channel the application uses for transferring data in this format (for example, global handle or disk file).

fmtIsLinkable is **true** if the application allows containers to link to data in this format and **false** if it does not.

aspect combines bit flags to specify the presentation aspects the application supports for the data format. The flags are defined as ocrxxxx aspect constants.

direction indicates whether the application can both give and receive data in the given format. Possible values are defined as ocrxxxx direction constants.

See Also

ocrxxxx spect Constants

ocrxxxx Clipboard Constants

ocrxxxx Direction Constants

ocrxxxx Medium Constants

TOcFormat::operator =

TOcFormat

Syntax

```
void operator = (const TOcFormatInfo& formatInfo);
```

Description

Takes information from the *formatInfo* parameter and reinitializes the *TOcFormat* object.

TOcFormat::operator ==

TOcFormat

Syntax

```
bool operator ==(const TOcFormat& other);
```

Description

Returns **true** if *&other* is the same as **this**.

TOcFormat::Disable

TOcFormat

Syntax

```
void Disable(bool disable = true);
```

Description

Marks the format as available or unavailable. Disabling a format in effect removes it temporarily from the view's list. Set *disable* to **true** if you decide not to let the view work with a particular data format.

TOcFormat::GetAspect

See Also TOcFormat

Syntax

```
uint GetAspect() const;
```

Description

Returns a bit flag mask indicating the presentation aspects that the view supports for the data format. The bit flags are defined as ocrxxxx aspect constants.

See Also

ocrxxxx Aspect Constants

TOcFormat::SetAspect

TOcFormat::GetDirection

See Also TOcFormat

Syntax

```
uint GetDirection() const;
```

Description

Returns a value indicating whether the view can both give and receive data in the format. Possible values are defined as ocrxxxx direction constants.

See Also

ocrxxxx Direction Constants

TOcFormat::SetDirection

TOcFormat::GetFormatId

See Also TOcFormat

Syntax

```
uint GetFormatId() const;
```

Description

Returns the number that identifies the data format for the operating system. For standard formats, the number is a constant such as CF_TEXT or CF_BITMAP. For custom formats, it is the ID returned from *RegisterClipboardFormat*.

See Also
ocrxxxx Clipboard Constants
TOcFormat::SetFormatId

TOcFormat::GetFormatInfo

See Also TOcFormat

Syntax

```
void GetFormatInfo(TOcFormatInfo far& formatInfo);
```

Description

Fills the *formatInfo* structure with information about the data format.

See Also
TOcFormat::operator =
TOcFormatInfo

TOcFormat::GetFormatName

See Also TOcFormat

Syntax

```
char far* GetFormatName();
```

Description

Returns the string that names the data format. This is the name that appears in the list box of the Paste Special dialog box.

The strings associated with standard formats ("Bitmap", "DIF") are defined in OLEVIEW.RC. The strings associated with custom formats come from TOcApp::AddUserFormatName.

See Also

TOcApp::AddUserFormatName

TOcFormat::SetFormatName

TOcFormat::GetMedium

See Also TOcFormat

Syntax

```
uint GetMedium() const;
```

Description

Returns a number indicating the channel the application uses for transferring data in this format (for example, global handle or disk file). Possible values are defined as ocrxxxx medium constants.

See Also
ocrxxxx Medium Constants
TOcFormat::SetMedium

TOcFormat::IsDisabled

See Also TOcFormat

Syntax

```
bool IsDisabled() const;
```

Description

Returns **true** if the format is currently disabled. A view disables formats if for any reason the view decides not to support the format after all. Disabling a format signals that the view no longer sends or receives data in that format.   To reenable the format, call Disable.

See Also
TOcFormat::Disable

TOcFormat::SetAspect

See Also TOcFormat

Syntax

```
void SetAspect(uint aspect);
```

Description

aspect is a bit flag mask indicating the presentation aspects that the view supports for the data format. The bit flags are defined as ocrxxxx aspect constants.

See Also
ocrxxxx Aspect Constants

TOcFormat::GetAspect

TOcFormat::SetDirection

See Also TOcFormat

Syntax

```
void SetDirection(uint direction);
```

Description

direction indicates whether the view can both give and receive data in the format. Possible values for *direction* are defined as ocrxxxx direction constants.

See Also

ocrxxxx Direction Constants

TOcFormat::GetDirection

TOcFormat::SetFormatId

See Also TOcFormat

Syntax

```
void SetFormatId(uint id);
```

Description

is identifies the data format for the operating system. For standard formats, *id* is a constant such as CF_TEXT or CF_BITMAP. For custom formats, it is the number returned by RegisterClipboardFormat.

See Also
ocrxxxx Clipboard Constants
TOcFormat::GetFormatId
TClipboard::RegisterClipboardFormat (OWL.HLP)

TOcFormat::SetFormatName

See Also TOcFormat

Syntax

Form 1

```
void SetFormatName(uint id, TOcApp& ocApp);
```

Form 2

```
void SetFormatName(char far* name, TOcApp& ocApp);
```

Description

Both forms of *SetFormatName* retrieve from the format list in TOcApp two strings to describe a particular data format: the user-registered name and the format result string. The retrieved strings are stored internally. Form 1 identifies the format by its ID and Form 2 identifies the format by its Windows-registered name.

The *TOcApp* object keeps a list of all the Clipboard formats the application knows about. Each TOcView keeps a separate list of the Clipboard formats it supports. *SetFormatName* updates strings for individual views from the central application list.

The Windows-registered name is the one passed to RegisterClipboardFormat. For standard Clipboard formats, the name comes from OLEVIEW.RC. The user-registered name can differ from the Windows-registered name for user-defined formats if you have called TOcApp::AddUserFormatName. The user-registered name is the one displayed in Clipboard dialog boxes.

The result string is a longer string describing the format to the user.

See Also

TClipboard::RegisterClipboardFormat (OWL.HLP)

TOcApp::AddUserFormatName

TOcFormat::GetFormatName

TOcFormat::SetMedium

See Also TOcFormat

Syntax

```
void SetMedium(uint medium);
```

Description

Sets the transfer mechanism that an application uses for transferring data in this format (for example, global handle or disk file). Possible values for medium are defined as ocrxxxx medium constants.

See Also

ocrxxxx Medium Constants

TOcFormat::GetMedium

TOcFormatInfo struct

See Also   Linking and Embedding Structs

Header File
ocf/ocobject.h

Description
Holds information describing a Clipboard format. Used for communicating with the BOCOLE support library. In ObjectComponents, each TOcView object keeps a list of the data formats it supports in a TOcFormatList object. The list contains TOcFormat items where the view records its preferences for each data format. The calls that communicate data format information to the support library place the information in *TOcFormatInfo* structures.

Clipboard format data is managed for you inside the TOcApp and *TOcView* objects. Normally you don't need to use this structure directly.

Public Data Members
```
OLECHAR Name[32];
OLECHAR ResultName[32];
WORD Id;
uint Medium;
bool IsLinkable;
```

See Also
TOcApp
TOcFormat
TOcFormatList
TOcView

TOcFormatInfo::Id

See Also TOcFormatInfo

Syntax

```
WORD Id;
```

Description

Holds the number that identifies a Clipboard format. For standard formats, the ID is a constant such as CF_TEXT. For custom formats, it is the value returned by RegisterClipboardFormat.

See Also
TClipboard::RegisterClipboardFormat (OWL.HLP)

TOcFormatInfo::IsLinkable

TOcFormatInfo

Syntax

```
bool IsLinkable;
```

Description

Holds **true** if the view lets data in this format be the source for a linked object and **false** if it does not.

TOcFormatInfo::Medium

See Also TOcFormatInfo

Syntax

```
uint Medium;
```

Description

Indicates the transfer mechanism the view uses to send or receive data in this format. Possible values for *Medium* are defined as ocrxxxx medium constants.

See Also
ocrxxxx Medium Constants

TOcFormatInfo::Name

TOcFormatInfo

Syntax

```
OLECHAR Name[32];
```

Description

Holds a string naming the format. OLECHAR changes to a wide character if UNICODE is defined.

TOcFormatInfo::ResultName

TOcFormatInfo

Syntax
```
OLECHAR ResultName[32];
```

Description
Holds a string describing the format for the user. OLECHAR changes to a wide character if UNICODE is defined.

TOcFormatList Class

See Also Linking and Embedding Classes

Header File
ocf/ocview.h

Description
Manages a list of Clipboard formats that a particular view supports.

TOcFormat, *TOcFormatList*, and TOcFormatListIter all work together to maintain the list of formats. *TOcFormatList* adds and deletes *TOcFormat* objects from the list. *TOcFormatListIter* enumerates the items in the list whenever the view needs to examine them one by one. Because TOcView creates and maintains this list internally, it is usually not necessary for you to use any of these classes directly.

When ObjectComponents receives your document registration table, it sees entries for each Clipboard format that the document receives or produces. From these entries, TOcView creates a list of objects of type *TOcFormat*, each object representing one format. The view needs this list to know when a Clipboard command or drag and drop operation can succeed. For example, if the user drags a bitmap over a view that accepts only text, *TOcView* knows the object cannot be dropped and adjusts the cursor accordingly.

Public Constructor and Destructor
```
TOcFormatList();
~TOcFormatList();
```

Public Member Functions
```
TOcFormat*& operator [](unsigned index);
int Add(TOcFormat* format);
void Clear(int del = 1);
virtual uint Count() const;
int Detach(const TOcFormat* format, int del = 0);
unsigned Find(const TOcFormat* format) const;
int IsEmpty() const;
```

See Also

TOcFormat

TOcFormatListIter

TOcView

TOcFormatList Public Constructors and Destructor

See Also TOcFormatList

Constructor

```
TOcFormatList();
```

Destructor

```
~TOcFormatList();
```

Description

Creates an empty list object. To insert items in the list, call the *Add* method.

Destructor

Deletes all the items in the list.

See Also
TOcFormatList::Add

TOcFormatList::operator []

TOcFormatList

Syntax

```
TOcFormat*& operator [](unsigned index);
```

Description

Retrieves a Clipboard format by its position in the list. If index is 1, for example, the [] returns the second item in the list. The order of items depends on the priority assigned to them when they are registered.

TOcFormatList::Add

See Also TOcFormatList

Syntax

```
int Add(TOcFormat* format);
```

Description

Inserts a new Clipboard format item in the list. Returns 0 for failure and 1 for success.

See Also
TOcFormatList::Clear

TOcFormatList::Clear

See Also TOcFormatList

Syntax

```
void Clear(int del = 1);
```

Description

Removes all the items from the list. If *del* is 1, *Clear* also deletes all the TOcFormat objects.

See Also

TOcFormatList::Add

TOcFormatList::Detach

TOcFormatList::Count

See Also TOcFormatList

Syntax

```
virtual uint Count() const;
```

Description

Returns the number of items in the list.

See Also
TOcFormatList::IsEmpty

TOcFormatList::Detach

See Also TOcFormatList

Syntax

```
int Detach(const TOcFormat* format, int del = 0);
```

Description

Removes one format item from the list. If *del* is 1, then *Detach* also deletes the TOcFormat object.

See Also

TOcFormatList::Add

TOcFormatList::Clear

TOcFormatList::Find

TOcFormatList

Syntax

```
unsigned Find(const TOcFormat* format) const;
```

Description

Searches the list for the object passed as format. If the object is found, then *Find* returns the object's position in the list. (The first position is 0.) If format is not in the list, *Find* returns UINT_MAX.

TOcFormatList::IsEmpty

See Also TOcFormatList

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the list object currently contains no TOcFormat items and 0 if the list is not empty.

See Also
TOcFormatList::Count

TOcFormatListIter Class

See Also Linking and Embedding Classes

Header File

ocf/ocview.h

Description

Enumerates all the Clipboard formats that a particular view supports.

TOcFormat, TOcFormatList, and *TOcFormatListIter* all work together to manage the list of formats. *TOcFormatList* adds and deletes *TOcFormat* objects from the list. *TOcFormatListIter* enumerates the items in the list whenever the view needs to examine them one by one. Because TOcView creates and maintains this list internally, it is usually not necessary for you to use any of these classes directly.

When ObjectComponents receives your document registration table, it sees entries for each Clipboard format that the document receives or produces. From these entries, *TOcView* creates a list of objects of type *TOcFormat*, each object representing one format. The view needs this list to know when a Clipboard command or drag and drop operation can succeed. For example, if the user drags a bitmap object over a view that accepts only text, *TOcView* knows the object cannot be dropped and adjusts the cursor accordingly.

Public Constructor

```
TOcFormatListIter(const TOcFormatList& collection)
```

Public Member Functions

```
TOcFormat* operator ++(int);
TOcFormat* operator ++();
TOcFormat* Current() const;
operator int() const;
void Restart();
void Restart(unsigned start, unsigned stop);
```

See Also

TOcFormat

TOcFormatList

TOcView

TOcFormatListIter::Public Constructor

TOcFormatListIter

Syntax

```
TOcFormatListIter(const TOcFormatList& collection)
```

Description

Constructs an iterator to enumerate the Clipboard formats contained in collection.

TOcFormatListIter::operator ++

TOcFormatListIter

Syntax

Form 1

```
TOcFormat* operator++();
```

Form 2

```
TOcFormat* operator++(int);
```

Description

Form 1: Returns the current format and then advances the iterator to point to the next format (post-increment).

Form 2: Advances the iterator to point to the next format in the list and then returns that format (pre-increment).

TOcFormatListIter::Current

TOcFormatListIter

Syntax

```
TOcFormat* Current() const;
```

Description

Returns the format that the iterator currently points to.

TOcFormatListIter::operator int

TOcFormatListIter

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value in order to test whether the iterator has finished enumerating the collection. If parts remain unenumerated, the operator returns the iterator's current position in the list of parts. If the iterator has reached the end of the list, the operator returns zero.

TOcFormatListIter::Restart

TOcFormatListIter

Syntax

Form 1
```
void Restart();
```

Form 2
```
void Restart(unsigned start, unsigned stop);
```

Description

Form 1: Resets the iterator to begin again with the first format in the list.

Form 2: Resets the iterator to enumerate a subset of the format list, beginning with the object at position start and ending with the object at position stop.

TOcFormatName Class

See Also Linking and Embedding Classes

Header File
ocf/ocapp.h

Description
TOcApp uses this class internally to hold the strings that describe a Clipboard data format such as text or bitmap. *TOcApp* displays these strings in standard OLE dialog boxes such as Paste Link.

Every Clipboard format has three associated pieces of information: an ID value, a name string, and a result name. For standard formats, the ID is a constant such as CF_SYLK. The name string is a short name such as "Sylk." The result name is a longer string that tells the user what pasting this data produces--for example, "a spreadsheet". A *TOcFormatName* object holds all three values for one format.

TOcApp makes a TOcNameList object to hold all the format names it needs. It loads descriptive strings into *TOcFormatName* objects and adds the objects one by one to its name list. Both objects are created and managed inside *TOcApp*. Usually you do not have to manipulate either of them directly.

Public Constructors and Destructor
```
TOcFormatName();
TOcFormatName(char far* fmtName, char far* fmtResultName, char far* id =
  0);
~TOcFormatName();
```

Public Member Functions
```
bool operator ==(const TOcFormatName& other);
const char far* GetId();
const char far* GetName();
const char far* GetResultName();
```

See Also

TOcApp

TOcNameList

TOcFormatName Public Constructors and Destructor

TOcFormatName

Form 1
```
TOcFormatName();
```

Form 2
```
TOcFormatName(char far* fmtName, char far* fmtResultName, char far* id =
  0);
```

Destructor
```
~TOcFormatName();
```

Description

Form 1: Constructs an empty format name object.

Form 2: Constructs a format name object and initializes it with three values that describe a Clipboard format. *fmtName* is the name of the format ("metafile"). *fmtResultName* describes what the user gets by pasting this format ("a Windows metafile picture"). *id* is the value that Windows assigns to identify the format (CF_METAFILEPICT) but expressed as a string of decimal digits ("3").

Destructor

Releases the object.

TOcFormatName::operator ==

TOcFormatName

Syntax

```
bool operator==(const TOcFormatName& other);
```

Description

Returns **true** if *other* is the same object as **this**.

TOcFormatName::GetId

TOcFormatName

Syntax

```
const char far* GetId();
```

Description

Returns a pointer to the string that the system uses to designate the format.

TOcFormatName::GetName

TOcFormatName

Syntax

```
const char far* GetName();
```

Description

Returns a pointer to the name of the format.

TOcFormatName::GetResultName

TOcFormatName

Syntax

```
const char far* GetResultName();
```

Description

Returns the descriptive string that tells the user what pasting data of this format produces.

TOcInitHow enum

See Also Linking and Embedding Enums

Header File
ocf/ocobject.h

Syntax
```
enum TOcInitHow
```

Description
These values tell whether a container is to link or embed a new object it is receiving. The container passes this information to a TOcInitInfo object when it receives a new OLE object.

Constant	Meaning
ihLink	Link to the object. Create a reference in the container's document that points to the place in the server's document where the data actually resides.
ihEmbed	Embed the object. Copy the object's data directly into the container's document.
ihMetafile	Embed a static object that draws itself as a metafile.
ihBitmap	Embed a static object that draws itself as a bitmap.

See Also

TOcInitInfo Public Constructors

TOcInitWhere enum

TOcInitInfo Class

See Also   Linking and Embedding Structs

Header File
ocf/ocobject.h

Description

TOcInitInfo holds information that tells ObjectComponents how to create a new part. When the user pastes, inserts, or drops an object into a container, ObjectComponents creates a *TOcInitInfo* object, initializes it with information about the incoming OLE object, and passes the info object to the TOcPart constructor. The info object tells the part where to find its data and how to create itself.

If you are using ObjectWindows, TOleWindow manages these details for you. If you are programming without ObjectWindows, you can find sample code for using *TOcInitInfo* objects in the TOleWindow methods that insert objects: look at the code for CmEditInsertObject and CmEditPasteSpecial.

Public Data Members
```
IBContainer far* Container;
HICON HIcon;
TOcInitHow How;
IStorage far* Storage;
TOcInitWhere Where;
union {
  IDataObject* Data;
  LPCOLESTR Path;
  BCID CId;
  struct{
    HANDLE Data;
    uint DataFormat;
  } Handle;
};
```

Public Constructors
```
TOcInitInfo(IBContainer far* container);
TOcInitInfo(TOcInitHow how, TOcInitWhere where, IBContainer far*
  container);
```

Public Member Functions
```
uint32 ReleaseDataObject();
```

See Also

TOcPart

TOleWindow::CmEditInsertObject (OWL.HLP)

TOleWindow::CmEditPasteSpecial (OWL.HLP)

TOcInitInfo::Container

See Also TOcInitInfo

Syntax

```
IBContainer far* Container;
```

Description

Container is the view object that is about to receive the object. *IBContainer* is an undocumented custom OLE interface defined in the BOCOLE support library and implemented in TOcView. The *Container* data member can hold an object of type TOcView.

See Also
TOcView

TOcInitInfo::HIcon

TOcInitInfo

Syntax

```
HICON HIcon;
```

Description

HIcon holds the icon to draw if the user chooses the Display As Icon option from the Insert Object dialog box. The *HIcon* handle is actually a global memory handle to a metafile containing the icon. The Browse and BrowseClipboard functions in TOcApp handle the Insert Object dialog box for you, so usually you do not need to display the icon directly yourself.

TOcInitInfo::How

See Also TOcInitInfo

Syntax

```
TOcInitHow How;
```

Description

Tells whether the object should be linked or embedded when it is added to the document.

See Also
TOcInitHow enum

TOcInitInfo::Storage

TOcInitInfo

Syntax

```
IStorage far* Storage;
```

Description

Storage is the storage object in a compound file. The container provides the storage to hold data transferred from the server. *IStorage* is a standard OLE interface. ObjectComponents implements the *IStorage* interface in TOcStorage, so *Storage* usually holds a *TOcStorage* object.

TOcInitInfo::Where

See Also TOcInitInfo

Syntax

```
TOcInitWhere Where;
```

Description

Tells where the server will place the object's data. For example, the server can choose to transfer data by placing it in a file, in a storage, or in a memory handle.

See Also
TOcInitWhere enum

TOcInitInfo::CId

See Also TOcInitInfo

Syntax

```
BCID CId;
```

Description

One of four data fields in an anonymous union, this field is used when Where is *iwNew* indicating that the incoming object is brand new, being freshly created. *CId* is the class ID that the server registered for one of its document factories. It tells the server what kind of object to create.

See Also

TOcApp::RegisterClasses

TOcInitInfo::Where

TOcInitInfo::Data

TOcInitInfo

Syntax

```
IDataObject* Data;
```

Description

One of four data fields in an anonymous union, this field is used when Where is *iwDataObject* indicating that the server has created an OLE data object to transfer the data for the incoming object. *Data* points to the *IDataObject* interface on the server's data transfer object. (*IDataObject* is a standard OLE interface.) This is the normal transfer method for objects received from the Clipboard or through a drag-and-drop operation.

TOcInitInfo::Handle

TOcInitInfo

Syntax

```
struct{
  HANDLE Data;
  uint DataFormat;
} Handle;
```

Description

One of four data fields in an anonymous union, this structure is used when Where is *iwHandle* indicating that the server has placed the data for the incoming object in a memory handle. *Data* is the handle itself and *DataFormat* identifies a Clipboard format for the data in the handle.

