ObjectWindows Reference Guide

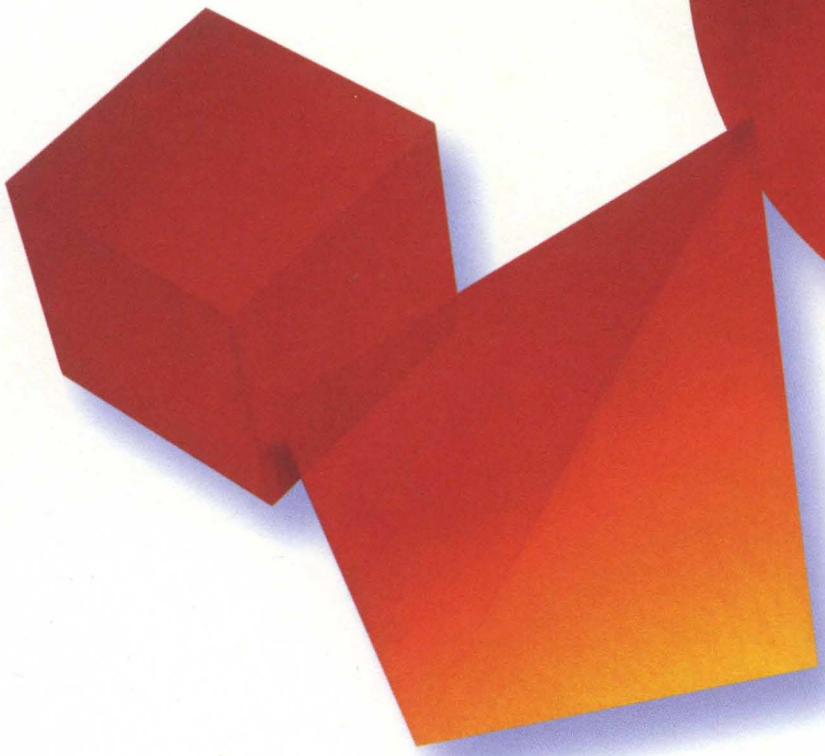
- ObjectWindows Classes
- Event-handling Functions
- Dispatch Functions

- ObjectComponents Classes
- Linking and Embedding
- Automation

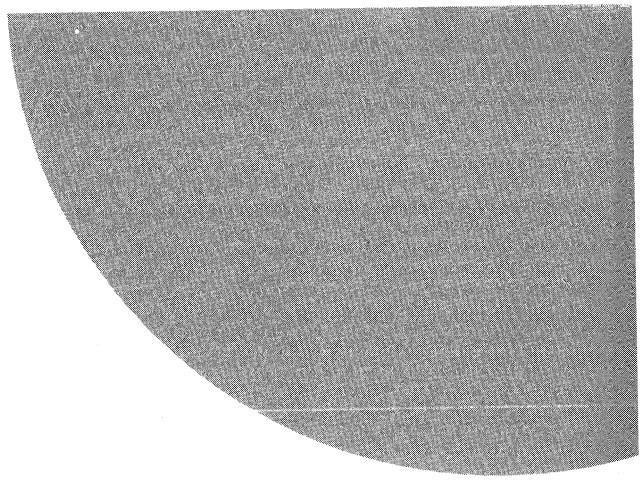
Borland

Reference Guide

VERSION 2.5



Borland® ObjectWindows®



Reference Guide

VERSION 2.5

Borland® ObjectWindows®

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Printed in the U.S.A.

1E0R1094
9495969798-9 8 7 6 5 4 3 2
H1

Contents

Tables

Figures

Introduction

This *Reference Guide* can be used to help you perform the following tasks in ObjectWindows:

- Look up the overall purpose for each class.
- Learn the details about how to use a particular ObjectWindows class and its members and functions.
- View the virtual and nonvirtual multiple inheritance relationships among ObjectWindows classes.
- Learn which classes introduce or redefine functions.
- Determine which ancestor of a class introduced a data member or member function.
- Learn how data members and member functions are declared.
- Create OLE2 applications easily by using ObjectComponents classes.
- Use the ObjectSupport Library (OSL) to support mathematical and file operations.
- Use event-handling functions to respond to messages.
- Use dispatch functions to crack Windows messages.

What's new in ObjectWindows 2.5

ObjectWindows 2.5 provides several new features that make it easier for you to design applications that run faster, write code that's easier to debug, and create programs that implement linking and embedding technology. ObjectWindows 2.5 provides the following enhancements over version 2.0:

- Complete encapsulation of OLE2 using ObjectComponents including
  - Linking and embedding containers
  - Linking and embedding servers
  - Automation servers
  - Automation controllers
  - OLE clipboard operations

  - OLE drag and drop operations
  - In-place editing
  - OLE user interface, including menu merging, pop-up menus for activated object verbs on the container's Edit menu
  - Compound file storage
  - Registration
  - Localized strings for international support
  - Type libraries
- New data type definitions. See Chapter 1 of this manual for a list of the new data type definitions.
- Internal diagnostic classes for increased debugging capabilities
- ObjectSupport classes that include new classes as well as utility classes previously included in the ObjectWindows library. See Chapter 1 of this manual for a description of the new support classes.