TOcInitInfo::Path

TOcInitInfo

Syntax

```
LPCOLESTR Path;
```

Description

One of four data fields in an anonymous union, this field is used when *Where* is *iwFile* indicating that the server has placed the data for the incoming object in a file. *Path* points to the name of the file where the data is stored.

TOcInitInfo Public Constructors

TOcInitInfo

Form 1
```
TOcInitInfo(IBContainer far* container);
```

Form 2
```
TOcInitInfo(TOcInitHow how, TOcInitWhere where, IBContainer far*
  container);
```

Description

Both forms of the constructor create a *TOcInitInfo* object for placing a new part in *container*. *container* is the view that will hold the new part. *IBContainer* is a custom OLE interface defined in the BOCOLE support library and implemented in TOcView. *container*can be an object of type *TOcView.*

Form 1: Use Form 1 when invoking the server to create a new object from scratch--for example, when processing the Insert Object command. The new part will be embedded, not linked.

Form 2: Use Form 2 when creating a part to hold an object that already exists--for example, when loading a part from a storage in a compound document. *how* tells whether the object will be linked or embedded. *where* tells what medium the server will use to transfer data from the existing object.

TOcInitInfo::ReleaseDataObject

See Also TOcInitInfo

Syntax

```
uint32 ReleaseDataObject();
```

Description

If the *TOcInitInfo* object holds a pointer to the data object from which the new part is about to be created, then *ReleaseDataObject* decrements the data object's reference count. Call this when you are through with the data object.

See Also
TOcInitInfo::Data

TOcInitWhere enum

See Also Linking and Embedding Enums

Header File
ocf/ocobject.h

Syntax
```
enum TOcInitWhere
```

Description
These values tell where the data for an object resides. A container passes this information to a TOcInitInfo object when it receives a new OLE object for linking or embedding. The server can choose any of several available channels for transferring the data in the object.

Constant	Meaning
iwFile	The server passes the data in a disk file.
iwStorage	The server passes the data in a storage object (part of a compound file).
iwDataObject	The server passes the data in a data transfer object, one that supports the standard *IDataObject* OLE interface. (Objects transferred through the Clipboard or by dragging support this interface. *TOcInitInfo* holds a pointer to the interface.)
iwNew	The server will be asked to create a new object.
iwHandle	The server passes a memory handle for the data.

See Also

TOcInitInfo Public Constructors

TOcInitHow enum

TOcInvalidate enum

See Also		Linking and Embedding Enums

Header File
ocf/ocobject.h

Syntax
```
enum TOcInvalidate
```

Description
Functions that invalidate an object use these enumeration values to indicate whether the data in the object has changed or the appearance of the object has changed. It is possible for the data in an object to change without invalidating the view of the object. For example, if the object is drawn as an icon, then editing the data probably does not call for an update to the view. If both the data and the view change, then combine both flags with the bitwise OR operator (|).

If the view is invalid, the object needs to be redrawn. If the data is invalid, then the object needs saving. (It is not necessary to save the object right away. *invData* simply indicates that the object is dirty and needs to be saved before the document is closed.)

Constant	Meaning
invData	The data in an object has changed and should be updated in the container.
invView	The appearance of an object needs to change and should be updated in the container.

See Also
TOcRemView::Invalidate
TOleWindow::InvalidatePart (OWL.HLP)

TOcLinkView Class

See Also Linking and Embedding Classes

Header File
oclink.h

Description

Derived from *TUnknown*, *TOcLinkView* provides linking support fo a portion of or an entire document. *TOcLinkView* is a connector object located between the server application and the container that helps containers establish links to the server document. *TOcLinkView* is similar to TOcRemView in that they both comminicate with the container through OLE interfaces. Application specific operators, such as saving the selection and painting the selection, are deligated to an OWL OLE class TOLeLinkView.

The main purpose of a class derived from *TOcLink*View is to attach a view to a portion of a document whenever a link is created to a selection within a server document. After this link is established, the container receives change notification messages via the following sequence of steps:

1. When *TOleLinkView* receives a notification message, it checks to see if the selection it represents has changed. If the selection has changed, *TOleLinkView* notifies *TOcLinkView* about the change by calling TOcLinkView::Invalidate() method.
2. When *TOcLinkView TOcLinkView::Invalidate* in turn notifies the container through OLE interfaces that the selection has changed..

Public Constructor

```
TOcLinkView(TOcView* ocView, TRegList* regList = 0, IUnknown* outer = 0);
```

Public Member Functions

```
ulong _IFUNC AddRef();
int Detach();
void Disconnect();
void GetLinkRect();
TString& GetMoniker();
void Invalidate(TOcInvalidate);
HRESULT _IFUNC QueryInterface(const GUID far& iid, void far*far* iface)
ulong _IFUNC Release();
void SetMoniker(const char far* name);
```

Protected Member Functions

```
HRESULT _IFUNC Close();
uint _IFUNC CountFormats();
HRESULT _IFUNC DoQueryInterface(const IID farpif);
HRESULT _IFUNC Draw(HDC, const RECTL far*, const RECTL far*, TOcAspect,
  TOcDraw bd);
HRESULT _IFUNC GetFormat(uint index, TOcFormatInfo far* fmt);
HANDLE _IFUNC GetFormatData(TOcFormatInfo far*);
HRESULT _IFUNC GetPartSize(TSize far*);
HRESULT _IFUNC Open(bool open);
HRESULT QueryObject(const IID far& iid, void far* far* iface);
HRESULT _IFUNC SetPartPos(TRect far*);
HRESULT _IFUNC SetPartSize(TSize far*);
```

See Also
TOcView
TOcRemView
TOleView

TOcLinkView Public Constructor

See Also TOcLinkView

Syntax

```
TOcLinkView(TOcView* ocView, TRegList* regList = 0, IUnknown* outer = 0);
```

Description

Constructs a *TOcLinkView* object and creates a site for this remote view. *view* refers to the TOcView object that corresponds to the link view. *TOcView* manages the appearance of the document on the screen, and *TOcLinkView* manages the appearance of the linked view.

regList is the registration structure for a particular document. Use the BEGIN_REGISTRATION and END_REGISTRATION macros to create an object of type TRegList.

outer is the root interface of an outer object inside which the new linked view is asked to aggregate itself.

See Also

Registration Macros (OWL.HLP)

TAutoObject

TOcDocument

TOcLinkView::AddRef

See Also TOcLinkView

Syntax

```
ulong _IFUNC AddRef();
```

Description

Increases the reference count on the linked view object. Initializes the reference count to 1. If **this** is not aggregated, then returns the *TLinkView* object's reference count. The reference count indicates how many client applications hold pointers to this object, and have linked views to this object.

See Also
TUnknown::GetRefCount

TOcLinkView::Detach

See Also TOcLinkView

Syntax

```
int Detach();
```

Description

Detaches the link view from its document and releases the site. Call Detach before cutting and pasting a document to the clipboard.

See Also
TOcPart::Detach

TOcLinkView::Disconnect

TOcLinkView

Syntax

```
void Disconnect();
```

Description

Disconnects this link view from the client application's site (the area the container has alloted for displaying the linked object)..

TOcLinkView::GetLinkRect

See Also TOcLinkView

Syntax

```
void GetLinkRect();
```

Description

Gets the initial size and position of the link view from the application. This is the actual area where the container application can draw the linked object.

See Also
TOcRemView::GetInitialRect

TOcLinkView::GetMoniker

See Also TOcLinkView

Syntax

```
TString& GetMoniker();
```

Description

Returns the moniker (the source file's path name and the object hierarchy) for the selection in a container document associated with this *TOcLinkView* server's view.

By looking at the moniker, the application can find the corresponding objects in its document. A moniker functions much like a map in that it shows where the linked object's data is stored and explains how to find the data. For example, the moniker returned from a word processor for a selected range of text could be the start and end offset of a text stream. The moniker returned for a spreadsheet range could be something like A1:D6. Anything that a server application can use to map to its data can be used as a moniker.

See Also
TOcLinkView::SetMoniker

TOcLinkView::Invalidate

See Also TOcLinkView

Syntax

```
void Invalidate(TOcInvalidate);
```

Description

Invalidates the container's site corresponding to this remote view. Notifies the container's active view that the server has changed either the appearance or the content of the active linked view.

See Also
TOcRemView::Invalidate

TOcLinkView::QueryInterface

See Also TOcLinkView

Syntax

```
HRESULT _IFUNC QueryInterface(const GUID far& iid, void far*far* iface)
```

Description

Asks whether the link view object supports the interface identified by iid. If the object supports the interface, QueryInterface returns HR_NOERROR and places a pointer to the interface in *pif*.

See Also
TUnknown::QueryObject

TOcLinkView::Release

TOcLinkView

Syntax
```
ulong _IFUNC Release();
```

Description
Releases the server's link views. Deletes the object when the reference count reaches zero.

TOcLinkView::SetMoniker

See Also		TOcLinkView

Syntax

```
void SetMoniker(const char far* name);
```

Description

Establishes the moniker for this server document's link viewer object.

See Also
TOcLinkView::GetMoniker

TOcLinkView::Close

TOcLinkView

Syntax

```
HRESULT _IFUNC Close();
```

Description

Disconnects the linked view from from its container.

TOcLinkView::CountFormats

TOcLinkView

Syntax

```
uint _IFUNC CountFormats();
```

Description

CountFormats returns the number of clipboard formats the server supports.

TOcLinkView::DoQueryInterface

See Also TOcLinkView

Syntax

```
HRESULT _IFUNC DoQueryInterface(const IID farpif);
```

Description

Call *DoQueryInterface* on the linked view to do a *QueryInterface* on the actual server that *the view supports*. This function gives sites the ability to ask for server interfaces.

See Also
TOcLinkView::QueryInterface

TOcLinkView::Draw

See Also TOcLinkView

Syntax

```
HRESULT _IFUNC Draw(HDC, const RECTL far*, const RECTL far*, TOcAspect,
  TOcDraw bd);
```

Description

Draws the linked server objects on the screen.

See Also
TOcPart::Draw

TOcLinkView::GetFormat

See Also TOcLinkView

Syntax

```
HRESULT _IFUNC GetFormat(uint index, TOcFormatInfo far* fmt);
```

Description

GetFormat is mainly used internally to return a clipboard format with the given index. The index is the clipboard format number used in the registration table.

See Also
TOcFormat

TOcLinkView::GetFormatData

TOcLinkView

Syntax

```
HANDLE _IFUNC GetFormatData(TOcFormatInfo far*);
```

Description

Request native data for pasting into client application. This function is called only at paste time and not at copy time.

ect and/or the application.

TOcLinkView::GetPartSize

TOcLinkView

Syntax

```
HRESULT _IFUNC GetPartSize(TSize far*);
```

Description

Gets the size of the part object.

TOcLinkView::Open

TOcLinkView

Syntax

```
HRESULT _IFUNC Open(bool open);
```

Description

If *open* is true, *Open* invokes the server to initiate an out-of-place editing session. That is, it asks the server to execute its Open verb. If *open* is false, the function tells the server to hide its open editing window but does not end the session.

TOcLinkView::QueryObject

See Also TOcLinkView

Syntax

```
HRESULT QueryObject(const IID far& iid, void far* far* iface);
```

Description

Does queryInterface on its aggregated helper objects.

See Also

TUnknown::QueryObject

TOcLinkView::SetPartPos

TOcLinkView

Syntax

```
HRESULT _IFUNC SetPartPos(TRect far*);
```

Description

Sets the position of the selections.

TOcLinkView::SetPartSize

TOcLinkView

Syntax

```
HRESULT _IFUNC SetPartSize(TSize far*);
```

Description

Sets the size of the selection.

TOcMenuDescr struct

See Also Linking and Embedding Structs

Header File
ocf/ocapp.h

Description
The menu descriptor structure is used when merging the menus of a container and server for in-place editing. The structure holds a handle to a shared Windows menu object and a count of the number of drop-down menus in each group.

If you are using ObjectWindows, use the information in the structure to construct a *TMenuDescr* object for the other application. To merge two menus, call TMenuDescr::Merge. If you are not using ObjectWindows, call the Windows API routines such as *InsertMenu* to place your own commands in the shared menu.

The following messages carry a *TOcMenuDescr* **struct** in their *lParam*s: OC_APPINSMENUS, OC_APPMENUS, and OC_VIEWINSMENUS. The ObjectWindows OLE-enabled window and view classes process these messages for you. Unless you are programming without ObjectWindows, you usually will not have to use *TOcMenuDescr* directly. For examples of how to process the messages, see the source code for the relevant event handlers in TOleView, TOleWindow, TOleFrame, and TOleMDIFrame.

Public Data Members
```
HMENU HMenu;
int Width[6];
```

See Also
OC_APPxxxx Messages
OC_VIEWxxxx Messages
TMenuDescr (OWL.HLP)

TOcMenuDescr::HMenu

TOcMenuDescr

Syntax

```
HMENU HMenu;
```

Description

Holds a handle to the shared menu. The handle is valid only while the menu is constructed. Do not store it for later use.

TOcMenuDescr::Width

See Also TOcMenuDescr

Syntax

```
int Width[6];
```

Description

The *Width* array contains the number of pop-up menus in each menu group. The groups, in order, are File, Edit, Container, Object, Windows, and Help.

The array is meant to help you construct a object. The numbers it holds control how the menu is merged.

See Also
TMenuDescr Public Constructors and Destructor

TOcMenuEnable enum

See Also		Linking and Embedding Enums

Syntax

```
enum TOcMenuEnable
```

Description

These enumeration values are flags that can be combined with the bitwise OR operator (|). A container passes them to the TOcApp::EnableEditMenu function in order to determine which OLE commands on the Edit menu should be enabled. The answer depends on whether the container supports any of the data formats currently present on the Clipboard.

Constant	Menu Command Enabled
meEnablePaste	The Paste command places an object from the Clipboard in the open document. The format of the new data object depends on what the server prefers and the container supports.
meEnablePasteLink	The Paste Link command adds to the open document a link to the object on the Clipboard.
meEnableBrowseClipboard	The Paste Special command invokes a standard dialog box that shows all the data formats available for the object currently on the Clipboard and lets the user choose among them.
meEnableBrowseLinks	The Links command displays a list of all the linked objects in the open document, allowing the user to update or delete them.

See Also

Registration Macros (OWL.HLP)

TOcApp::EnableEditMenu

TOcModule class

See Also Linking and Embedding Classes

Header File
ocf/ocapp.h

Description
TOcModule is a mix-in class for deriving OLE-enabled application classes. Any ObjectComponents application that supports linking and embedding should derive its application class from both TApplication and *TOcModule*. The ObjectComponents module class coordinates some basic housekeeping chores related to registration and memory management. It also holds a pointer to the TOcApp object that connects your application object to OLE through ObjectComponents. Allowing *TOcModule* to do this work also makes it easy to use the same code for both .EXE and .DLL versions of the same server.

Public Constructor and Destructor
```
TOcModule();
~TOcModule();
```

Public Member Functions
```
TRegistrar& GetRegistrar();
bool IsOptionSet(uint32 option) const;
void OcInit(TOcApp* ocApp, uint32 options = ULONG_MAX);
```

Public Data Members
```
TOcApp* OcApp;
TOleAllocator OleMalloc;
```

See Also

TApplication

TOcApp

TOcModule Public Constructor and Destructor

See Also TOcModule

Constructor

```
TOcModule();
```

Destructor

```
~TOcModule();
```

Description

Builds a *TOcModule*. After creating a *TOcModule* object, you need to call OcInit.

Destructor

Releases the TOcApp object. An application that derives from *TOcModule* does not need to call the TOcApp::ReleaseObject method when it closes down.

Note: Never call **delete** to destroy a *TOcApp* object.

See Also

TOcModule::OcInit

TOcModule::GetRegistrar

See Also TOcModule

Syntax

```
TRegistrar& GetRegistrar();
```

Description

Returns the application's registrar object. Be sure to call OcInit first.

See Also

TOcModule::OcInit

TRegistrar

TOcModule::IsOptionSet

See Also TOcModule

Syntax

```
bool IsOptionSet(uint32 option) const;
```

Description

Returns **true** if the command-line flag indicated by *option* is set and **false** if it is not. The registrar sets the flags for you when it interprets OLE-related switches on the application's command line. The possible values for *option* are enumerated in TOcAppMode.

TOcModule::IsOptionSet internally queries the TOcApp object it created in TOcModule::OcInit. The registrar object also holds mode flags. Usually the registrar and the *TOcApp* object hold the same set of flags, but in a DLL server the registrar holds the server's original flags and the *TOcApp* holds the flags for the currently active instance of the DLL.

See Also

TOcAppMode enum

TOcApp::IsOptionSet

TOcAppMode enum

TOcModule::OcInit

TRegistrar::IsOptionSet

TOcModule::OcInit

See Also TOcModule

Syntax

```
void OcInit(TOcRegistrar& registrar, uint32 options);
```

Description

Initializes ObjectComponents support for the code module. This call causes ObjectComponents to create the TOcApp connector object that attaches the application to the OLE system. Always call TOcModule::IsOptionSet right after constructing the module object.

registrar is the application registrar object. It must be created before you call *OcInit*.

options is a set of bit flags describing command-line options set for this instance of the program. To test for particular options, call IsOptionSet. The possible option flags are defined in TOcAppMode.

See Also

TOcApp

TOcAppMode enum

TOcModule::IsOptionSet

TOcRegistrar

TOcModule::OcApp

See Also TOcModule

SyntaxModule

```
TOcApp* OcApp;
```

Description

Holds the TOcApp object that is the ObjectComponents partner object for your TApplication-derived class. This member is initialized when you call OcInit.

See Also

TOcApp

TOcModule::OcInit

TOcModule::OleMalloc

See Also TOcModule

Syntax

```
TOleAllocator OleMalloc;
```

Description

Sets up an allocator object that initializes the OLE system and sets up the memory allocator. OLE allows each program to set up a memory manager for OLE to use when allocating and de-allocating memory on behalf of that application.

TOcModule simply chooses the default allocator. If you have unusual memory management needs and want to supply your own custom memory allocator, set its *IMalloc* interface in *OleMalloc*::*Mem*.

See Also

TOleAllocator

TOleAllocator::Mem

TOcNameList Class

See Also Linking and Embedding Classes

Header File

ocf/ocapp.h

Description

TOcApp uses this class internally to manage a collection of TOcFormatName. Each format name object holds three strings that describe a Clipboard data format such as text or bitmap. *TOcApp* displays these strings in standard OLE dialog boxes such as Paste Link.

The list of format names is created and managed inside *TOcApp*. Usually you do not have to manipulate the list directly. To put your own custom formats in the list, however, you do have to register them. See TOcApp::AddUserFormatName for more information about setting up custom formats.

Standard Windows Clipboard formats are always added to the list for you. The name and result strings for standard formats are defined in OLEVIEW.RC. To localize the strings, edit this file. (Standard formats do not have an identifier string. Instead they have a registration number, such as CF_TEXT.)

Public Constructor and Destructor

```
TOcNameList();
~TOcNameList();
```

Public Member Functions

```
TOcFormatName* operator [](char far*);
TOcFormatName*& operator [](unsigned index);
int Add(TOcFormatName* name);
void Clear(int del = 1);
virtual uint Count() const;
int Detach(const TOcFormatName* name, int del = 0);
unsigned Find(const TOcFormatName* name) const;
int IsEmpty() const;
```

See Also
LocalizingSymbolNames
TOcApp
TOcApp::AddUserFormatName
TOcFormatName

TOcNameList Public Constructors and Destructor

See Also   TOcNameList

Constructor
```
TOcNameList();
```

Destructor
```
~TOcNameList();
```

Description

Constructs a name list containing no items. To insert names in the list, call Add.

Destructor

Destroys the list and the objects in the list.

See Also
TOcNameList::Add

TOcNameList::operator []

See Also   TOcNameList

Syntax

Form 1

```
TOcFormatName*& operator[](unsigned index);
```

Form 2

```
TOcFormatName* operator[](char far* id);
```

Description

Form 1: Returns the item at position *index* in the list of format name objects. The first object is at index 0. If *index* points past the end of the list, the function throws a precondition exception.

Form 2: Returns the format name object whose format ID string matches *id*. The return value is 0 if no match is found.

See Also
TOcFormatName

TOcNameList::Add

See Also TOcNameList

Syntax

```
int Add(TOcFormatName* name);
```

Description

Inserts the object *name* into the list. Returns 1 for success and 0 for failure.

See Also

TOcNameList::Clear

TOcNameList::Detach

TOcNameList::Clear

See Also TOcNameList

Syntax

```
void Clear(int del = 1);
```

Description

Empties the list. If *del* is 1, *Clear* also deletes each object in the list.

See Also

TOcNameList::Add

TOcNameList::Detach

TOcNameList::Count

See Also TOcNameList

Syntax

```
virtual uint Count() const;
```

Description

Returns the number of items in the list.

See Also
TOcNameList::IsEmpty

TOcNameList::Detach

See Also TOcNameList

Syntax

```
int Detach(const TOcFormatName* name, int del = 0);
```

Description

Removes the single object *name* from the list. If *del* is 1, *Detach* also deletes the object *name*.

See Also

TOcNameList::Clear

TOcNameList::Add

TOcNameList::Find

TOcNameList

Syntax

```
unsigned Find(const TOcFormatName* name) const;
```

Description

Searches the list and returns the position of *name*. If the *name* object is not in the list, *Find* returns UINT_MAX.

TOcNameList::IsEmpty

See Also TOcNameList

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the list currently contains no items and 0 if it contains at least one item.

See Also
TOcNameList::Count

TOcPart Class

See Also Linking and Embedding Classes

Header File
ocf/ocpart.h

Base Class
TUnknown

Description

A *TOcPart* object represents a linked or embedded object in a document. It represents the linked or embedded object as the container sees it. From the server's side, the same linked or embedded OLE object has two parts: data (TOcDocument) and a graphical representation of the data (TOcRemView). *TOcPart* manages a site in the container's document where a server places an OLE object.

TOcPart is a COM object and implements the *IUnknown* interface.

Public Constructors

```
TOcPart(TOcDocument& document, TOcInitInfo far& initInfo, TRect pos, int id
  = 0);
TOcPart(TOcDocument& document, const char far* name);
```

Public Member Functions

```
bool operator ==(const TOcPart& other);
bool Activate(bool activate);
bool Close();
void Delete();
int Detach();
bool DoVerb(uint whichVerb);
bool Draw(HDC dc, const TRect& pos, const TRect& clip, TOcAspect aspect =
  asDefault);
bool EnumVerbs(const TOcVerb&);
LPCOLESTR GetName();
int GetNameLen();
TPoint GetPos() const;
TRect GetRect() const;
LPCOLESTR GetServerName(TOcPartName partName);
TSize GetSize() const;
bool IsActive() const;
bool IsLink() const;
bool IsSelected() const;
bool IsVisible(const TRect& logicalRect) const;
bool IsVisible() const;
bool Load();
bool Open(bool open);
void Rename();
bool Save(bool SameAsLoaded = true);
bool Save(IStorage* storage, bool sameAsLoad, bool remember);
void Select(bool select);
void SetActive();
bool SetHost(IBContainer far* container);
void SetPos(const TPoint& pos);
void SetSize(const TSize& size);
void SetVisible(bool visible);
bool Show(bool show);
```

```
void UpdateRect();
```

Protected Destructor

```
~TOcPart();
```

See Also
Activating and Deactivating Objects
Connector Objects
Insert Object command
Painting the Document
TOcDocument
TOcPartCollection
TOcRemView
TUnknown

TOcPart Public Constructors

See Also TOcPart

Form 1

```
TOcPart(TOcDocument& document, TOcInitInfo far& initInfo, TRect pos, int id
  = 0);
```

Form 2

```
TOcPart(TOcDocument& document, const char far* name);
```

Description

Both constructors expect to receive the container's own TOcDocument object. This represents the compound document where the new object will be placed.

Form 1: *document* is the container's *TOcDocument* object representing the compound document that will hold the newly created part. *initInfo* contains information about the object being inserted. It is usually obtained during a paste, drop, or insertion operation. The coordinates in *pos* designate the area where the new object will be drawn. *id* is any arbitrary unique integer used to distinguish this object from others in the same document. If *id* is 0, *TOcPart* generates a new ID automatically.

Form 2: *document* is the same as for Form 1. The *name* string is the name of a linked or embedded part. The second form is used when loading a part from a compound document. The name of the part is also the name of the storage where the part was written.

See Also

TOcDocument

TOcInitInfo

TOcPart::Delete

TRect (OWL.HLP)

TOcPart::operator ==

TOcPart

Syntax

```
bool operator==(const TOcPart& other);
```

Description

Returns **true** if *other* is the same *TOcPart* as **this**. This operator is defined for the use of the TOcPartCollection class.

TOcPart::Activate

See Also TOcPart

Syntax

```
bool Activate(bool activate);
```

Description

If *activate* is **true**, this function activates the part by asking the server to execute its primary (or default) verb for the object. If the default verb is Edit, for example, *Activate* initiates an in-place editing session. If *activate* is **false**, then this function de-activates an in-place editing session.

Activate returns **true** if the server is able to execute the command.

See Also

TOcPart::IsActive

TOcPart::Open

TOcPart::Close

TOcPart

Syntax

```
bool Close();
```

Description

Disconnects the embedded object from its server. Returns **true** if the server closes successfully.

TOcPart::Delete

See Also TOcPart

Syntax

```
void Delete();
```

Description

Delete is used when the user selects an embedded object and presses the Delete key (or does a cut operation.) It first calls Close to disconnect the container from the embedded object. Then it releases the reference to the embedded part.

See Also
TOcPart::Close

TOcPart::Detach

TOcPart

Syntax

```
int Detach();
```

Description

Separates a part from its document. Call *Detach* before cutting a part to the Clipboard, for example.

TOcPart::DoVerb

See Also TOcPart

Syntax

```
bool DoVerb(uint whichVerb);
```

Description

Tells the server to execute one of its commands on the part. A verb is usually an action such as Edit or Play. One server can support several verbs, and *whichVerb* identifies a particular verb by its ordinal value. (The first verb, the primary or default verb, is zero). *DoVerb* returns **true** if the server is able to complete the requested action. Executing a verb can cause the part to become activated.

See Also
TOcPart::EnumVerbs

TOcPart::Draw

See Also TOcPart

Syntax

```
bool Draw(HDC dc, const TRect& pos, const TRect& clip, TOcAspect aspect =
  asDefault);
```

Description

Draws the part on the screen. If the part has not yet been loaded, *Draw* loads it first.

dc is a Windows device context where the part is to be drawn. The coordinates in *pos* tell where in the window to place the part. The *clip* rectangle designates an area outside of which the server cannot draw. *clip* and *pos* can be the same. If *clip* describes an empty rectangle, then the server can draw anywhere. *aspect* controls how the data were presented--as an icon, for example.

See Also
TOcAspect enum
TRect (OWL.HLP)

TOcPart::EnumVerbs

TOcPart

Syntax

```
bool EnumVerbs(const TOcVerb& verb);
```

Description

Call *EnumVerbs* to find out what verbs the server supports for a particular part. Each call to *EnumVerbs* places another verb in the *verb* parameter. When all the server's verbs have been enumerated, *EnumVerbs* returns **false**.

TOleWindow calls *EnumVerbs* in order to place verbs for the active object on the container's Edit menu.

TOcPart::GetName

See Also TOcPart

Syntax

```
LPCOLESTR GetName();
```

Description

Returns the string that identifies the part. Every part in a document has a different name. ObjectComponents creates the names for you automatically by incrementing an internal ID number for each new part.

See Also

TOcPart::GetNameLen

TOcPart::Rename

TOcPart::GetNameLen

See Also		TOcPart

Syntax

```
int GetNameLen();
```

Description

Returns the number of characters in the name string that identifies the part. The count does not include the terminating null character.

See Also

TOcPart::GetName

TOcPart::Rename

TOcPart::GetPos

See Also TOcPart

Syntax

```
TPoint GetPos() const;
```

Description

Returns the part's position within its container document. The position specifies the part's upper left corner in client area coordinates. The coordinates take into account any scaling set for the TOcView object that holds the part.

See Also
TOcPart::GetRect
TOcPart::GetSize
TOcPart::SetPos
TPoint (OWL.HLP)

TOcPart::GetRect

See Also TOcPart

Syntax

```
TRect GetRect() const;
```

Description

Returns the rectangle that bounds the image of the part in the container's client area. The position of the rectangle is given in client area coordinates.

See Also
TOcPart::GetPos
TOcPart::GetSize
TOcPart::UpdateRect
TRect (OWL.HLP)

TOcPart::GetServerName

See Also TOcPart

Syntax

```
LPCOLESTR GetServerName(TOcPartName partName);
```

Description

Asks OLE for the name of the object or of the object's server, depending on the value of partName. A container might want to display this information in its title bar.

In the current implementation of ObjectComponents, this function is not used. The TOcView object automatically updates the container window title.

See Also
TOcPartName enum

TOcPart::GetSize

See Also TOcPart

Syntax

```
TSize GetSize() const;
```

Description

Returns the size of the part's image in the container document. The fields of the return value give the width and height of the part in client area coordinates. If there is scaling, the coordinates take that into account.

See Also
TOcPart::GetPos
TOcPart::GetRect
TOcPart::SetSize
TSize (OWL.HLP)

TOcPart::IsActive

See Also TOcPart

Syntax

```
bool IsActive() const;
```

Description

Returns **true** if the part is currently active and **false** if it is not.

See Also
TOcPart::SetActive

TOcPart::IsLink

TOcPart

Syntax

```
bool IsLink() const;
```

Description

Returns **true** if the part represents a linked OLE object and **false** if it represents an embedded OLE object. A container might use this method to distinguish visually between linked and embedded objects. For an example, look at the source code for TOleWindow::PaintParts.

TOcPart::IsSelected

See Also TOcPart

Syntax

```
bool IsSelected() const;
```

Description

Returns **true** if the part is currently selected and **false** if it is not. This function is frequently called in loops that process all the selected objects in a document. For example, when TOleView paints the parts in a document, it calls *IsSelected* for each one to determine where to paint selection boxes.

Selection state information is maintained entirely in *TOcPart* and does not affect the OLE object itself.

See Also
TOcPart::Select

TOcPart::IsVisible

See Also TOcPart

Syntax

Form 1
```
bool IsVisible() const;
```

Form 2
```
bool IsVisible(const TRect& logicalRect) const;
```

Description

Form 1: Returns **true** if the part is currently visible and **false** if it is hidden.

Form 2: Returns **true** if the part is currently visible within the given *logicalRect* area of the container's window. Returns **false** if the part is not visible, perhaps because the user has scrolled to another part of the document.

See Also
TOcPart::SetVisible

TOcPart::Load

See Also TOcPart

Syntax

```
bool Load();
```

Description

Initializes a *TOcPart* object with information read from a storage.

See Also
TOcPart::Save

TOcPart::Open

See Also TOcPart

Syntax

```
bool Open(bool open);
```

Description

If *open* is **true**, the *Open* command invokes the server to initiate an out-of-place editing session. More specifically, it asks the server to execute its Open verb. If *open* is **false**, the command tells the server to hide its open editing window but does not end the session.

Open returns **true** for success. If the server does not support editing, *Open* returns **false**.

Note: TOcPart::Close is not the opposite of *TOcPart::Open*. To terminate editing, pass **false** to *Open*.

See Also
TOcPart::Activate

TOcPart::Rename

See Also TOcPart

Syntax

```
void Rename();
```

Description

Causes the part to update the internal name that ObjectComponents generates to distinguish the parts in a document. Call *Rename* whenever you rename the document's file. OLE uses the object's name when creating links, so the object name must accurately reflect the file name in order for links to work.

See Also

TOcPart::GetName

TOcPart::GetNameLen

TOcPart::Save

See Also TOcPart

Syntax

Form 1

```
bool Save(bool sameAsLoaded = true);
```

Form 2

```
bool Save(IStorage* storage, bool sameAsLoaded, bool remember);
```

Description

Form 1: Causes the part to write itself into the document's file stream. If *sameAsLoaded* is **true**, then the part saves itself in the same storage where it was last written. Setting *sameAsLoaded* to **false** causes the part to create a new storage for itself under the document's new root storage. Usually *sameAsLoaded* should be **true** in response to a File|Save command and **false** in response to File|Save As.

(A storage is a compartment within a compound file. ObjectComponents manages the storages for you. Usually you do not have to give explicit instructions about where to store parts.)

Form 2: The second form accepts a pointer to an *IStorage* interface, allowing you to control where the object is written. *sameAsLoaded* is the same as in Form 1. *remember* tells the part whether or not to remember the object in *storage*. When saving a part to its usual file, you typically want it to remember its own storage. When copying a part, on the other hand, you typically want the part to keep its original storage object, not the one where you are saving the copy. When saving a copy to a file for the Clipboard, for example, *remember* should be **false**.

See Also
TOcPart::Load

TOcPart::Select

See Also TOcPart

Syntax

```
void Select(bool select);
```

Description

Tells the part whether or not it is currently selected. Make *select* **true** to select the part and **false** to deselect it. The user selects objects in order to perform operations on them. For example, the user selects an object before copying it to the Clipboard. When TOleView paints its parts, it queries each one and draws a selection box around any that the user has selected.

See Also
TOcPart::IsSelected

TOcPart::SetActive

See Also TOcPart

Syntax

```
void SetActive();
```

Description

Synchronizes an internal flag with the object's actual state, active or inactive. Usually you should not have to call this function. To make a part active, call TOcPart::Activate instead.

See Also

TOcPart::Activate

TOcPart::IsActive

TOcPart::SetHost

TOcPart

Syntax

```
bool SetHost(IBContainer far* container);
```

Description

Moves the part from one container to another. *container* can be an object of type TOcView (or one derived from *TOcView*). It designates the view that receives the part. *SetHost* is not usually called from within the application.

IBContainer is a custom interface defined within the BOCOLE support library. *TOcView* implements this interface.

TOcPart::SetPos

See Also TOcPart

Syntax

```
void SetPos(const TPoint& pos);
```

Description

Sets the part's position within its container document. The position specifies the part's upper left corner in pixels measured from the upper left corner of the container's client window. If there is scaling, the coordinates take that into account.

See Also
TOcPart::GetPos
TOcPart::SetSize
TOcPart::UpdateRect
TPoint (OWL.HLP)

TOcPart::SetSize

See Also TOcPart

Syntax

```
void SetSize(const TSize& size);
```

Description

Sets the size of the part's image in the container document. *size* sets the width and height of the part in client area coordinates. The coordinates take into account any scaling set for the TOcView object that holds the part.

See Also

TOcPart::GetPos

TOcPart::UpdateRect

TSize (OWL.HLP)

TOcPart::SetVisible

See Also TOcPart

Syntax

```
void SetVisible(bool visible);
```

Description

Shows or hides the part, according to the value of *visible*.

See Also

TOcPart::IsVisible

TOcPart::Show

See Also TOcPart

Syntax

```
bool Show(bool show);
```

Description

Makes the part visible. *Show* is used to ask the Link Source to show itself in the container window. If *show* is **false**, the part hides itself. The return value is **true** for success.

See Also
TOcPart::IsVisible

TOcPart::UpdateRect

See Also TOcPart

Syntax

```
void UpdateRect();
```

Description

Sets the part to a new rectangle when its size or position changes. It is called by SetPos and GetRect.

See Also

TOcPart::GetRect

TOcPart::SetPos

TOcPart::SetSize

TOcPart Protected Destructor

TOcPart

Syntax
```
~TOcPart();
```

Destructor
Destroys the *TOcPart* object.

TOcPartCollection Class

See Also Linking and Embedding Classes

Header File
ocf/ocpart.h

Description

Manages a set of TOcPart objects. Every TOcDocument creates a part collection object to maintain the set of OLE objects linked or embedded in the document. The part collection object adds parts, deletes parts, finds them, counts them, and generally helps the document keep track of what it has.

Because *TOcDocument* contains a part collection object, usually you do not have to create or manipulate the collection directly yourself.

Public Constructor and Destructor

```
~TOcPartCollection();
TOcPartCollection();
```

Public Member Functions

```
int Add(TOcPart* const& part);
void Clear();
virtual unsigned Count() const;
int Detach(TOcPart* const& part, int del = 0);
unsigned Find(TOcPart* const& part) const;
int IsEmpty() const;
TOcPart* Locate(TPoint& point);
bool SelectAll(bool select = false);
```

See Also
Painting the Document
TOcPart
TOcPartCollectionIter

TOcPartCollection Public Constructor and Destructor

TOcPartCollection

Constructor
```
TOcPartCollection();
```

Destructor
```
~TOcPartCollection();
```

Description
Creates an empty collection. Call Add to insert parts in the collection.

Destructor
Releases all the servers that supply the linked or embedded objects.

TOcPartCollection::Add

See Also TOcPartCollection

Syntax

```
int Add(TOcPart* const& part);
```

Description

Adds a new part to the collection. Returns 1 for success and 0 for failure.

See Also
TOcPart

TOcPartCollection::Clear

TOcPartCollection

Syntax

```
void Clear();
```

Description

Disconnects all the parts in the collection from their servers, removes them from the collection, and releases them. Tells OLE that this collection has no further need for the servers.

TOcPartCollection::Count

TOcPartCollection

Syntax

```
virtual unsigned Count() const;
```

Description

Returns the number of parts currently in the collection.

TOcPartCollection::Detach

See Also TOcPartCollection

Syntax

```
int Detach(TOcPart* const& part, int del = 0);
```

Description

Removes *part* from the collection. If *del* is nonzero, then *Detach* also releases TOcPart object. If the part's internal reference count reaches zero as a result, the part deletes itself. Returns 1 for success and 0 for failure.

See Also
TOcPart

TOcPartCollection::Find

See Also TOcPartCollection

Syntax

```
unsigned Find(TOcPart* const& part) const;
```

Description

Searches for *part* and returns its position in the collection. If *part* is not in the collection, *Find* returns UINT_MAX.

See Also
TOcPart

TOcPartCollection::IsEmpty

TOcPartCollection

Syntax

```
int IsEmpty() const;
```

Description

Returns **true** if the collection currently contains no objects and **false** if it does contain at least one object.

TOcPartCollection::Locate

See Also TOcPartCollection

Syntax

```
TOcPart* Locate(TPoint& point);
```

Description

Returns the part object visible at a particular point on the screen. The numbers in point are interpreted as logical coordinates. If no part in the collection occupies the given point, *Locate* returns 0.

See Also
TPoint (OWL.HLP)

TOcPartCollection::SelectAll

See Also TOcPartCollection

Syntax

```
bool SelectAll(bool select = false);
```

Description

Sets the selection state of all the parts in the collection. If *select* is **true**, *SelectAll* selects them all. If *select* is **false**, it deselects all the parts. The user can perform actions (such as dragging, deleting, and copying) that affect all the selected objects.

The container conventionally marks selected objects by drawing a rectangle with grapples (handles for moving the rectangle) around each of them. The TOleWindow class does this automatically in ObjectWindows programs.

See Also
TOleWindow (OWL.HLP)

TOcPartCollectionIter Class

See Also Linking and Embedding Classes

Header File
ocf/ocpart.h

Description
A part collection iterator enumerates the objects embedded in a compound document.

A compound document can contain many linked and embedded objects. Within the container, each object is represented by an object of type TOcPart. To manage all the parts it contains, TOcDocument creates a collection object of type TOcPartCollection. The collection object takes care of adding and deleting members of the collection. In order to walk through the current list of its parts, *TOcDocument* also creates a part collection iterator. An iterator basically points to an element in the collection. You can increment the iterator to walk through the list of objects. The iterator signals when it reaches the end (the ++ operator returns 0).

Together the collection and its iterator give the document much flexibility in managing its objects.

Public Constructor
```
TOcPartCollectionIter(const TOcPartCollection& coll);
```

Public Member Functions
```
TOcPart* operator ++(int);
TOcPart* operator ++();
TOcPart* Current() const;
operator int() const;
void Restart();
void Restart(unsigned start, unsigned stop);
```

See Also
Painting the Document
TOcPart
TOcPartCollection

TOcPartCollectionIter Public Constructor

See Also TOcPartCollectionIter

Syntax

```
TOcPartCollectionIter(const TOcPartCollection& coll);
```

Description

Constructs an iterator to enumerate the objects contained in the collection *coll*.

See Also
TOcPartCollection

TOcPartCollectionIter::operator ++

TOcPartCollectionIter

Syntax

Form 1

```
TOcPart* operator++(int);
```

Form 2

```
TOcPart* operator++();
```

Description

Form 1: Returns the current part and then advances the iterator to point to the next part (post-increment).

Form 2: Advances the iterator to point to the next part in the list and then returns that part (pre-increment).

TOcPartCollectionIter::Current

TOcPartCollectionIter

Syntax

```
TOcPart* Current() const;
```

Description

Returns the part that the iterator currently points to.

TOcPartCollectionIter::operator int

TOcPartCollectionIter

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value in order to test whether the iterator has finished enumerating the collection. Returns zero if the iterator has reached the end of the list and a nonzero value if it has not.

TOcPartCollectionIter::Restart

TOcPartCollectionIter

Syntax

Form 1
```
void Restart();
```

Form 2
```
void Restart(unsigned start, unsigned stop);
```

Description

Form 1: Resets the iterator to begin again with the first part in the document.

Form 2: Resets the iterator to enumerate a partial range of objects in the document, beginning with the object at position *start* in the list and ending with the object at position *stop*.

TOcPartName enum

See Also Linking and Embedding Enums

Header File
ocf/ocobject.h

Syntax
```
enum TOcPartName
```

Description
When a container asks the server for the name of a part, it might want any of several possible answers. These values indicate which name the container wants to see.

Constant	Meaning
pnLong	The string the server registered as the *description* for this type of object.
pnShort	The string the server registered as the *progid* for this type of object.
pnApp	The string the server registered as the *description* for the server application as a whole.

See Also
description Registration Key
progid Registration Key
TOcPart::GetServerName

TOcRegistrar Class

See Also Linking and Embedding Classes

Header File
ocf/ocapp.h

Base Class
TRegistrar

Description

TOcRegistrar manages all the registration tasks for an application. It processes OLE-related switches on the command line and records any necessary information about the application in the system registration database. If the application is already registered in the database, the registrar confirms that the registered *path*, *progid*, and *clsid* are still accurate. If not, it reregisters the application.

Every ObjectComponents application needs to create a registrar object. If your application supports linking and embedding, then create a *TOcRegistrar* object. If your application supports automation but not linking and embedding, then you should create a TRegistrar object instead. *TOcRegistrar* extends *TRegistrar* by connecting the application to the BOCOLE support library interfaces that support linking and embedding.

An application's main procedure usually performs these actions with its registrar:

- Construct the registrar, passing it a pointer to the application's factory callback.
- Call IsOptionSet to check for options that might affect how the application chooses to start (for example, remaining invisible if invoked for embedding.
- Call Run to enter the program's message loop.

TOcRegistrar inherits both *IsOptionSet* and *Run* from its base class, *TRegistrar*.

Protected Constructor and Destructor

```
TOcRegistrar(TRegList& regInfo, TComponentFactory callback, string&
  cmdLine, HINSTANCE hInst = _hInstance);
~TOcRegistrar();
```

Public Member Functions

```
HRESULT BOleComponentCreate(IUnknown far* far* retIface, IUnknown far*
  outer, BCID idClass);
void CreateOcApp(uint32 options, TOcApp*& ret);
TAppDescriptor& GetAppDescriptor();
```

Protected Member Functions

```
bool CanUnload();
void far* GetFactory(const GUID& clsid, const GUID far& iid);
void LoadBOle();
```

See Also
clsid Registration Key
Creating a Registrar Object
path Registration Key
progid Registration Key
TRegistrar

TOcRegistrar Public Constructor and Destructor

See Also TOcRegistrar

Syntax

Constructor

```
TOcRegistrar(TRegList& regInfo, TComponentFactory callback, string&
  cmdLine, HINSTANCE hInst = _hInstance);
```

Destructor

```
~TOcRegistrar();
```

Description

regInfo is the application registration structure (conventionally named *appReg*).

callback is the factory callback function that ObjectComponents invokes when it is time for the application to create an object. An ObjectWindows program can use the *TOleFactory* class to implement this callback. For a description of *TOleFactory*, see Factory Template Classes and TOleFactoryBase.

cmdLine holds the command line string that invoked the application.

hInst is the application's instance.

Destructor

Destroys objects the registrar uses internally.

See Also
Factory Template Classes (OWL.HLP)
TComponentFactory typedef
TOleFactoryBase (OWL.HLP)

TOcRegistrar::BOleComponentCreate

See Also TOcRegistrar

Syntax

```
HRESULT BOleComponentCreate(IUnknown far* far* retIface, IUnknown far*
  outer, BCID idClass);
```

Description

Calls the BOCOLE support library to create one of the helper objects that ObjectComponents uses internally. Usually you do not need to call *BOleComponentCreate* yourself.

retIface receives an interface to the requested component.

outer is the *IUnknown* interface of the outer object that you want the new component to become a part of.

idClass identifies the particular component you want to create. The possible values are defined as *cidBolexxxx* constants in ocf/boledefs.h.

The return value is an OLE result, either HR_OK for success or HR_FAIL for failure.

See Also
HR_xxxx Return Constants

TOcRegistrar::CreateOcApp

See Also TOcRegistrar

Syntax

```
void CreateOcApp(uint32 options, TOcApp*& ret);
```

Description

Creates the connector object that attaches an application to OLE. *options* is a set of bit flags indicating the application's running mode. The possible option flags are defined in TOcAppMode. *ret* is where *CreateOcApp* places a pointer to the newly created TOcApp connector object.

CreateOcApp is called during TOcModule::OcInit. You shouldn't have to call it directly yourself.

The purpose of *CreateOcApp* is to shield you from the details of the TOcApp connector object. *TOcApp* is closely tied to the implementation of ObjectComponents, and the details of initializing an OLE session are subject to change.

See Also

TDocTemplate (OWL.HLP)

TOcApp

TOcAppMode enum

TOcModule::OcInit

TOcRegistrar::GetAppDescriptor

TOcRegistrar

Syntax

```
TAppDescriptor& GetAppDescriptor();
```

Description

Returns the application descriptor. ObjectComponents uses an application descriptor internally to hold information about a module. (A DLL gets an application descriptor of its own.) *TAppDescriptor* is undocumented because it is used only internally and is subject to change. The registrar classes, *TOcRegistrar* and TRegistrar, are the supported interfaces to the application descriptor. The registrar constructs the descriptor and most of its member functions call descriptor functions to perform the work.

Usually you will not need to call this method yourself.

TOcRegistrar::CanUnload

TOcRegistrar

Syntax

```
bool CanUnload();
```

Description

Returns **true** if the application is not currently serving any OLE clients and **false** otherwise.

TOcRegistrar::GetFactory

TOcRegistrar

Syntax
```
void far* GetFactory(const GUID& clsid, const GUID far& iid);
```

Description

Returns a pointer to the factory interface for creating the type of object indicated by *clsid*. *iid* names the particular interface you want to receive. If the registrar is unable to find an *iid* interface for *clsid* objects, it returns zero.

ObjectComponents calls a DLL's *GetFactory* member every time a new client loads the DLL. Usually you do not need to call *GetFactory* yourself.

TOcRegistrar::LoadBOle

TOcRegistrar

Syntax

```
void LoadBOle();
```

Description

Loads and initializes the ObjectComponents support library (BOCOLE.DLL). *LoadBOle* throws a TXObjComp exception if it cannot find OCOLE.DLL, or if the installed version is not compatible with the application's version of the library.

TOcRemView Class

See Also Linking and Embedding Classes

Header File
ocf/ocremvie.h

Base Class
TOcView

Description
A linking and embedding server creates a remote view object in order to draw its OLE object in the container's window. *TOcRemView* only draws the object. To load and save the data in the object, the server also needs to create a TOcDocument object. The document and the remote view together represent an OLE object as the server sees it.

The container creates a TOcPart object for every OLE object it receives. The container's part object communicates with the server's document and view objects through OLE. The part tells the server's view when and where to draw the object. It tells the server's document when and where to load or save the object.

Do not confuse the two kinds of views, TOcView with *TOcRemView*. A container creates a single view (*TOcView*) for its compound document. This view can contain parts received from other applications. Each part draws itself by invoking a remote view from its server. Containers create *TOcView* objects and servers create *TOcRemView* objects. (A *TOcRemView* object can become a container also, however, if the user embeds objects within objects.)

In spite of the similar names, *TOcDocument*, *TOcView*, and *TOcRemView* are not part of the ObjectWindows Doc/View model. The nature of OLE makes it beneficial to separate data from its graphical representation, and the terms *document* and *view* express that separation even outside of ObjectWindows.

TOcRemView is a COM object and implements the *IUnknown* interface.

Public Constructor
```
TOcRemView(TOcDocument& doc, TRegList* regList = 0, IUnknown* outer = 0);
```

Public Member Functions
```
virtual bool Copy();
virtual void EvClose();
virtual LPCSTR GetContainerTitle();
void GetInitialRect();
void Invalidate(TOcInvalidate);
bool IsOpenEditing() const;
bool Load(IStorage* storageI);
virtual void Rename();
bool Save(IStorage* storageI);
```

See Also
Connector Objects
Creating Helper Objects for a Document
TOcDocument
TOcView

TOcRemView Public Constructor

See Also TOcRemView

Syntax

```
TOcRemView(TOcDocument& doc, TRegList* regList = 0, IUnknown* outer = 0);
```

Description

A remote view is always associated with a TOcDocument object. The document loads and saves data in an OLE object and the remote view draws the data in the container's window. In both forms of the constructor, *doc* is the document to associate with the view. That means the document must always be created first.

Also, in both forms *regList* is a document registration table. A server that creates different kinds of objects needs several document registration tables, one for each type. The *regList* parameter determines the type of object that the view represents. *outer* points to the *IUnknown* interface of a master object under which the new object is asked to aggregate itself.

Registration tables are built with the BEGIN_REGISTRATION and END_REGISTRATION macros.

The destructor for *TOcRemView* is private. ObjectComponents releases the object when it is no longer needed.

See Also
Registration Macros (OWL.HLP)
TAutoObject
TOcDocument

TOcRemView::Copy

TOcRemView

Syntax
```
virtual bool Copy();
```

Description
Copies the object to the Clipboard. Returns **true** for success.

TOcRemView::EvClose

TOcRemView

Syntax

```
virtual void EvClose();
```

Description

The application's remote view window calls this function when it closes. *EvClose* disconnects the view from any parts displayed in it.

TOcRemView::GetContainerTitle

TOcRemView

Syntax

```
virtual LPCOLESTR GetContainerTitle();
```

Description

Asks the container for its name. The server usually includes this string in its own title bar during out-of-place editing (when the user edits a linked or embedded object in the server's own window, not in the container's).

TOcRemView::GetInitialRect

See Also TOcRemView

Syntax

```
void GetInitialRect();
```

Description

Requests the initial size and position of the area where the server can draw its object. The function initializes Extent, a protected data member that *TOcRemView* inherits from TOcView.

See Also
TOcView::Extent

TOcRemView::Invalidate

See Also TOcRemView

Syntax

```
void Invalidate(TOcInvalidate invalid);
```

Description

Notifies the container's active view that the server has changed either the contents or the appearance of the object. The *invalid* parameter indicates what needs changing. It can be *invData*, *invView*, or both combined with the OR operator (|). If the container is an ObjectComponents application, its active view generates an OC_VIEWPARTINVALID message.

See Also

OC_VIEWxxxx messages

TOcInvalidate enum

TOcRemView::IsOpenEditing

TOcRemView

Syntax

```
bool IsOpenEditing() const;
```

Description

Returns **true** if the view is currently engaged in an open editing session. Open editing occurs when the user chooses an object's Open verb. Open editing takes place in the server's own frame window, unlike in-place editing, which takes place in the container's window. Remote view objects are used in both kinds of editing.

TOcRemView::Load

See Also TOcRemView

Syntax
```
bool Load(IStorage* storageI);
```

Description

Reads from *storageI* information specific to the remote view. This information is part of the data the server stores in the container's file when asked to save an object. *Load* returns **true** for success.

IStorage is a pointer to an OLE interface. *storageI* can be a pointer to a TOcStorage object, the ObjectComponents implementation of that interface.

See Also
TOcRemView::Save

TOcRemView::Rename

TOcRemView

Syntax

```
virtual void Rename();
```

Description

Updates the name string ObjectComponents generates to distinguish the parts in a compound document. *TOcRemView* calls *Rename* during construction to find out what the container wants to call the object. It is usually not necessary for you to call *Rename* directly.

TOcRemView::Save

See Also TOcRemView

Syntax

```
bool Save(IStorage* storageI);
```

Description

Writes to *storageI* information specific to the remote view. This information becomes part of the object data stored in the container's compound document file. Returns **true** for success.

IStorage is a pointer to an OLE interface. *storageI* can be a pointer to a TOcStorage object, the ObjectComponents implementation of that interface.

See Also
TOcRemView::Load

TOcSaveLoad struct

See Also		Linking and Embedding Structs

Header File
ocf/ocview.h

Description
Holds information that a view uses when loading and saving its OLE object parts. The OC_VIEWLOADPART and OC_VIEWSAVEPART messages carry a pointer to this structure in their *lParam*s.

TOleView processes the load and save messages for you. If you are programming with the ObjectWindows Doc/View model, then you do not need to use the *TOcSaveLoad* structure directly. For examples that show how to process the load and save messages, look at the source code for the TOleView::EvOcViewLoadPart and TOleView::EvOcViewSavePart methods in TOleView.

Public Data Members
```
bool Release;
IStorage far* StorageI;
```

See Also
OC_VIEWxxxx messages
TOleView::EvOcViewLoadPart (OWL.HLP)
TOleView::EvOcViewSavePart (OWL.HLP)

TOcSaveLoad::Release

TOcSaveLoad

Syntax

```
bool Release;
```

Description

Is **true** if the view should keep the storage object for future file operations and **false** if it should forget the storage object after using it once.

TOcSaveLoad::StorageI

TOcSaveLoad

Syntax

```
IStorage far* StorageI;
```

Description

Points to the storage object assigned to hold the part. ObjectComponents implements the standard OLE *IStorage* interface in TOcStorage, so *TOcStorage* can be used to construct an *IStorage*.

TOcScaleFactor Class

See Also Linking and Embedding Classes

Header File
ocf/ocview.h

Description
The *TOcScaleFactor* class carries information from a container to a server about how the container wants to scale its document. For example, if the container has a Zoom command and the user chooses to magnify the document to 120%, the server should match the scaling factor when it draws objects embedded in the container.

ObjectComponents passes a reference to an *TOcScaleFactor* object in the *lParam* of OC_VIEWGETSCALE and OC_VIEWSETSCALE messages. When a container receives OC_VIEWGETSCALE, it fills in the object with scaling information. When a server receives the OC_VIEWSETSCALE information, it reads the scaling values and can use them in its paint procedure.

TOcScaleFactor stores scaling information in its two TSize members, *SiteSize* and *PartSize*. The names refer to the area where the container wants to draw an object (the site) and the object itself (the part.) The values in the members need not be the actual size of the site or the part, however. What matters is the ratio of the two sizes. If the *SiteSize* values are twice as large as the *PartSize* values, then the server is being asked to draw the object at twice its default size.

If you are programming with ObjectWindows, then the TOleWindow class takes care of scaling for you. For examples showing how to handle scaling without the benefit of ObjectWindows, look at the source code for the following *TOleWindow* methods: EvOcViewGetScale, EvOcViewSetScale, and SetupDC.

Public Data Members
```
TSize PartSize;
TSize SiteSize;
```

Public Constructors
```
TOcScaleFactor(const BOleScaleFactor far& scaleFactor);
TOcScaleFactor
(const RECT& siteRect, const TSize& partSize);
TOcScaleFactor();
```

Public Member Functions
```
TOcScaleFactor& operator =(const TOcScaleFactor& scaleFactor);
TOcScaleFactor& operator =(const BOleScaleFactor far& scaleFactor);
uint16 GetScale();
void GetScaleFactor(BOleScaleFactor far& scaleFactor) const;
bool IsZoomed();
void SetScale(uint16 percent);
```

See Also

OC_VIEWxxxx Messages

TOleWindow::EvOcViewGetScale (OWL.HLP)

TOleWindow::EvOcViewSetScale (OWL.HLP)

TOleWindow::SetupDC (OWL.HLP)

TSize (OWL.HLP)

TOcScaleFactor::PartSize

See Also TOcScaleFactor

Syntax

```
TSize PartSize;
```

Description

Holds two values describing the default horizontal and vertical extent of a server's object. The values in *PartSize* do not need to be actual measurements. What matters is the ratio of the values here to the values in SiteSize. That ratio determines how an image should be scaled.

See Also

TOcScaleFactor::SiteSize

TSize (OWL.HLP)

TOcScaleFactor::SiteSize

See Also TOcScaleFactor

Syntax

```
TSize SiteSize;
```

Description

Holds two values describing the horizontal and vertical extent of the area a container has allotted for displaying a linked or embedded object. The values in *SiteSize* do not need to be actual measurements. What matters is the ratio of the values here to the values in PartSize. That ratio determines how an image should be scaled.

See Also
TOcScaleFactor::PartSize

TSize (OWL.HLP)

TOcScaleFactor Public Constructors

See Also TOcScaleFactor

Syntax

Constructors

Form 1
```
TOcScaleFactor();
```

Form 2
```
TOcScaleFactor(const RECT& siteRect, const TSize& partSize);
```

Form 3
```
TOcScaleFactor(const BOleScaleFactor far& scaleFactor);
```

Destructor
```
~TOcScaleFactor();
```

Description

Usually you do not have to construct a *TOcScaleFactor* object directly. ObjectComponents creates it for you and passes it in the OC_VIEWGETSCALE or OC_VIEWSETSCALE message.

Form 1: Initializes the site and part extents to 1 so the scaling factor is 100%.

Form 2: Bases the initial scaling factor on the values in the given rectangle structure and size object. Calculates the extents of the rectangle *siteRect* and sets them in SiteSize. Copies *partSize* to PartSize.

Form 3: Bases the initial scaling factor on the values in *scaleFactor*. *BOleScaleFactor* is a structure that the BOCOLE support library uses internally to carry scaling information. You should not have to use the structure directly.

See Also

OC_VIEWxxxx Messages

TOcScaleFactor::PartSize

TOcScaleFactor::SiteSize

TSize (OWL.HLP)

TOcScaleFactor::operator =

See Also TOcScaleFactor

Syntax

Form 1
```
TOcScaleFactor& operator =(const BOleScaleFactor far& scaleFactor);
```

Form 2
```
TOcScaleFactor& operator =(const TOcScaleFactor& scaleFactor);
```

Description

Both forms of the assignment operator copy the values from one scaling object into another.

Form 1: Copies the values in a *BOleScaleFactor* structure. The BOCOLE support library uses this structure internally to carry scaling information.

Form 2: Copies one *TOcScaleFactor* into another.

See Also
OC_VIEWxxxx Messages

TOcScaleFactor::GetScale

See Also TOcScaleFactor

Syntax

```
uint16 GetScale();
```

Description

Retrieves a percentage value expressing the ratio of the part's size to the site's size. For example, if the part size is 20 x 20 and the site size is 40 x 40, then *GetScale* returns 200.

See Also
TOcScaleFactor::SetScale

TOcScaleFactor::GetScaleFactor

TOcScaleFactor

Syntax

```
void GetScaleFactor(BOleScaleFactor far& scaleFactor) const;
```

Description

Fills in *scaleFactor* with values from the *TOcScaleFactor* object. *BOleScaleFactor* is a structure that the BOCOLE library uses to hold the same scaling information. Usually you do not have to call this function directly.

TOcScaleFactor::IsZoomed

TOcScaleFactor

Syntax

```
bool IsZoomed();
```

Description

Returns **true** if the sizes stored for the part and the site do not match.

TOcScaleFactor::SetScale

See Also TOcScaleFactor

Syntax

```
void SetScale(uint16 percent);
```

Description

Sets the ratio of the part's size to the site's size. More specifically, *SetScale* sets the size of the part to 100 and the size of the site to *percent*.

See Also
TOcScaleFactor::GetScale

TOcScrollDir enum

See Also Linking and Embedding Enums

Header File
ocf/ocobject.h

Syntax
```
enum TOcScrollDir
```

Description
The OC_VIEWSCROLL event tells the container when the user performs a drag movement that should scroll the window. The *lParam* of the WM_OCEVENT message carries one of these values to indicate which direction the window has been asked to scroll.

Constant	Meaning
sdScrollUp	Scroll toward the top of the document.
sdScrollDown	Scroll toward the bottom of the document.
sdScrollLeft	Scroll toward the left edge of the document.
sdScrollRight	Scroll toward the right edge of the document.

See Also
Object Components Messages (OWL.HLP)
OC_VIEWxxxx messages
TOleWindow::EvOcViewScroll (OWL.HLP)
WM_OCEVENT message

TOcStorage Class

See Also Compound File I/O Classes

Header File

ocf/ocstorag.h

Description

The *TOcStorage* class encapsulates OLE's *IStorage* interface. It manages storages in compound files.

A compound file contains storages and streams. Storages are analogous to file directories and streams to files within a directory. Streams hold data. Storages hold streams and other storages.

Storages have names, just as directories do. The name of the root storage in a compound file is also the name of the disk file itself, so the rules for naming the root storage are the same as the rules for naming any file. Names of substorages under the root, however, can be up to 32 characters, including the terminating null. For more information about names, see the topic Storage Naming Conventions in OLE.HLP.

When you create a storage, you can choose to open it in transacted mode (the default) or in direct mode. In direct mode, write operations have immediate effect, just as they do in normal file I/O. Transacted mode stores all changes in temporary buffers until you call the Commit method. Commit makes the changes permanent. This makes it possible to undo recent changes by calling Revert, which simply discards the current change buffers.

Commands that return HRESULT values correspond directly to the *IStorage* methods with the same names. The *TOcStorage* versions sometimes perform error checking and throw exceptions, but because in other respects they correspond closely to the *IStorage* versions, you can consult the *IStorage* documentation in the OLE.HLP file for more information.

Public Constructors and Destructors

```
TOcStorage(IStorage* storage);
TOcStorage(ILockBytes far* lkbyt, bool create, uint32 mode =
  STGM_READWRITE|STGM_TRANSACTED);
TOcStorage(TOcStorage& parent, const char far* name, bool create, uint32
  mode = STGM_READWRITE);
TOcStorage(const char far* fileName, bool create, uint32 mode =
  STGM_READWRITE|STGM_TRANSACTED);
~TOcStorage();
```

Public Member Functions

```
HRESULT Commit(uint32 grfCommitFlags);
HRESULT CopyTo(uint32 ciidExclude, IID const far* rgiidExclude, SNB
  snbExclude, TOcStorage& dest);
HRESULT DestroyElement(const char far* name);
HRESULT EnumElements(uint32 reserved1, void far* reserved2, uint32
  reserved3, IEnumSTATSTG far*far* ppenm);
IStorage* GetIStorage();
static HRESULT IsStorageFile(const char far* pwcsName);
static HRESULT IsStorageILockBytes(ILockBytes far* plkbyt);
HRESULT MoveElementTo(char const far* name, TOcStorage& dest, char const
  far* newName, uint32 grfFlags);
HRESULT RenameElement(const char far* oldName, const char far* newName);
HRESULT Revert();
HRESULT SetClass(const IID far& clsid);
HRESULT SetElementTimes(const char far* name, FILETIME const far* pctime,
  FILETIME const far* patime, FILETIME const far* pmtime);
HRESULT SetStateBits(uint32 grfStateBits, uint32 grfMask);
```

```
static HRESULT SetTimes(char const far* lpszName, FILETIME const far*
  pctime, FILETIME const far* patime, FILETIME const far* pmtime);
HRESULT Stat(STATSTG far *pstatstg, uint32 grfStatFlag);
HRESULT SwitchToFile(const char far* newPath);
```

Protected Member Functions

```
ulong AddRef();
HRESULT CreateStorage(const char far* name, uint32 mode, uint32 rsrvd1,
  uint32 rsrvd2, IStorage far*far* storage);
HRESULT CreateStream(const char far* name, uint32 mode, uint32 rsrvd1,
  uint32 rsrvd2, IStream far* far* stream);
HRESULT OpenStorage(const char far* name, IStorage far* stgPriority, uint32
  mode, SNB snbExclude, uint32 rsrvd, IStorage far*far* storage);
HRESULT OpenStream(const char far* name, void far *rsrvd1, uint32 grfMode,
  uint32 rsrvd2, IStream far *far *stream);
ulong Release();
```

See Also
TOcStream

TOcStorage Public Constructors and Destructor

See Also TOcStorage

Form1
```
TOcStorage(const char far* fileName, bool create, uint32 mode =
  STGM_READWRITE|STGM_TRANSACTED);
```

Form2
```
TOcStorage(TOcStorage& parent, const char far* name, bool create, uint32
  mode = STGM_READWRITE);
```

Form3
```
TOcStorage(ILockBytes far* lkbyt, bool create, uint32 mode =
  STGM_READWRITE|STGM_TRANSACTED);
```

Form4
```
TOcStorage(IStorage* storage);
```

Destructor
```
~TOcStorage();
```

Description

The *create* and *mode* parameters are common to most forms of the constructor.

create determines what happens if a storage with the given name does not exist. If *create* is **true**, the constructor creates the storage. If *create* is **false**, the constructor throws the *TXObjComp::xRootStorageOpenError* or *TXObjComp::xStorageOpenError* exception.

mode determines the access modes for the new storage. The possible values are defined as STGM_*xxxx* flags. The constructors ignore any share mode flags (such as STGM_SHARE_EXCLUSIVE) and determine the sharing mode automatically based on other mode flags.

For root storages, if you specify STGM_READWRITE, the share mode is STGM_SHARE_DENY_WRITE. If you specify STGM_READONLY, the share mode is STGM_SHARE_DENY_NONE.

Substorages and streams are always opened in STGM_SHARE_EXCLUSIVE mode.

Form1

Constructs a new root storage object.

fileName is the name of the storage (analogous to a directory name). Because it names a root storage, *fileName* must conform to the operating system's restrictions on file names. If *fileName* is 0, the constructor creates a temporary storage that is deleted from the disk when the storage object is released.

Form2

Constructs a substorage within another storage (like a subdirectory within a directory).

parent is the name of the parent storage that contains the new substorage.

name is the name of the new substorage. The name can be up to 32 characters, including the terminating null. It cannot contain any of the following characters: ! . : / \.

Form3

Constructs a new storage based on the given *ILockBytes* interface. *ILockBytes* is OLE's interface for low-level media access. The new storage performs all its I/O through *lkbyt*.

If it fails, the constructor throws the *TXObjComp::xStorageILockError* exception.

Form4

Constructs a new *TOcStorage* object using the given *IStorage* interface pointer. Adds a reference count to the pointer. The new object implements all its standard OLE methods through the pointer.

Destructor

Releases the storage object.

See Also

STGM xxxx constants

TXObjComp

TOcStorage::AddRef

See Also TOcStorage

Syntax

```
ulong AddRef();
```

Description

Increments the storage object's reference count. ObjectWindows uses this method internally. Unless you are passing the storage object between tasks, you probably won't need to use *AddRef* yourself.

The return value is the object's new reference count. The OLE specifications indicate that the return value from any *AddRef* call should be used for diagnostic purposes only, not in shipping code.

See Also
TOcStorage::Release

TOcStorage::Commit

See Also TOcStorage

Syntax

```
HRESULT Commit(uint32 commitFlags);
```

Description

Makes permanent any changes made to the storage object since the last *Commit* command, or, if *Commit* was not called, since the storage was opened.

commitFlags controls how the changes are committed. It contains flags from the STGC **enum**.

If the storage was opened in direct mode, *Commit* has no effect. The mode is set in the constructor. Also, changes committed to a substorage do not actually become permanent until the parent commits, as well. If the parent reverts, its substorages also revert.

See Also

STGC enum

TOcStorage Public Constructors and Destructor

TOcStorage::Revert

TOcStorage::CopyTo

See Also TOcStorage

Syntax

```
HRESULT CopyTo(uint32 ciidExclude, IID const far* rgiidExclude, SNB
  snbExclude, TOcStorage& dest);
```

Description

Copies one storage to another. The first three parameters allow you to exclude some elements from the copy operation, identifying them either by interface IDs or by the names of stream and substorage elements. All subelements nested within the source, down to any level, are copied as well. Copied elements are added to the existing contents of the destination, overwriting any with the same names.

ciidExclude tells how many items are in the *rgiidExclude* array. If *rgiidExclude* is 0, then this parameter is ignored.

rgiidExclude points to an array of interfaces identifiers that the caller takes responsibility for copying. *CopyTo* leaves them alone. If *rgiidExclude* is 0, all elements are copied. If *rgiidExclude* is not 0 but *ciidExclude* is 0, *CopyTo* copies only the state of the storage, not its contents.

snbExclude points to a string name block (SNB) naming elements in the storage that should not be copied. An SNB value points to an array of string pointers. The last element in the array is a null pointer. The strings themselves follow contiguously in memory after the null pointer. This parameter is ignored if the *rgiidExclude* array contains *IID_IStorage* (the GUID identifier for the *IStorage* interface).

dest is the *TOcStorage* object that receives the copied information.

See Also

TOcStorage::MoveElementTo

TOcStorage::SwitchToFile

TOcStream::Write

TOcStorage::CreateStorage

See Also TOcStorage

Syntax

```
HRESULT CreateStorage(const char far* name, uint32 mode, uint32 rsrvd1,
  uint32 rsrvd2, IStorage far*far* storage);
```

Description

Creates a new substorage (like creating a subdirectory within a directory).

name points to the name of the new storage.

mode combines STGM xxxx flags to set access modes for the new storage.

rsrvd1 and *rsrvd2* are reserved for future use and must be set to 0.

storage is where the command returns the *IStorage* interface to the newly created object. If creation fails, *storage* is set to 0.

See Also

STGM xxxx constants

TOcStorage::CreateStream

TOcStorage::DestroyElement

TOcStorage::OpenStorage

TOcStorage::CreateStream

See Also TOcStorage

Syntax

```
HRESULT CreateStream(const char far* name, uint32 mode, uint32 rsrvd1,
  uint32 rsrvd2, IStream far* far* stream);
```

Description

Creates a new stream within the storage (like a file within a directory).

name points to the name of the new stream.

mode combines STGM xxxx flags to set access modes for the new stream.

rsrvd1 and *rsrvd2* are reserved for future use and must be set to 0.

stream is where the command returns the *IStream* interface to the newly created object. If creation fails, *stream* is set to 0.

See Also

STGM xxxx constants

TOcStorage::CreateStorage

TOcStorage::DestroyElement

TOcStorage::OpenStream

TOcStorage::DestroyElement

See Also TOcStorage

Syntax

```
HRESULT DestroyElement(const char far* name);
```

Description

Deletes an element (a storage or a stream) from within the storage. *name* is the identifier assigned to the element when it was created. If the storage uses transacted mode, this command can be undone using *Revert*.

See Also

TOcStorage::CreateStream

TOcStorage::CreateStorage

TOcStorage::EnumElements

See Also TOcStorage

Syntax

```
HRESULT EnumElements(uint32 reserved1, void far* reserved2, uint32
  reserved3, IEnumSTATSTG far*far* ppenm);
```

Description

Returns an enumerator object whose methods can list all the elements (storages and streams) within the storage. The first three parameters are reserved and must be 0. *ppenm* is where the command returns the *IEnumSTATSTG* interface to the newly created enumeration object.

For information on the *IEnumSTATSTG* interface, see OLE.HLP.

See Also
TOcStorage::RenameElement

TOcStorage::GetIStorage

TOcStorage

Syntax

```
IStorage* GetIStorage();
```

Description

Returns a pointer to the object's *IStorage* interface without adding a reference count. This pointer can be returned in response to *QueryInterface*.

TOcStorage::IsStorageFile

See Also TOcStorage

Syntax

```
static HRESULT IsStorageFile(const char far* pwcsName);
```

Description

Determines whether the file named *pwcsName* contains an *IStorage* object. If the file contains a storage object, the function returns S_OK. If it does not, the function returns S_FALSE.

See Also
TOcStorage::IsStorageILockBytes

TOcStorage::IsStorageILockBytes

See Also TOcStorage

Syntax

```
static HRESULT IsStorageILockBytes(ILockBytes far* plkbyt);
```

Description

Determines whether the *plkbyt* byte array object contains an *IStorage* object. If the lockbytes object contains a storage, the function returns S_OK. If it does not, it returns S_FALSE.

See Also
TOcStorage::IsStorageFile

TOcStorage::MoveElementTo

See Also TOcStorage

Syntax

```
HRESULT MoveElementTo(char const far* name, TOcStorage& dest, char const
  far* newName, uint32 grfFlags);
```

Description

Moves an element from one storage to another (like moving files from one directory to another).

name identifies the element to be moved.

dest is the *TOcStorage* object where the element will be placed.

newName is the element's new identifier.

grfFlags determines whether the original element is deleted after being copied to *dest*. The possible values are STGMOVE_MOVE and STGMOVE_COPY.

See Also

TOcStorage::CopyTo

TOcStorage::DestroyElement

TOcStorage::SwitchToFile

TOcStorage::OpenStorage

See Also TOcStorage

Syntax

```
HRESULT OpenStorage(const char far* name, IStorage far* stgPriority, uint32
  mode, SNB snbExclude, uint32 rsrvd, IStorage far*far* storage);
```

Description

Opens an existing substorage object within the storage.

name identifies the storage to open.

stgPriority is usually 0. If you have already opened the *name* storage element, passing the original *IStorage* pointer causes *OpenStorage* to close and reopen the storage using the new *mode* access flags.

mode combines STGM_*xxxx* flags to indicate the access modes for the new storage object.

snbExclude points to an array of elements within the substorage that will be emptied (but not destroyed) when the storage is opened. This parameter is usually used together with *stgPriority*.

rsrvd is reserved and must be 0.

storage is where the command returns a pointer to the newly opened storage object's *IStorage* interface. If the command fails, *storage* is 0.

For more information about the priority and exclusion parameters, see the description of *StgOpenStorage* in OLE.HLP.

See Also

STGM xxxx constants

TOcStorage::OpenStream

TOcStorage::OpenStream

See Also TOcStorage

Syntax

```
HRESULT OpenStream(const char far* name, void far *rsrvd1, uint32 grfMode,
  uint32 rsrvd2, IStream far *far *stream);
```

Description

Opens an existing stream object within the storage.

name identifies the stream to open.

rsrvd1 and *rsrvd2* are reserved and must be 0.

grfMode combines STGM_*xxxx* flags to indicate the access modes for the new stream object.

stream is where the command returns a pointer to the newly opened storage object's *IStream* interface. If the command fails, *stream* is 0.

See Also

STGM xxxx constants

TOcStorage::OpenStorage

TOcStorage::Release

See Also TOcStorage

Syntax

```
ulong Release()
```

Description

Decrements the object's reference count. ObjectWindows uses this method internally. Unless you are passing the storage object between tasks, you probably won't need to use *Release* yourself.

The return value is the object's new reference count. When it reaches 0, the object destroys itself. The OLE specifications indicate that the return value from any *Release* call should be used for diagnostic purposes only, not in shipping code.

See Also
TOcStorage::AddRef

TOcStorage::RenameElement

See Also TOcStorage

Syntax

```
HRESULT RenameElement(const char far* oldName, const char far* newName);
```

Description

Changes the identifier assigned to a stream or substorage within the storage.

oldName identifies the element whose name you want to change.

newName is the new identifier to give the element.

See Also
TOcStorage::EnumElements

TOcStorage::Revert

See Also TOcStorage

Syntax

```
HRESULT Revert();
```

Description

Revokes any changes made to a storage, or to any elements it contains, since the last *Commit* command. If *Commit* was not called, the storage reverts to the state it was in when opened.

See Also
TOcStorage::Commit

TOcStorage::SetClass

See Also TOcStorage

Syntax

```
HRESULT SetClass(const IID far& clsid);
```

Description

Assigns *clsid* as the globally unique identifier (GUID) for the storage. The intitial GUID for any new storage is CLSID_NULL. Call *Stat* to determine the storage's current class ID. The class ID is a persistent attribute.

See Also

STATSTG::clsid

TOcStorage::Stat

TOcStorage::SetElementTimes

See Also TOcStorage

Syntax
```
HRESULT SetElementTimes(const char far* name, FILETIME const far* pctime,
  FILETIME const far* patime, FILETIME const far* pmtime);
```

Description

Compound files can record the times of creation, last modification, and last access for each element within the file. This command sets these values explicitly for any element. Time values are expressed as FILETIME structures. To determine the values already set for these times, call *Stat*.

name identifies the element whose time attributes you want to modify.

pctime points to a structure containing the new creation time.

patime points to a structure containing the new time to show for last access to the file.

pmtime points to a structure containing the new time to show for last modification to the file.

Leave any of the FILETIME parameters 0 to avoid changing the existing value.

Not all implementations of compound files support all three times for all elements. The attributes of a root storage, for example, depend on the underlying file system. Microsoft's OLE implementation does not support any time stamps for internal streams. If you attempt to assign values for unsupported attributes, the values are ignored.

See Also

FILETIME struct

TOcStorage::SetTimes

TOcStorage::Stat

TOcStorage::SetStateBits

See Also TOcStorage

Syntax

```
HRESULT SetStateBits(uint32 grfStateBits, uint32 grfMask);
```

Description

Stores information about the current state of the storage as bits in a 32-bit value. When first created, a storage's state is 0. Currently no state bits are defined, but all 32 are reserved for system use and applications should not use them privately.

See Also
STATSTG::grfStateBits

TOcStorage::SetTimes

See Also TOcStorage

Syntax

```
static HRESULT SetTimes(char const far* name, FILETIME const far* pctime,
  FILETIME const far* patime, FILETIME const far* pmtime);
```

Description

Sets the times of creation, last modification, and last access for a disk file. Time values are expressed as FILETIME structures.

name identifies the file whose time attributes you want to modify.

pctime points to a structure containing the new creation time.

patime points to a structure containing the new time to show for last access to the file.

pmtime points to a structure containing the new time to show for last modification to the file.

To avoid changing an existing value, set the corresponding FILETIME parameter to 0.

The underlying operating system determines whether all three of the time attributes are supported. If you attempt to assign values for unsupported attributes, the values are simply ignored.

See Also
FILETIME struct
TOcStorage::SetElementTimes

TOcStorage::Stat

See Also TOcStorage

Syntax

```
HRESULT Stat(STATSTG far *pstatstg, uint32 statFlag);
```

Description

Returns information about the storage.   The information includes, for example, the storage's name, its size, and the times it was created, modified, and last used.

pstatstg points to a STATSTG structure that receives information from *Stat*.

statFlag controls the level of detail in the information retrieved. It can be either STATFLAG_DEFAULT or STATFLAG_NONAME. Choosing not to receive the name saves an allocation operation.

See Also
TOcStorage::RenameElement
TOcStorage::SetElementTimes
TOcStorage::SetTimes

TOcStorage::SwitchToFile

See Also TOcStorage

Syntax

```
HRESULT SwitchToFile(const char far* newPath);
```

Description

Copies an entire compound file to a new disk file and associates the storage with the new disk file. *newPath* names the destination file. *SwitchToFile* fails if the storage object is not a root storage (supporting the *IRootStorage* interface).

See Also

TOcStorage::CopyTo

TOcStorage::MoveElementTo

TOcStream Class

See Also   Compound File I/O Classes

Header File
ocf/ocstorag.h

Description
The *TOcStream* class encapsulates the methods of OLE's *IStream* interface. It reads and writes data in compound files.

In compound file I/O, storages are analogous to directories and streams to files. A storage can contain both streams and other storages. Each stream represents a set of data and each substorage is a compartment for more streams.

A *TOcStream* object always works together with the *TOcStorage* object that contains it.

The streams within a storage have names, just as files do. A stream's name can contain up to 32 characters, including the terminating null.   Stream names cannot contain the following characters: ! . / \ :.   For more information about names, see the topic Storage Naming Conventions in OLE.HLP.

The OLE definition of *IStream* includes methods such as *Commit* and *Revert*, to allow for operation in transacted mode, where changes are stored in a temporary buffer and become permanent only when committed. The current implementation of OLE supports transacted only for storages, not for streams. At present, *TOcStream::Commit* and *TOcStream::Revert* have no effect. The *LockRegion* and *UnlockRegion* similarly support features not yet available in OLE.

Commands that return HRESULT values correspond directly to the *IStream* methods with the same names. The *TOcStream* versions sometimes perform error checking and throw exceptions, but because they correspond closely to the *IStream* versions, you can consult the *IStream* documentation in the OLE.HLP file for supplementary information.

Public Constructors and Destructors
```
TOcStream(TOcStorage& storage, const char far* name, bool create, uint32
  mode = STGM_READWRITE);
TOcStream(TOcStream& stream);
TOcStream(IStream* stream);
~TOcStream();
```

Public Member Functions
```
HRESULT Commit(uint32 commitFlags);
HRESULT CopyTo(TOcStream& stream, uint64 cb, uint64 far* read = 0, uint64
  far* written = 0);
IStream* GetIStream();
HRESULT LockRegion(uint64 offset, uint64 cb, uint32 lockType);
HRESULT Read(void HUGE* pv, ulong cb, ulong far* pcbRead = 0);
HRESULT Revert();
HRESULT Seek(int64 move, uint32 origin, uint64 far* newPosition = 0);
HRESULT SetSize(uint64 newSize);
HRESULT Stat(STATSTG far* statstg, uint32 statFlag);
HRESULT UnlockRegion(uint64 offset, uint64 cb, uint32 lockType);
HRESULT Write(void const HUGE* pv, ulong cb, ulong far* written = 0);
```

See Also
TOcStorage

TOcStream Public Constructors and Destructor

See Also TOcStream

Form1

```
TOcStream(TOcStorage& storage, const char far* name, bool create, uint32
  mode = STGM_READWRITE);
```

Form2

```
TOcStream(TOcStream& stream);
```

Form3

```
TOcStream(IStream* stream);
```

Destructor

```
~TOcStream();
```

Description

Form1

Constructs a new stream object inside the given *storage* object. *name* is the name of the stream (analogous to a file name). *create* determines what happens if a stream called *name* does not exist. If *create* is **true**, the constructor creates the stream. If *create* is **false**, the constructor throws the *TXObjComp::xStreamOpenError* exception. *mode* is a combination of STGM_*xxxx* flags (storage mode flags).

(Internally, the constructor parameters are passed to the *storage.OpenStream* or *storage.CreateStream* method.)

Form2

Creates a new stream object by copying *stream*. (Internally the constructor calls *stream.Clone*.)

Form3

Constructs a *TOcStream* given an existing *IStream* interface pointer. Does not add a reference count to the pointer.

Destructor

Releases the stream object.

See Also

STGM xxxx constants

TOcStorage

TXObjComp

TOcStream::Commit

See Also TOcStream

Syntax

```
HRESULT Commit(uint32 commitFlags);
```

Description

Commits any changes made to the storage object containing the stream.

commitFlags controls how the changes are committed. It contains flags from the STGC **enum**.

Currently this command has no effect because OLE does not support transacted mode for stream objects.

See Also

STGC enum

TOcStream::Revert

TOcStream::CopyTo

See Also TOcStream

Syntax

```
HRESULT CopyTo(TOcStream& stream, uint64 cb, uint64 far* read = 0, uint64
  far* written = 0);
```

Description

Copies data to another stream. The copy operation begins from the current file pointer and writes to the pointer position in *stream*. *cb* is the number of bytes to copy. *read* is the number of bytes read from the source stream, and *written* is the number of bytes written to the destination.

See Also
TOcStream::Write

TOcStream::GetIStream

TOcStream

Syntax
```
IStream* GetIStream();
```

Description
Returns a pointer to the object's *IStream* interface without adding a reference count. This pointer can be returned in response to *QueryInterface*.

TOcStream::LockRegion

See Also		TOcStream

Syntax

```
HRESULT LockRegion(uint64 offset, uint64 cb, uint32 lockType);
```

Description

Locks a range of bytes in the stream, restricting the access other programs are allowed.

Locking is not supported in the current release of OLE, so the *LockRegion* command currently has no effect.

offset is the file position of the first byte to lock. *cb* is the number of bytes to lock. *lockType* determines whether to block all access or only write access. The possible values are LOCK_WRITE, LOCK_EXCLUSIVE, and LOCK_ONLYONCE.

See Also

STATSTG::grfLocksSupported

TOcStream::UnlockRegion

TOcStream::Read

See Also TOcStream

Syntax

```
HRESULT Read(void HUGE* pv, ulong cb, ulong far* pcbRead = 0);
```

Description

Copies bytes from the stream to the *pv* buffer. *cb* is the number of bytes to read. After the command executes, *pcbRead* points to the number of bytes actually read.

Read alters the position of the seek pointer within the stream. *Read* fails if the stream object was not constructed with the STGM_READ mode flag.

See Also

STGM xxxx constants

TOcStream::Write

TOcStream::Revert

See Also TOcStream

Syntax

```
HRESULT Revert();
```

Description

Discards all changes since the the last *Commit* command (or since the stream was opened, if no *Commit* has occurred.)

Currently this command has no effect because OLE does not support transacted mode for stream objects.

See Also
TOcStream::Commit

TOcStream::Seek

See Also   TOcStream

Syntax

```
HRESULT Seek(int64 move, uint32 origin, uint64 far* newPosition = 0);
```

Description

Moves the seek pointer within the stream. The seek pointer points to the location in the file where the next read or write operation will begin. The *Read* and *Write* commands both move the seek pointer, as well.

Seek can offset the pointer from its present position or move it to an absolute position measured from the beginning or the end of the file.

move is a distance measurement within the file. Its interpretation depends on the value of *origin*.

origin tells whether move measures a distance from the pointer's current position or from one end of the file.

After *Seek*, *newPosition* points to the beginning of the stream.

This table shows the possible values of *origin*.

Constant	**Meaning**
STREAM_SEEK_SET	Set the pointer forward *move* bytes from the start of the file.
STREAM_SEEK_CUR	Offset the pointer *move* bytes from its current position. (*move* can be negative.)
STREAM_SEEK_END	Offset the pointer *move* bytes from the end of the file. (*move* can be negative.)

It is legal to move the pointer past the end of the stream (but not to move it before the start.)

To determine the current position of the pointer, call *Seek* passing 0 for *move* and STREAM_SEEK_SET for *origin*. *newPosition* then tells the pointer's position measured from the start of the file.

See Also

TOcStream::Read

TOcStream::Write

TOcStream::SetSize

See Also TOcStream

Syntax

```
HRESULT SetSize(uint64 newSize);
```

Description

Changes the size of the stream. If *newSize* is bigger than the current size, bytes are added to the end. (The content of the new bytes is undefined.) If *newSize* is smaller, the stream is truncated.

SetSize does not affect the position of the seek pointer.

See Also
TOcStream::Seek

TOcStream::Stat

See Also TOcStream

Syntax

```
HRESULT Stat(STATSTG far* statstg, uint32 statFlag);
```

Description

Returns information about the open stream.   The information includes, for example, the stream's name, its size, and the times it was created, modified, and last used.

statstg points to a STATSTG structure that receives information from *Stat*.

statFlag controls the level of detail in the information retrieved. It can be either STATFLAG_DEFAULT or STATFLAG_NONAME. Choosing not to receive the name saves an allocation operation.

See Also
STATSTG struct

TOcStream::UnlockRegion

See Also TOcStream

Syntax

```
HRESULT UnlockRegion(uint64 offset, uint64 cb, uint32 lockType);
```

Description

Unlocks a range of bytes previously locked with LockRegion.

Locking is not supported in the current release of OLE, so *LockRegion* and *UnlockRegion* currently have no effect.

offset is the file position of the first byte to unlock. *cb* is the number of bytes to unlock. *lockType* indicates the kind of restriction to remove.

All three parameters must exactly match those set in a previous call to LockRegion. Every call to *LockRegion* must have a corresponding *UnlockRegion*.

See Also

STATSTG::grfLocksSupported

TOcStream::LockRegion

TOcStream::Write

See Also TOcStream

Syntax

```
HRESULT Write(void const HUGE* pv, ulong cb, ulong far* written = 0);
```

Description

Copies bytes from the *pv* buffer to the stream. *cb* is the number of bytes to write. After the command executes, *written* points to the number of bytes actually written.

Write alters the position of the seek pointer within the stream. *Write* fails if the stream object was not constructed with the STGM_WRITE mode flag.

See Also

TOcStream::CopyTo

TOcStream::Read

TOcStream::Seek

TOcToolBarInfo struct

See Also Linking and Embedding Structs

Header File
ocf/ocview.h

Description
The OC_VIEWSHOWTOOLS message carries a pointer to this structure in its *lParam*. The message asks a server for handles to its tool bars so the container can display them in its own window. This happens during in-place editing when the user opens an object in the container in order to modify it.

The structure has four fields, allowing the server to return handles for up to four tool bars. Each tool bar occupies a different edge of the container's client area.

For examples, look at the source code for TOleWindow::EvOcViewShowTools. The default implementations of these methods allow a single tool bar at the top of the client area. To give the container more tool bars, handle the OC_VIEWSHOWTOOLS message directly yourself.

Public Data Members
```
HWND HBottomTB;
HWND HFrame;
HWND HLeftTB;
HWND HRightTB;
HWND HTopTB;
bool Show;
```

See Also
OC_VIEWxxxx messages
TOleWindow::EvOcViewShowTools (OWL.HLP)

TOcToolBarInfo::HBottomTB

TOcToolBarInfo

Syntax

```
HWND HBottomTB;
```

Description

Holds a handle to the tool bar that the server wants to place at the bottom of the container's client area.

TOcToolBarInfo::HFrame

See Also TOcToolBarInfo

Syntax

```
HWND HFrame;
```

Description

If *Show* is **true** and the server is being asked to display its tool bar, then *HFrame* holds a handle to the frame window where the tool bar is to appear. If *Show* is **false**, then *HFrame* holds a handle to the server's own frame window.

See Also
TOcToolBarInfo::Show

TOcToolBarInfo::HLeftTB

TOcToolBarInfo

Syntax

```
HWND HLeftTB;
```

Description

Holds a handle to the tool bar that the server wants to place at the left edge of the container's client area.

TOcToolBarInfo::HRightTB

TOcToolBarInfo

Syntax
```
HWND HRightTB;
```

Description
Holds a handle to the tool bar that the server wants to place at the right edge of the container's client area.

TOcToolBarInfo::HTopTB

TOcToolBarInfo

Syntax

```
HWND HTopTB;
```

Description

Holds a handle to the tool bar that the server wants to place at the top of the container's client area.

TOcToolBarInfo::Show

TOcToolBarInfo

Syntax

```
bool Show;
```

Description

Is **true** to ask that the server display its tool bar or **false** to request that the server hide the tool bar.

TOcVerb Class

See Also Linking and Embedding Classes

Header File
ocf/ocpart.h

Description
Holds information about a single verb that a server supports for its objects.

A verb is an action the server can perform with one of its objects. A server that creates text objects, for example, might support an Edit verb. A server for sound objects might support Edit, Play, and Rewind.

When the user selects an object in a compound document, the container asks the TOcPart object for a list of the verbs it can execute. The container displays the verbs on its Edit menu. The command for enumerating verbs is TOcPart::EnumVerbs.

Whenever the user selects a part, the container modifies its Edit menu by adding an item for manipulating the object. If the object is part of a Quattro Pro spreadsheet, for example, the container adds the command Notebook Object to its Edit menu. If the user selects this command, then the container shows a pop-up menu with the notebook's verbs, Edit and Open.

For an example of how to implement these items on the edit menu, look at the source code for TOleWindow::CeEditObject in OLEWINDO.CPP.

Public Constructor
```
TOcVerb();
```

Public Data Members
```
bool CanDirty;
LPCOLESTR TypeName;
uint VerbIndex;
LPCOLESTR VerbName;
```

See Also
TOcPart::EnumVerbs

TOcVerb Public Constructor

TOcVerb

Syntax

```
TOcVerb();
```

Description

Creates an empty verb object.

TOcVerb::CanDirty

TOcVerb

Syntax

```
bool CanDirty;
```

Description

Is **true** if executing the verb can modify the object so that it might need to be saved or redrawn afterwards. For example, the *CanDirty* field of an Edit verb is always **true**, and the *CanDirty* field of a Play verb is usually **false**.

TOcVerb::TypeName

See Also TOcVerb

Syntax

```
LPCOLESTR TypeName;
```

Description

Points to the name of the type of object to which this verb belongs. The container usually shows this name in the Object item of its Edit menu. For example, if the user has selected an object inserted from the server in Step 15 of the ObjectWindows tutorial, *TypeName* is "Drawing Pad," and the container's Edit menu should have an item saying "Drawing Pad." Choosing this item leads to a pop-up menu with all the picture's verbs on it.

The *TypeName* string comes from the value the server registered for the menuname key in its document registration table.

See Also

TOcPart::EnumVerbs

menuname Registration Key

TOcVerb::VerbIndex

TOcVerb

Syntax

```
uint VerbIndex;
```

Description

Holds the index number that identifies this verb in the server's list of possible verbs. The first verb is always 0 and is considered the default verb. If the user double-clicks the object, the container should ask the server to execute its default verb.

TOcVerb::VerbName

TOcVerb

Syntax

```
LPCOLESTR VerbName;
```

Description

Points to the name of the verb. This is the string that the container adds to its Edit menu.

TOcView Class

See Also Linking and Embedding Classes

Header File
ocf/ocview.h

Base Class
TUnknown

Description
TOcView manages the presentation of a container's compound document containing linked and embedded objects. Each object in the document is represented by an object of type TOcPart. The document view knows which parts are selected or activated. It scrolls the window and remembers which parts are visible. It transfers parts to and from the document through the Clipboard or through drag-and-drop operations.

Every *TOcView* has a corresponding TOcDocument. The ObjectComponents document object implements the OLE interfaces that manipulate the data in a compound document. *TOcView* implements the interfaces the manipulate the appearance of a compound document.

TOcView is a COM object and implements the *IUnknown* interface.

Public Constructor
```
TOcView(TOcDocument& doc, TRegList* regList = 0, IUnknown* outer=0);
```

Public Member Functions
```
bool ActivatePart(TOcPart* part);
bool BrowseClipboard(TOcInitInfo& initInfo);
bool BrowseLinks();
bool Copy(TOcPart* part);
void EvActivate(bool activate);
virtual void EvClose();
void EvResize();
bool EvSetFocus(bool set);
TOcPart* GetActivePart();
TOcDocument& GetOcDocument();
TPoint GetOrigin() const;
TRect GetWindowRect() const;
void InvalidatePart(const TOcPart* part);
bool Paste(bool linking = false);
bool RegisterClipFormats(TRegList& regList);
virtual void ReleaseObject();
virtual void Rename();
void ScrollWindow(int dx, int dy);
void SetLink(bool pasteLink);
void SetupWindow(HWND hWin, bool embedded = false);
```

Protected Destructor
```
~TOcView();
```

Protected Member Functions
```
uint32 ForwardEvent(int eventId, const void far* param);
uint32 ForwardEvent(int eventId, uint32 param = 0);
void Init(TRegList* regList);
void Shutdown();
```

Protected Data Members
```
TOcPart* ActivePart;
```

```
TSize Extent;
TOcFormatList FormatList;
int Link;
TOcApp& OcApp;
TOcDocument& OcDocument;
TPoint Origin;
HWND Win;
string WinTitle;
```

See Also

Connector Objects

Creating ObjectComponents View and Document Objects

TOcApp

TOcDocument

TOcPart

TUnknown

TOcView Public Constructor

See Also TOcView

Syntax

```
TOcView(TOcDocument& doc, TRegList* regList = 0, IUnknown* outer=0);
```

Description

doc refers to the TOcDocument object that corresponds to the view. *TOcDocument* manages the data in a compound document, and *TOcView* manages the appearance of the document on the screen.

regList is the registration structure for a particular document. Use the BEGIN_REGISTRATION and END_REGISTRATION macros to create an object of type TRegList.

outer is the root interface of an outer object inside which the new view is asked to aggregate itself.

See Also
Registration Macros (OWL.HLP)
TAutoObject
TOcDocument

TOcView Protected Destructor

TOcView

Syntax

```
~TOcView();
```

Description

Destroys the view object.

TOcView::ActivatePart

See Also TOcView

Syntax

```
bool ActivatePart(TOcPart* part);
```

Description

Attempts to activate the given part (by calling TOcPart::Activate). Returns **true** if the designated part becomes active and **false** otherwise. If any other part was already active, it is deactivated first.

See Also
TOcPart::Activate
TOcView::ActivePart
TOcView::GetActivePart

TOcView::BrowseClipboard

See Also TOcView

Syntax

```
bool BrowseClipboard(TOcInitInfo& initInfo);
```

Description

Displays the Paste Special dialog box showing the available formats for the data currently on the Clipboard, allowing the user to choose what format to paste. Returns **true** if the user pastes data and **false** if the user cancels or the dialog box fails.

Create *initInfo* first by passing the view to the TOcInitInfo constructor. *BrowseClipboard* fills *initInfo* with information about the object. Then use *initInfo* to create a new TOcPart.

This function calls TOcApp::BrowseClipboard.

See Also

TOcApp::BrowseClipboard

TOcInitInfo

TOcPart

TOcView::BrowseLinks

See Also TOcView

Syntax

```
bool BrowseLinks();
```

Description

Displays the Links dialog box showing all the linked objects in the compound document and what they are linked to. The user can modify the displayed links, perhaps to reconnect with a file that was moved. Returns **false** if an error prevents the dialog box from being displayed or if the user cancels the dialog box.

See Also
TOcApp::BrowseClipboard

TOcView::Copy

TOcView

Syntax

```
bool Copy(TOcPart* part);
```

Description

Creates a copy of a linked or embedded object and places it on the Clipboard. Returns **true** if the operation succeeds. Call *Copy* in response to Cut or Copy commands from the Edit menu.

TOcView::EvActivate

See Also TOcView

Syntax

```
void EvActivate(bool activate);
```

Description

A container calls this function if any of its windows gains focus while any of its linked or embedded objects is being edited in place. *EvActivate* restores focus to the in-place activated view. If the user clicks in the client window of an MDI frame, for example, the client window needs to shift the focus back to the view, which in turn restores focus to the activated part. A part engaged in in-place editing should always retain the focus.

activate should be **true** if the window is gaining focus and **false** if it is losing it.

See Also
TOcView::EvClose
TOcView::EvResize
TOcView::EvSetFocus

TOcView::EvClose

See Also TOcView

Syntax

```
virtual void EvClose();
```

Description

A container calls this function to tell ObjectComponents that the window associated with the view has closed.

See Also

TOcView::EvActivate

TOcView::EvResize

TOcView::EvSetFocus

TOcView::EvResize

See Also TOcView

Syntax

```
void EvResize();
```

Description

A container calls this function to tell OLE when the window associated with the view changes size. OLE might need this information to let a server modify its tool bar during in-place editing.

See Also
TOcView::EvActivate
TOcView::EvClose
TOcView::EvSetFocus

TOcView::EvSetFocus

See Also TOcView

Syntax

```
bool EvSetFocus(bool set);
```

Description

A container calls this function to tell OLE that the window associated with the view has either received or lost the input focus. Make *set* **true** if the window gained the focus or **false** if it lost the focus.

The function returns **false** if the view is unable to receive the focus. That happens if an object in the view is engaged in in-place editing. Such objects retain the focus until the editing session ends.

See Also

TOcView::EvActivate

TOcView::EvResize

TOcView::EvSetFocus

TOcView::GetActivePart

See Also TOcView

Syntax

```
TOcPart* GetActivePart();
```

Description

Returns the currently active part. If the view does not contain an active part, the return value is 0.

See Also

TOcView::ActivePart

TOcView::ActivatePart

TOcView::GetOcDocument

See Also TOcView

Syntax

```
TOcDocument& GetOcDocument();
```

Description

Returns the ObjectComponents document associated with the view. Views and documents work in pairs. *TOcView* manages the appearance of a compound document and TOcDocument manages the data in it.

See Also

TOcDocument

TOcView::OcDocument

TOcView::GetOrigin

See Also TOcView

Syntax

```
TPoint GetOrigin() const;
```

Description

Returns the physical coordinates currently mapped to the upper left corner of the container window's client area. ObjectWindows programmers can ignore this method because TOleWindow performs scrolling for you.

See Also
TOcView::Origin
TOcView::ScrollWindow
TOleWindow (OWL.HLP)
TPoint (OWL.HLP)

TOcView::GetWindowRect

See Also TOcView

Syntax

```
TRect GetWindowRect() const;
```

Description

Returns the client rectangle for the view window.

See Also
TOcView::GetOrigin
TRect (OWL.HLP)

TOcView::InvalidatePart

TOcView

Syntax

```
void InvalidatePart(const TOcPart* part);
```

Description

Sends an OC_VIEWPARTINVALID message to the container window. If the container window responds with **false** to indicate it has not processed the message, *InvalidatePart* tells the system that the area inside the part's bounding rectangle is invalid and needs repainting.

TOcView::Paste

TOcView

Syntax

```
bool Paste(bool linking = false);
```

Description

Inserts an object from the Clipboard into the compound document. If *linking* is true, *Paste* will try to create a link rather than embedding the new object. Make *linking* **true** when processing the Paste Link command.

TOcView::RegisterClipFormats

See Also TOcView

Syntax

```
bool RegisterClipFormats(TRegList& regList);
```

Description

Tells OLE what Clipboard formats the document understands. The list of formats comes from *regList*, the document's registration structure. Use the BEGIN_REGISTRATION and END_REGISTRATION to create *regList*. Also, the REGFORMAT places Clipboard format entries in the structure. To register custom Clipboard formats, be sure to call TOcApp::AddUserFormatName as well. For more information, see Registration Macros in OWL.HLP.

RegisterClipFormats is called automatically when the view is constructed.

See Also
Registration Macros (OWL.HLP)
TOcApp::AddUserFormatName
TOcView::FormatList

TOcView::ReleaseObject

See Also TOcView

Syntax

```
virtual void ReleaseObject();
```

Description

Call this instead of delete to destroy a *TOcView* object when you are through with it. *ReleaseObject* decrements the view's internal reference count and dissociates the view from its window.

See Also
TOcView::SetupWindow

TOcView::Rename

TOcView

Syntax

```
virtual void Rename();
```

Description

Tells OLE when the name assigned to a compound document has changed. OLE updates its internal records. Also, the associated TOcDocument object passes the new name to any linked or embedded objects it contains.

TOcView::ScrollWindow

See Also TOcView

Syntax

```
void ScrollWindow(int dx, int dy);
```

Description

Brings new areas of a document into view by adjusting the origin of the container window. *dx* and *dy* are horizontal and vertical offsets added to the origin. This function is usually called in response to messages from the window scroll bars or from the arrow keys.

See Also

TOcView::GetOrigin

TOcView::Origin

TOcView::SetLink

See Also TOcView

Syntax

```
void SetLink(bool pasteLink);
```

Description

Sets an internal flag that determines whether Paste operations create linked or embedded objects. More specifically, *SetLink* alters the priority of the document's registered Clipboard formats. You set the original priorities with the first parameter of the REGFORMAT macro. If *pasteLink* is **true**, then *SetLink* moves the Link Source format to the top of the list. If *pasteLink* is **false**, it restores the Link Source format to its original position behind Embed Source.

It is usually not necessary to call *SetLink* directly because the Paste method calls it for you.

See Also
Registration Macros (OWL.HLP)
TOcView::Paste

TOcView::SetupWindow

See Also TOcView

Syntax

```
void SetupWindow(HWND hWin, bool embedded = false);
```

Description

Tells the view what window is associated with it. The view sometimes sends notification messages to its window. Usually this function should be called from the SetupWindow member of the container's window class. TOleWindow performs this task automatically.

embedded should be **true** only in a server invoked by OLE to support a container. (Check for this condition by calling TOcModule::IsOptionSet and looking for the *amEmbedded* flag.) Passing **true** for *embedded* directs ObjectComponents to execute registration steps in a sequence that permits a container/server application to embed its own objects within itself.

See Also
OC_VIEWxxxx messages
TOcAppMode
TOcModule::IsOptionSet
TOleWindow::SetupWindow (OWL.HLP)
TOcView::Win

TOcView::ForwardEvent

See Also TOcView

Syntax

Form 1

```
uint32 ForwardEvent(int eventId, const void far* param);
```

Form 2

```
uint32 ForwardEvent(int eventId, uint32 param = 0);
```

Description

Both forms send a WM_OCEVENT message to the container's window. The *eventId* parameter becomes the message's *wParam* and should be one of the OC_APPxxxx or OC_VIEWxxxx constants. The second parameter becomes the message's *lParam* and may be either a pointer (Form 1) or an integer (Form 2). Which form you use depends on the information a particular event needs to send in its *lParam*.

See Also
OC_APPxxxx messages
OC_VIEWxxxx messages
TOcView::Win
WM_OCEVENT message

TOcView::Init

See Also TOcView

Syntax

```
void Init(TRegList* regList);
```

Description

Initializes a newly created view object. *Init* is called by both of the *TOcView* constructors. Usually you don't need to call it directly yourself. TRegList is the data type that holds all the registry keys and associated values for a single registration table. *regList* must be a document registration table (the structure created by the registration macrosand conventionally named *DocReg*.)

Init makes this view the document's active view, connects with the BOCOLE support library, and registers supported Clipboard formats.

See Also
Registration Macros (OWL.HLP)

TOcView::Shutdown

See Also TOcView

Syntax

```
void Shutdown();
```

Description

Called by the destructor of derived classes to release helper objects that the view holds internally.

See Also
TOcView Public Constructors and Destructor

TOcView::ActivePart

See Also TOcView

Syntax

```
TOcPart* ActivePart;
```

Description

Remembers which part in a document is the currently active part.

See Also

TOcView::ActivatePart

TOcView::GetActivePart

TOcView::Extent

See Also TOcView

Syntax
```
TSize Extent;
```

Description
Holds the current width and height of the container window's client area. Both are measured in device units.

See Also

TOcView::GetWindowRect

TSize (OWL.HLP)

TOcView::FormatList

See Also TOcView

Syntax

```
TOcFormatList FormatList;
```

Description

Holds information about all the Clipboard formats the compound document supports. The list is generated from information the application registers for the types of documents it supports.

See Also

TOcFormatList

TOcView::RegisterClipFormats

TOcView::Link

TOcView

Syntax

```
int Link;
```

Description

Used internally by the Paste method to adjust the priority of link source format.

TOcView::OcApp

TOcView

Syntax

```
TOcApp& OcApp;
```

Description

A view stores the application that owns it in this protected data member.

TOcView::OcDocument

See Also TOcView

Syntax

```
TOcDocument& OcDocument;
```

Description

A view stores the document object that owns the view in this protected data member. The view object manages the appearance of a compound document, and the document object manages the data.

See Also
TOcView::GetOcDocument

TOcView::Origin

See Also TOcView

Syntax

```
TPoint Origin;
```

Description

Holds the coordinates of the point currently mapped to the upper left corner of the container window's client area.

See Also
TOcView::GetOrigin

TOcView::Win

See Also TOcView

Syntax

```
HWND Win;
```

Description

Holds a handle to the window where the view draws itself. The ForwardEvent method sends messages to this window.

See Also

TOcView::ForwardEvent

TOcView::SetupWindow

TOcView::WinTitle

TOcView

Syntax
```
string WinTitle;
```

Description
Holds the original caption string of the container's window. The caption is usually modified as the user moves from part to part within the document. When no part is active, the view restores the window's title to this original string.

TOcViewCollectionIter Class

See Also Linking and Embedding Classes

Header File
oclink.h

Description
A view collection iterator enumerates the views of a compound document. A compound document can have many views. Within the container, each object is represented by an object of type TOcPart. To manage all the views it contains, TOcDocument creates a collection object of type TOcViewCollection. The collection object takes care of adding and deleting members of the collection. In order to walk through the current list of its views, *TOcDocument* also creates a view collection iterator. An iterator basically points to an element in the collection. You can increment the iterator to walk through the list of objects. The iterator signals when it reaches the end (the ++ operator returns 0).

Together the collection and its iterator give the document much flexibility in managing its objects.

Public Constructor
```
TOcViewCollectionIter(const TOcViewCollection& c);
```

Public Member Functions
```
TOcLinkView* operator ++();
TOcLinkView* operator ++(int);
TOcLinkView* Current() const;
operator int() const;
void Restart();
void Restart(unsigned start, unsigned stop);
```

See Also

TOcLinkView

TOcViewCollection

TOcViewCollectionIter Public Constructor

See Also TOcViewCollectionIter

Syntax

```
TOcViewCollectionIter(const TOcViewCollection& c);
```

Description

Constructs an iterator to enumerate the view objects contained in the collection *c*

See Also
TOcViewCollection

TOcViewCollectionIter::operator ++

TOcViewCollectionIter

Syntax

```
TOcLinkView* operator ++;
TOcLinkView* operator ++(int);
```

Description

Form 1: Returns the current view and then advances the iterator to point to the next view (post-increment).

Form 2: Advances the iterator to point to the next view in the list and then returns that view (pre-increment).

TOcViewCollectionIter::Current

TOcViewCollectionIter

Syntax

```
TOcLinkView* Current() const;
```

Description

Returns the view that the iterator currently points to.

TOcViewCollectionIter::operator int

TOcViewCollectionIter

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value in order to test whether the iterator has finished enumerating the collection. Returns zero if the iterator has reached the end of the list and a nonzero value if it has not.

TOcViewCollectionIter::Restart

TOcViewCollectionIter

Syntax

Form 1
```
void Restart();
```

Form 2
```
void Restart(unsigned start, unsigned stop);
```

Description

Form 1: Resets the iterator to begin again with the first view of the document.

Form 2: Resets the iterator to enumerate a partial range of views of the document, beginning with the object at position *start* in the list and ending with the object at position *stop*.

TOcViewCollection Class

See Also Linking and Embedding Classes

Header File
oclink.h

Description

Manages a set of TOcLinkView objects. Every TOcDocument creates a link view collection object to maintain the set of link views associated with the document. The view collection object adds views, deletes views, finds them, counts them, and generally helps the document keep track of what views it has.

Because *TOcDocument* contains a view collection object, usually you do not have to create or manipulate the collection directly yourself.

Public Constructors and Destructors

```
TOcViewCollection();
~TOcViewCollection();
```

Public Member Functions

```
int Add(TOcLinkView* const& View);
void Clear();
virtual unsigned Count() const;
void operator delete(void* ptr) ;
int Detach(TOcLinkView* const& view, int del = 0);
unsigned Find(TOcLinkView* const& view) const;
TOcLinkView* Find(TString const moniker) const;
int IsEmpty() const;
```

See Also

TOcView

TOcLinkView

TOcViewCollectionIter

TOcViewCollection Public Constructor and Destructor

TOcViewCollection

Constructor
```
TOcViewCollection();
```

Destructor
```
~TOcViewCollection();
```

Description
Creates an empty collection. Call Add to insert link views into the collection.

Destructor
Releases all the servers that supply the link views in this collection.

TOcViewCollection::Add

See Also TOcViewCollection

Syntax

```
int Add(TOcLinkView* const& View);
```

Description

Adds a new link view to the collection. Returns 1 for success and 0 for failure.

See Also
TOcView

TOcViewCollection::Clear

TOcViewCollection

Syntax

```
void Clear();
```

Description

Disconnects all the link views in the collection from their servers, removes them from the collection, and releases them. Tells OLE that this collection has no further need for the servers.

TOcViewCollection::Count

TOcViewCollection

Syntax

```
virtual unsigned Count();
```

Description

Returns the number of link views currently in the collection.

TOcViewCollection::Delete

TOcViewCollection

Syntax

```
void operator delete(void* ptr);
```

Description

Deletes a link view object (*ptr*) from the collection.

TOcViewCollection::Detach

See Also TOcViewCollection

Syntax

```
int Detach(TOcLinkView* const& view, int del = 0);
```

Description

Removes this link view from the collection. If *del* is nonzero, then *Detach* also releases the TOcLinkViewobject. If the view's internal reference count reaches zero as a result, the view deletes itself. Returns 1 for success and 0 for failure.

See Also
TOcLinkView

TOcViewCollection::Find

See Also TOcViewCollection

Syntax

Form 1

```
unsigned Find(TOcLinkView* const& view);
```

Form 2

```
TOcLinkView* Find(TString const moniker) const;
```

Description

Form 1

Searches for *view* and returns its position in the collection. If *view* is not in the collection, *Find* returns UINT_MAX.

Form 2

Search for the view with the given moniker and returns a pointer to the view. If the view is not in the collection, *Find* returns 0.

See Also
TOcLinkView

TOcViewCollection::IsEmpty

TOcViewCollection

Syntax

```
int IsEmpty();
```

Description

Returns **true** if the collection currently contains no objects and **false** if it contains at least one object.

TOcViewPaint struct

See Also Linking and Embedding Structs

Header File
ocf/ocview.h

Description
The OC_VIEWPAINT message carries a pointer to this structure in its *lParam*. The message notifies a server that it should update its painting of an object. The structure carries information about the area that needs repainting. Generally a program should respond by calling paint methods on the window or view that receives the message. For examples, look at the source code for TOleWindow::EvOcViewPaint.

Public Data Members
```
TOcAspect Aspect;
TRect* Clip;
HDC DC;
TOcPart* Part;
TRect* Pos;
```

See Also
OC_VIEWxxxx Messages
TOleWindow::EvOcViewPaint (OWL.HLP)

TOcViewPaint::Aspect

See Also TOcViewPaint

Syntax

```
TOcAspect Aspect;
```

Description

Holds an enumerated value that tells how the part is to be drawn. A single object can often be drawn in more than one way. For example, the server might show the object's full contents, a miniature representation of the contents, or an icon that represents the type of object without indicating its specific contents.

See Also
TOcAspect enum

TOcViewPaint::Clip

See Also TOcViewPaint

Syntax

```
TRect* Clip;
```

Description

Designates the area where the part should be allowed to draw. The server can clip the output to this area to avoid drawing outside its allotted space.

See Also
TRect (OWL.HLP)

TOcViewPaint::DC

TOcViewPaint

Syntax
```
HDC DC;
```

Description
Contains a handle to the device context where the repainting should occur.

TOcViewPaint::Part

See Also TOcViewPaint

Syntax

```
TOcPart* Part;
```

Description

Points to the part that needs to be redrawn. This member can be used to ask the part to repaint itself. In the current implementation of ObjectComponents, this member is not used.

See Also
TOcPart

TOcViewPaint::Pos

See Also TOcViewPaint

Syntax

```
TRect* Pos;
```

Description

Specifies the upper left corner of the server object that has become invalid and needs repainting.

See Also
TRect (OWL.HLP)

TOleAllocator Class

See Also General OLE Classes, Macros, and Type Definitions

Header File
ocf/oleutil.h

Description
A linking and embedding .EXE application creates a memory allocator object in order to tell OLE what memory manager the system should use when allocating and deallocating memory on behalf of the server. Unless you have particular memory management needs, it's easiest to let OLE use its default allocator.

When writing a linking and embedding application, you usually do not need to create a memory allocator object directly because your TOcApp object takes care of it for you. The only applications that create memory allocators directly are automation servers that do not support linking and embedding. Because automation servers don't create *TOcApp* objects, they do need to create *TOleAllocator*s.

DLL servers do not need a memory allocator because the system uses whatever allocator the .EXE client designates.

Public Constructors and Destructor
```
~TOleAllocator();
TOleAllocator();
TOleAllocator(IMalloc* mem = 0);
```

Public Member Functions
```
void far* Alloc(unsigned long size);
void Free(void far* pv);
```

Public Data Member
```
IMalloc* Mem;
```

See Also

Creating a TOleAllocatorObject

TOcApp

TOleAllocator Public Constructors and Destructor

See Also TOleAllocator

Constructor

Form 1
```
TOleAllocator(IMalloc* mem = 0);
```

Form 2
```
TOleAllocator();
```

Destructor
```
~TOleAllocator();
```

Description

Form 1: Initializes the OLE system library and, if *mem* is nonzero, registers a custom memory allocator. Unless you have particular memory management needs, it is easiest to let OLE use its default allocator. To implement your own allocator, refer to the OLE documentation on the *IMalloc* interface.

Form 2: Tells OLE to use the custom memory allocator. Does not initialize the OLE system library. In .EXE applications, the registrar object initializes the OLE library. In DLL servers, the .EXE client provides the allocator.

Destructor

Releases the memory allocator (either the default allocator or a custom allocator) and uninitializes the OLE system.

See Also

TOcRegistrar

TRegistrar

TOleAllocator::Alloc

See Also TOleAllocator

Syntax

```
void far* Alloc(unsigned long size);
```

Description

Calls the *Alloc* method on the active memory allocator to request a block of memory. *size* gives the size of the block. Unless you have registered a custom memory allocator, *Alloc* calls OLE's default allocator. If the request fails, *Alloc* returns 0.

See Also
TOleAllocator::Free

TOleAllocator::Free

See Also TOleAllocator

Syntax

```
void Free(void far* block);
```

Description

Calls the *Free* method on the active memory allocator to release a block of memory previously allocated with *Alloc*. *block* points to the base of the area to be released. Unless you have registered a custom memory allocator, *Free* calls OLE's default allocator.

See Also
TOleAllocator::Alloc

TOleAllocator::Mem

TOleAllocator

Syntax

```
IMalloc* Mem;
```

Description

Points to the active memory allocator object. Unless you have registered a custom memory allocator, *Mem* points to OLE's default allocator.

TRegistrar Class

See Also   Automation Classes

Header File
ocf/ocreg.h

Description

TRegistrar manages all the registration tasks for an application. It processes OLE-related switches on the command line and records any necessary information about the application in the system registration database. If the application is already registered in the database, the registrar confirms that the registered *path*, *progid*, and *clsid* are still accurate. If not, it reregisters the application.

Every ObjectComponents application needs to create a registrar object. If your application supports automation but not linking and embedding, then create a *TRegistrar* object. To support linking and embedding--alone or along with automation--then create a TOcRegistrar instead. *TOcRegistrar* extends *TRegistrar* by connecting the application to the BOCOLE support library interfaces that support linking and embedding.

An application's main procedure usually performs these actions with its registrar:

- Construct the registrar, passing it a pointer to the application's factory callback.
- Call *IsOptionSet* to check for options that might affect how the application chooses to start (for example, terminating if the application was invoked simply for registration.)
- Call *Run* to enter the program's message loop.

Public Constructor and Destructor

```
TRegistrar(TRegList& regInfo, TComponentFactory callback, string& cmdLine,
  HINSTANCE hInst);
virtual ~TRegistrar();
```

Public Member Functions

```
virtual bool CanUnload();
TUnknown* CreateAutoApp(TObjectDescriptor app, uint32 options, IUnknown*
  outer=0);
TUnknown* CreateAutoObject(const void* obj, const typeinfo& objInfo, const
  void* app, const typeinfo& appInfo);
TUnknown* CreateAutoObject(TObjectDescriptor obj, TServedObject& app);
virtual void far* GetFactory(const GUID& clsid, const GUID far& iid);
uint32 GetOptions() const;
bool IsOptionSet(uint32 option) const;
void ProcessCmdLine(string& cmdLine);
void ReleaseAutoApp(TObjectDescriptor app);
void RegisterAppClass();
virtual int Run();
void SetOption(uint32 bit, bool state);
virtual void Shutdown(IUnknown* releasedObj, uint32 options);
void UnregisterAppClass();
```

Protected Data Member

```
TAppDescriptor& AppDesc;
```

Protected Constructor

```
TRegistrar(TAppDescriptor& appDesc);
```

See Also

Creating a Registrar Object

TOcRegistrar

TRegistrar Public Constructor and Destructor

See Also TRegistrar

Syntax

Constructor

```
TRegistrar(TRegList& regInfo, TComponentFactory callback, string& cmdLine,
  HINSTANCE hInst);
```

Destructor

```
virtual ~TRegistrar();
```

Description

regInfo is the application registration structure (conventionally named *appReg*). *callback* is the factory callback function that ObjectComponents invokes when it is time for the application to create a document. An ObjectWindows program can use the *TOleFactory* class to create this callback. *cmdLine* points to the command line received when the application was invoked. *hInst* is the application instance. For more a description of *TOleFactory*, see Factory Template Classes and TOleFactoryBase.

The constructor processes OLE-related switches and removes them from the command line. (Call *IsOptionSet* to determine what switches were found.) It also initializes some settings from the application registration table. If the application is a DLL, the constructor initializes the global *DllRegistrar* variable.

Destructor

Deletes objects the registrar maintains internally.

See Also
Factory Template Classes (OWL.HLP)
string class (CLASSLIB.HLP)
TComponentFactory typedef
TOleFactoryBase (OWL.HLP)
TRegistrar::IsOptionSet

TRegistrar Protected Constructor

See Also TRegistrar

Syntax

```
TRegistrar(TAppDescriptor& appDesc);
```

Description

The protected constructor is used only by the derived class TOcRegistrar.*TAppDescriptor* is a class that both registrar objects (*TRegistrar* and *TOcRegistrar*) use internally to hold information about an application.

See Also
TOcRegistrar

TRegistrar::CanUnload

TRegistrar

Syntax

```
virtual bool CanUnload();
```

Description

Returns **true** if the application is not currently serving any OLE clients and **false** otherwise.

TRegistrar::CreateAutoApp

See Also TRegistrar

Syntax

```
TUnknown* CreateAutoApp(TObjectDescriptor app, uint32 options, IUnknown*
  outer = 0);
```

Description

Creates an instance of an automated application. This method is usually called from the application's TComponentFactory callback function.

app is the automation server's primary automated class created from the TAutoObjectDelete<> template.

options contains the application's mode flags. This is usually the same value passed in to the factory callback function. The possible values are enumerated in TOcAppMode.

outer points to the *IUnknown* interface of an outer component under which the application is asked to aggregate.

The return value points to the new OLE application object.

See Also

TAutoObjectDelete<>

TComponentFactory typedef

TRegistrar::CreateAutoObject

TRegistrar::CreateAutoObject

See Also TRegistrar

Syntax

Form 1
```
TUnknown* CreateAutoObject(TObjectDescriptor obj, TServedObject& app);
```

Form 2
```
TUnknown* CreateAutoObject(const void* obj, const typeinfo& objInfo, const
  void* app, const typeinfo& appInfo);
```

Description

CreateAutoObject asks an automated application to instantiate one of its automated objects. It is usually called from the application's TComponentFactory callback function. Which form you call depends on what information you have to identify the kind of object you want to create.

Form 1: *app* is the automated OLE application object.

obj is the automated C++ object.

Form 2: *app* and *obj* are the same as in Form 1.

objInfo identifies the type of object in *obj*. *appInfo* identifies the type of object in *app*. Both values can be obtained using *typeid*.

See Also

TComponentFactory typedef

TRegistrar::CreateAutoApp

typeid (BCW.HLP)

typeinfo class (CLASSLIB.HLP)

TRegistrar::GetFactory

TRegistrar

Syntax

```
virtual void far* GetFactory(const GUID& clsid, const GUID far& iid);
```

Description

Returns a pointer to the factory interface for creating type object indicated by *clsid*. *iid* names the particular interface you want to receive. If the registrar is unable to find an *iid* interface for *clsid* objects, it returns zero.

ObjectComponents calls a DLL's *GetFactory* member every time a new client loads the DLL. Usually you do not need to call *GetFactory* yourself.

TRegistrar::GetOptions

See Also TRegistrar

Syntax

```
uint32 GetOptions() const;
```

Description

Returns a 32-bit integer containing bit flags that reflect the application's running mode. Some of the flags are set in response to command-line switches. Others are set directly by ObjectComponents. For a list of the mode flags, see the TOcAppMode **enum**.

See Also

TOcAppMode enum

TRegistrar::IsOptionSet

TRegistrar::ProcessCmdLine

TRegistrar::SetOption

TRegistrar::IsOptionSet

See Also TRegistrar

Syntax

```
bool IsOptionSet(uint32 option) const;
```

Description

Returns **true** if a particular option was set as a flag on the application's command line, and **false** if the option was not set. The flags are set by the ProcessCmdLine method.

For a list of possible values *option* can assume, see the TOcAppMode **enum**.

TOcApp objects also have an IsOptionSet method. In most cases they return the same results. In a DLL server, however, the registrar remembers the set of options that the server originally started with, while *TOcApp::IsOptionSet* queries the options for the currently active instance of the DLL.

Usually the TOcModule object manages the creation of the *TOcApp* object for you, so to query per-instance options it is easiest to call TOcModule::IsOptionSet.

See Also
TOcApp::IsOptionSet
TOcAppMode enum
TOcModule::IsOptionSet
TRegistrar::GetOptions
TRegistrar::ProcessCmdLine
TRegistrar::SetOption

TRegistrar::ProcessCmdLine

See Also TRegistrar

Syntax

```
void ProcessCmdLine(string& cmdLine);
```

Description

Locates any OLE-related switches on the application's command line (or passed in to a DLL server from ObjectComponents.) The switches tell the program whether it has been launched independently or as a server, whether it should register or unregister itself, whether to create a type library, and signal other running conditions as well. *ProcessCmdLine* records the presence of each flag it finds. You can call IsOptionSet to determine the results.

The command line is always processed for you when the registrar object is constructed. Usually you do not need to call this function directly.

cmdLine contains the string of arguments passed to the program on its command line. *ProcessCmdLine* removes OLE-related switches from the command line. That lets you process *cmdLine* afterwards for any of your own arguments without worrying about OLE arguments.

See Also
Processing the Command Line
string (CLASSLIB.HLP)
TRegistrar::IsOptionSet

TRegistrar::RegisterAppClass

See Also TRegistrar

Syntax

```
void RegisterAppClass();
```

Description

Tells OLE that an automated application is up and ready to create an application instance. Has no effect if called from an application that does not support automation.

For convenience, it is recommended that every ObjectComponents application, even those that do not support automation, call *RegisterAppClass* on starting up and UnregisterAppClass when closing down. This habit is harmless even if sometimes unnecessary and ensures that you will not forget to include registration functions if you later add automation.

See Also
TRegistrar::UnregisterAppClass

TRegistrar::ReleaseAutoApp

See Also TRegistrar

Syntax

```
void ReleaseAutoApp(TObjectDescriptor app);
```

Description

This method is used by an application's factory callback function if the application must detach itself from OLE before it can shut down. Detaching the application is necessary when an automated application has registered its application object for its class, allowing the controller to manipulate it.

See Also
Factory Template Classes (OWL.HLP)

TRegistrar::Run

See Also TRegistrar

Syntax

```
virtual int Run();
```

Description

Call this function to execute your program. If the application was built as an .EXE file, then *Run* lets the application enter its message loop. If the application was built as a DLL, then *Run* returns without entering the message loop. DLL servers must wait for OLE to call their factory before they run. The purpose of the *Run* function is to let you build your applications as either an .EXE or a DLL without having to modify your code.

In .EXE programs, *Run* performs the following steps:

- If the application is automated, call RegisterAppClass.
- Call the factory function to create and run the application. The application enters its message loop. (In an DLL server, creating and running are separate steps.)
- Call the factory function to shut down the application.
- Ensure that the application's TOcApp connector object is properly released.

See Also

TOcApp

TRegistrar::RegisterAppClass

TRegistrar::Shutdown

TRegistrar::SetOption

See Also TRegistrar

Syntax

```
void SetOption(uint32 bit, bool state);
```

Description

Modifies the application's running mode flags. *bit* contains bit flags from the TOcAppMode **enum**. If *state* is **true**, *SetOption* turns the flags on. If *state* is **false**, it turns the flags off. You should never have to call this function because ObjectComponents always maintains the mode flags.

See Also

TOcAppMode enum

TRegistrar::GetOptions

TRegistrar::IsOptionSet

TRegistrar::ProcessCmdLine

TRegistrar::Shutdown

See Also TRegistrar

Syntax

```
virtual void Shutdown(IUnknown* releasedObj, uint32 options);
```

Description

Calls the application's factory function and asks it to make the application stop. Ensures that the application's TOcApp connector object is properly released. In the normal path of execution, the Run command performs the same tasks. Call *Shutdown* to terminate the application directly.

See Also

TOcApp

TRegistrar::Run

TRegistrar::UnregisterAppClass

See Also TRegistrar

Syntax

```
void UnregisterAppClass();
```

Description

Announces that the application is no longer available for OLE interactions.

See Also
TRegistrar::RegisterAppClass

TRegistrar::AppDesc

TRegistrar

Syntax
```
TAppDescriptor& AppDesc;
```

Description

Holds the application descriptor. ObjectComponents uses an application descriptor internally to manage information about a component. (Like EXEs, each DLL gets an application descriptor of its own.) *TAppDescriptor* is undocumented because it is used only internally and is subject to change. The registrar classes, TOcRegistrar and TRegistrar, are the supported interfaces to the application descriptor. The registrar constructs the descriptor, and most of its member functions call descriptor functions to perform the work.

Usually you will not need to manipulate this data member directly.

TUnknown Class

See Also General OLE Classes, Macros, and Type Definitions

Header File
ocf/oleutil.h

Description
Implements the standard OLE *IUnknown* interface. ObjectComponents derives some of its own classes from *TUnknown*, so usually you do not need to use it directly yourself. Advanced users, however, might find *TUnknown* helpful in creating their own custom Component Object Model (COM) objects.

The *TUnknown* class is the basis for the ObjectComponents implementation of object aggregation. With aggregation, you can make distinct components work together as a single OLE object. A single primary object becomes the outer object, and secondary objects behave as though they are parts of the primary object. For this to work, whenever any inner object is asked for its *IUnknown* interface, it must return the *IUnknown* that belongs to the outer object. If the outer object is asked for an interface it does not support, it forwards the request to the chain of attached inner objects. All the interfaces supported by any object in the aggregation are available through the *QueryInterface* method of the outer object.

To add a new object to a chain of aggregated objects, call the *Aggregate* method from any object already in the chain. Each new component receives the *IUnknown* pointer to its outer object and returns its own *IUnknown* pointer to be placed in the chain of secondary objects.

Public Member Functions
```
IUnknown& Aggregate(TUnknown& inner);
IUnknown* GetOuter();
unsigned long GetRefCount();
operator IUnknown&();
operator IUnknown*();
IUnknown* SetOuter(IUnknown* outer);
```

Protected Constructor and Destructor
```
TUnknown();
virtual ~TUnknown();
```

Protected Member Functions
```
virtual HRESULT QueryObject(const GUID far& iid, void far* far* pif);
IUnknown& ThisUnknown();
```

Protected Data Member
```
IUnknown* Outer;
```

See Also
TComponentFactory typedef

TUnknown::Aggregate

See Also TUnknown

Syntax

```
IUnknown& Aggregate(TUnknown& inner);
```

Description

Aggregates a new object under the current object. *inner* points to the *IUnknown* interface of the new object. The current object stores *inner* for use in responding to future *QueryInterface* calls. It also calls AddRef on the inner pointer.

If **this** is already part of an aggregation, *inner* is passed down to the last inner object in the chain.

Aggregate returns a reference to the object's own outer *IUnknown* interface. The newly added object uses the return value as its *Outer* pointer. To aggregate **this** under an object that is not a *TUnknown*, call SetOuter instead.

Aggregate increments the reference count of the *inner TUnknown* object.

See Also

TUnknown::SetOuter

TUnknown::Outer

TUnknown::GetOuter

See Also		TUnknown

Syntax

```
IUnknown* GetOuter();
```

Description

Returns a pointer to the object's outer *IUnknown* interface, the one that belongs to the primary object in a group of aggregated objects. Has no effect on the reference count of the outer object.

See Also

TUnknown::SetOuter

TUnknown::Outer

TUnknown::GetRefCount

TUnknown

Syntax

```
unsigned long GetRefCount();
```

Description

Returns the reference count of the outer object. If **this** is not aggregated, then *GetRefCount* returns the object's own reference count.

The reference count tells how many clients hold pointers to the object.

TUnknown::operator IUnknown*()

See Also TUnknown

Syntax

```
operator IUnknown*();
```

Description

Returns a pointer to the object's outer *IUnknown* interface. Increments the object's reference count first.

See Also
TUnknown::operator IUnknown&

TUnknown::Protected Constructor and Destructor

See Also   TUnknown

Constructor

```
TUnknown();
```

Destructor

```
virtual ~TUnknown();
```

Description

These members are protected because only a derived class should be able to construct a *TUnknown* object. *TUnknown* is meant to be a base for other objects, not an independent object.

Constructor

Creates a *TUnknown* object with an initial reference count of 0. Initially the object is not aggregated with any other object.

Destructor

Called when the object's reference counnt reaches zero. The destructor is declared **virtual** to invoke the destructors of derived classes.

See Also
TUnknown::Aggregate

TUnknown::operator IUnknown&()

See Also   TUnknown

Syntax

```
operator IUnknown&();
```

Description

Returns a reference to the object's outer *IUnknown* interface. Does not increment the object's reference count.

See Also
TUnknown::operator IUnknown*

TUnknown::SetOuter

See Also TUnknown

Syntax

```
IUnknown* SetOuter(IUnknown* outer);
```

Description

Tells the object to aggregate itself under the object *outer*. When asked for its *IUnknown* interface, **this** always returns *outer*. *SetOuter* returns the object's own *IUnknown* interface to the outer object. It does not call AddRef before returning the pointer.

If *outer* is 0, *SetOuter* ignores *outer* but still returns its own *IUnknown* interface.

SetOuter is called to make the object aggregate under an unknown outer object. If the outer object is also a *TUnknown*, call Aggregate instead. *Aggregate* sets the object's inner pointer as well as its outer pointer.

See Also

TUnknown::Aggregate

TUnknown::GetOuter

TUnknown::Outer

TUnknown::QueryObject

See Also TUnknown

Syntax

```
virtual HRESULT QueryObject(const GUID far& iid, void far* far* pif);
```

Description

Asks whether the object supports the interface identified by *iid*. If the object supports the interface, the function returns HR_NOERROR and places a pointer to the interface in *pif*.

The implementation of *QueryObject* in *TUnknown* always fails. It always returns HR_NOINTERFACE. Classes derived from *TUnknown* must override this function.

See Also
HR_xxxx Return Constants

TUnknown::ThisUnknown

TUnknown

Syntax

```
IUnknown& ThisUnknown();
```

Description

Returns a reference to the *IUnknown* interface for **this**, not to the outer or inner aggregated objects.

TUnknown::Outer

See Also TUnknown

Syntax

IUnknown* Outer;

Description

Holds a pointer to the *IUnknown* interface of the outer object in a group of aggregated objects.

See Also

TUnknown::GetOuter

TUnknown::SetOuter

TXAuto Class

See Also ObjectComponents Exception Classes OWL Hierarchy

Header File
ocf/autodefs.h

Base Class
TXBase

Description
TXAuto is the exception object that ObjectComponents throws when it encounters an unexpected error while processing automation calls. The possible errors are indicated by the TError nested **enum** values.

Public Constructor
```
TXAuto(TXAuto::TError err);
```

Public Data Member
```
TError ErrorCode;
```

Type Definition
```
enum TError
```

See Also

Exception Handling in ObjectComponents

TXBase (OWL.HLP)

TXObjComp

TXOle

TXRegistry

TXAuto Public Constructor

See Also TXAuto

Syntax

```
TXAuto(TXAuto::TError err);
```

Description

Constructs an exception object to describe the problem indicated by *err*.

See Also
TXAuto::TError enum

TXAuto::TError enum

See Also TXAuto

Syntax
`enum TError`

Description
The values of the enumeration identify possible errors that can occur during automation.

Constant	Meaning
xNoError	No error occurred.
xConversionFailure	Problem converting a value from a VARIANT union to the expected data type.
xNotIDispatch	Attempted to send an automation command to an object that does not execute commands.
xForeignIDispatch	Attempted to send an automation command to an automated object that does not derive from TAutoProxy.
xTypeMismatch	A supplied argument cannot be converted to the required type.
xNoArgSymbol	A command attempted to use more arguments than the server recognizes.
xParameterMissing	An automation call failed to provide a required argument when setting a property value.
xNoDefaultValue	A parameter is missing and no default value was supplied.
xValidateFailure	The code in a user-defined validation hook indicated that the argument values it received are unacceptable.

See Also
Automation Hook Macros

TXAuto::ErrorCode

See Also TXAuto

Syntax

```
TError ErrorCode;
```

Description

Holds the code that identifies the problem this object was constructed to describe.

See Also
TXAuto::TError enum

TXObjComp Class

See Also ObjectComponents Exception Classes

Header File
ocf/ocdefs.h

Base Class
TXBase

Description
TXObjComp is the exception object that ObjectComponents throws when it encounters an unexpected error while processing its own internal code. The possible errors are indicated by the TError nested **enum** values.

Public Constructor
```
TXObjComp(TXObjComp::TError err, const char* msg = 0);
```

Public Member Function
```
TError ErrorCode;
```

Type Definition
```
enum TError;
```

See Also
Exception Handling in ObjectComponents
TXAuto
TXBase
TXOle
TXRegistry

TXObjComp Public Constructor

See Also TXObjComp

Syntax

```
TXObjComp(TXObjComp::TError err, const char* msg = 0);
```

Description

Constructs an exception object to describe the problem indicated by *err*. Associates the optional *msg* string with the error.

See Also
TXObjComp::TError

TXObjComp::TError enum

TXObjComp

Syntax

```
enum TError
```

Description

The values of the enumeration identify possible errors that can occur inside ObjectComponents.

Application Errors:

Constant	Meaning
xNoError	No error occurred.
xBOleLoadFail	The BOCOLE support library could not be loaded.
xBOleBindFail	ObjectComponents could not get a necessary interface from the BOCOLE support library.
xDocFactoryFail	TOcApp was unable to register or unregister the application with OLE.
xRegWriteFail	The registrar could not write to the system registration database.

Document and Part Errors:

Constant	Meaning
xMissingRootIStorage	The document where a part was asked to construct itself does not possess a root storage object. (Without a storage, the document has nowhere to store its parts.)
xInternalPartError	ObjectComponents was unable to create a part object.
xPartInitError	ObjectComponents was unable to initialize a newly created part.
xDocSaveError	A TOcDocument could not write itself to a file.

Storage Errors:

Constant	Meaning
xStorageOpenError	A document was unable to open its storage object.
xStreamOpenError	A document was unable to open the stream object it needs for file I/O.
xStreamWriteError	A document was unable to write to the stream object it needs for file I/O.

TXObjComp::ErrorCode

See Also TXObjComp

Syntax

```
TError ErrorCode;
```

Description

Holds the error code that identifies the problem this object was constructed to describe.

See Also
TXObjComp::TError enum

TXOle Class

See Also ObjectComponents Exception Classes OWL Hierarchy

Header File
ocf/oleutil.h

Base Class
TXBase

Description
TXOle is the exception object that ObjectComponents throws when it encounters an unexpected error while executing an OLE API call.

The object's *Check* method is static so that you can call it without actually creating a *TXOle* object. If the parameters you pass indicate an error has occurred, *Check* creates a *TXOle* object and throws the exception for you.

Public Constructors and Destructor
```
TXOle(const char far* msg, HRESULT stat);
TXOle(const TXOle& copy);
~TXOle();
```

Public Member Functions
```
static void Check(HRESULT stat, const char far* msg);
static void Check(HRESULT stat);
```

Public Data Member
```
long Stat;
```

See Also
Exception Handling in ObjectComponents
TXAuto
TXBase (OWL.HLP)
TXObjComp
TXOle and Error Codes
TXRegistry

TXOle::Stat

TXOle

Syntax

```
long Stat;
```

Description

Stat ("status") holds the result code returned from an OLE API.

TXOle Public Constructors and Destructor

See Also TXOle

Form 1
```
TXOle(const char far* msg, HRESULT stat);
```

Form 2
```
TXOle(const TXOle& copy);
```

Destructor
```
~TXOle();
```

Description

Usually you do not need to construct an OLE exception object directly. Call Check instead.

Form 1:

Creates an OLE exception object. *msg* points to an error message and *stat* holds the return value from an OLE API call.

Form 2:

Constructs a new OLE exception object by copying the one passed as *copy*.

Destructor

Destroys the *TXOle* object.

See Also

TXOle::Check

TXOle::Check

TXOle

Syntax

Form 1
```
static void Check(HRESULT stat, const char far* msg);
```

Form 2
```
static void Check(HRESULT stat);
```

Description

Checks whether an error has occurred and if so throws an exception. *stat* is the value returned by an OLE API call. *Check* is static so that you can call it without actually creating a *TXOle* object first. If *stat* indicates an error, then *Check* creates a *TXOle* object and throws an exception.

Form 1: If *stat* indicates an error, Form 1 throws a *TXOle* exception containing the *msg* error string.

Form 2: If *stat* indicates an error, Form 2 throws a *TXOle* exception containing the error string "OLE call FAILED, ErrorCode = *stat*" where *stat* is shown as an eight-digit hexadecimal value.

If you see this error message when running programs, you can look it up in the OLE_ERRS.TXT file, which for convenience matches the error codes to corresponding comments from the OLE system header files.

TXRegistry Class

See Also ObjectComponents Exception Classes OWL Hierarchy

Header File
ocf/ocdefs.h

Base Class
TXBase

Description
TXRegistry is the exception object that ObjectComponents throws when it encounters an unexpected error while reading from or writing to the system registration database.

The object's *Check* method is static so that you can call it without actually creating a *TXRegistry* object. If the parameters you pass indicate an error has occurred, *Check* creates a *TXRegistry* object and throws the exception for you.

Public Constructors
```
TXRegistry(const char* msg, const char* key);
TXRegistry(const TXRegistry& copy);
```

Public Member Functions
```
static void Check(long stat, const char* key);
const char* Key;
```

See Also

Exception Handling in ObjectComponents

TXAuto

TXBase (OWL.HLP)

TXObjComp

TXOle

TXRegistry Public Constructors

See Also		TXRegistry

Form 1
```
TXRegistry(const char* msg, const char* key);
```

Form 2
```
TXRegistry(const TXRegistry& copy);
```

Description

Usually you do not need to construct a registry exception directly. Call Check instead.

Form 1: Creates a registry exception object. *msg* points to an error message and *key* points to the name of the registry key that ObjectComponents was processing when the exception occurred.

Form 2: The copy constructor constructs a new registry exception object by copying the one passed as *copy*.

See Also
TXRegistry::Check

TXRegistry::Check

TXRegistry

Syntax

```
static void Check(long stat, const char* key);
```

Description

Tests the value of *stat* to determine if an error has occurred and if so throws an exception. *stat* is the return value from a registration command. *key* is the name of the key that the registration command was processing.

Check is static so that you can call it without actually creating a *TXRegistry* object first. If *stat* is nonzero, then *Check* creates a *TXRegistry* object and throws an exception. The exception carries the message string "Registry failure on key: *key*, ErrorCode = *stat*."

TXRegistry::Key

TXRegistry

Syntax

```
const char* Key;
```

Description

Points to the name of the registration key that ObjectComponents was processing when the exception occurred.

typehelp Registration Key

See Also Registration Keys

Description

Registers the name of a Help file (.HLP) containing information about the methods and properties your program exposes for automation. If the file is not in the same directory as the executable, be sure to register helpdir as well.

typehelp is valid in the application registration table of an automation server. It is optional. Also, the file name can be localized, making it easy to have different Help files for different languages.

To register *typehelp*, use the REGDATA macro, passing *typehelp* as the first parameter and file name as the second parameter.

See Also
helpdir Registration Key
Localizing Symbol Names
REGDATA macro (OWL.HLP)
Registration Macros (OWL.HLP)
Registration Keys

usage Registration Key

See Also Registration Keys

Description

Determines whether a single instance of your application is allowed to support multiple users or whether a new instance should be launched for each new OLE client. The *-Automation* command-line switch overrides this setting and forces single use when an automation server is invoked.

The *usage* key is valid in any server registration table. It is always optional. If you omit it, ObjectComponents by default registers the application to support only one client per instance.

To register the *usage* key, use the REGDATA macro, passing *usage* as the first parameter and one of the *ocrxxxx* Usage constants as the second parameter.

```
REGDATA(usage, ocrSingleUse)   // one client per instance (default)
```

See Also
ocrxxxx Usage constants
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)

verb*n* Registration Keys

See Also Registration Keys

Description

A string naming an action the server can perform with its objects. Containers add the active objects verbs to their Edit menus.

verb0 is the name of the primary (default) verb for the class. The primary verb is executed if the user double-clicks the object. Use *verb1* through *verb7* to register additional verbs. The ocrVerbLimit constant, defined in ocf/ocreg.h, represents the maximum number of verbs allowed (8).

The *verbn* keys are valid in the document registration tables of a server that supports linking and embedding. Every server should register a default verb. Other verbs are optional.

To register a verb, use the REGDATA macro, passing *verbn* as the first parameter and a menu item string as the second parameter.

```
REGDATA(verb0, "&Edit")  // default action
REGDATA(verb1, "&Open")  // another possible action (optional)
```

See Also

ocrxxxx Limit Constants

REGDATA macro (OWL.HLP)

Registration Macros (OWL.HLP)

Registration Keys

verb*n*opt Registration Keys

verb*n*opt Registration Keys

See Also   Registration Keys

Description

Registers option flags describing the server's verbs. The flags determine how the verbs appear on the container's menu. They can be grayed or disabled, for example.

Verb options are valid in the document registration table of any server that supports linking and embedding. They are always optional. Verb options are meaningless unless you also register verbs.

To register verb options, use the REGVERBOPT macro, passing a verb key (such as *verb0* or *verb1)* as the first parameter. For the second parameter, use ocrxxxx Verb Menu constants. For the third parameter, use ocrxxxx Verb Attribute constants.

```
REGVERBOPT(verb2, ocrGrayed, ocrOnContainerMenu | ocrNeverDirties)
```

See Also
ocrxxxx Verb Menu Flags
ocrxxxx Verb Attributes Constants
REGVERBOPT Macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)
verb*n* Registration Keys

version Registration Key

See Also Registration Keys

Description

Registers a version string for the application and type library. The string can include minor version numbers delimited by periods. OLE ignores version numbers after the first two (the major and minor version numbers).

The version key is valid in any registration table. It is always optional.

To register *version*, use the REGDATA macro, passing *version* as the first parameter and a version number string as the second parameter.

```
REGDATA(version, "1.0.5")
```

See Also
description Registration Key
permid Registration Key
permname Registration Key
REGDATA macro (OWL.HLP)
Registration Keys
Registration Macros (OWL.HLP)

WM_OCEVENT Message

See Also Linking and Embedding Messages

Header File

ocf/ocapp.h

Description

ObjectComponents defines the WM_OCEVENT message in order to notify an application's window when significant OLE-related events occur.

Message	Meaning
WM_OCEVENT	Notification of an OLE event from ObjectComponents. The *wParam* value identifies the particular event.

See Also

Handling WM_OCEVENT

Messages and Windows

ObjectComponents Messages

OC_APPxxxx Messages

OC_VIEWxxxx Messages