Contents of this manual

This manual is divided into three parts and includes two appendixes.

Part I, "ObjectWindows reference," includes the following four reference chapters:

Chapter 1, "Overview of ObjectWindows," provides an overview of the ObjectWindows classes, libraries, and header files. It organizes the classes according to functional groups and explains the purpose of each class within that group.

Chapter 2, "ObjectWindows library reference," is an alphabetical listing of all the standard ObjectWindows classes, including explanations of their purpose, usage, and members. It also describes the nonobject elements such as structures, constants, variables, and macros that classes use.

Chapter 3, "ObjectWindows event handlers," lists the ObjectWindows functions and notification codes that crack Windows messages.

Chapter 4, "ObjectWindows dispatch functions," lists all of the ObjectWindows functions that dispatch Windows messages.

Part II, "ObjectComponents reference." The second second part of this manual describes all of the ObjectComponents classes, structures, constants, types, and macros. It includes the following chapters:

Chapter 5, "Overview of ObjectComponents," provides an overview of the ObjectComponents classes, libraries, and header files. It describes the classes according to their functional groups and explains their purpose within that group.

Chapter 6, "ObjectComponents library reference," is an alphabetical listing of all the standard ObjectComponents classes, including explanations of their purpose, usage, and members. It also describes the nonobject elements such as structures, constants, variables, and macros that classes use.

Part III, "ObjectSupport reference." The third part of this manual describes all of the ObjectSupport classes, structures, constants, types, and macros. It includes the following chapters:

Chapter 7, "Overview of ObjectSupport," provides an overview of the ObjectWindows classes, libraries, and header files. It organizes the classes into functional groups and explains the purpose of each class within that group.

Chapter 8, "ObjectSupport library reference," is an alphabetical listing of all the standard ObjectSupport classes, including explanations of their purpose, usage, and members. It also describes the nonobject elements such as structures, constants, variables, and macros that classes use.

Appendix A, "Windows API encapsulated functions," lists the ObjectWindows functions that encapsulate Windows API functions.

Appendix B, "Windows API structs," lists the Windows structures that ObjectWindows uses.

Typefaces used in this manual

Boldface	Boldface type indicates language keywords (such as **char**, **switch**, and **begin**) and command-line options (such as **-rn**).
Italics	Italic type indicates program variables and constants that appear in text. This typeface is also used to emphasize certain words, such as new terms.
`Monospace`	Monospace type represents text as it appears onscreen or in a program. It is also used for anything you must type literally (such as `TD32` to start up the 32-bit Turbo Debugger).
Key1	This typeface indicates a key on your keyboard. For example, "Press *Esc* to exit a menu."
Key1+ Key2	Key combinations produced by holding down one or more keys simultaneously are represented as *Key1+Key2*. For example, you can execute the Program Reset command by holding down the *Ctrl* key and pressing *F2* (which is represented as *Ctrl+F2*).
Menu| Command	This command sequence represents a choice from the menu bar followed by a menu choice. For example, the command "File | Open" represents the Open command on the File menu.
Note	This icon indicates material you should take special notice of.

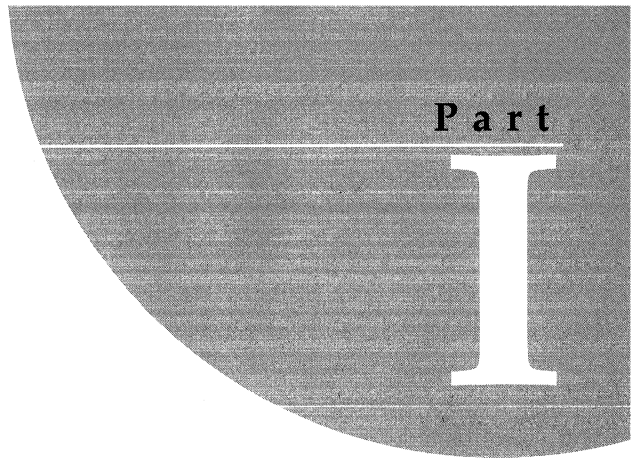
Conventions used in this manual

Cross-referenced entries to ObjectWindows functions include the class name, the scope resolution operator, and the function name. For example,

See also: *TApplication::PumpWaitingMessages*

C++ data types that are keywords (such as **int** and **long**) are in lowercase bold. Predefined Windows types (such as HWND) are in capital letters; for example,

```
bool TrackPopupMenu(uint flags, int x, int y, int rsvd, HWND wnd, TRect* rect=0);
```



ObjectWindows reference

Chapter

1

Overview of ObjectWindows

This chapter provides an overview of the ObjectWindows classes, libraries, and header files. It describes the classes according to the functional groups represented on the ObjectWindows hierarchy diagram.

Hierarchy diagram

The ObjectWindows hierarchy diagram shows the classes that are described in this manual. The classes are grouped according to functional categories, and all related classes are in one shaded unit. A rectangle surrounds the name of the class. A class is enclosed in dashed lines if it is a parent class for a multiply-inherited class. For example, *TListBox* is the parent class for *TListView*, which is derived from both *TView* and *TListBox*. Base classes are placed above inherited classes and are connected to inherited classes by straight lines. The triangle on the connecting lines indicates the type of inheritance association that exists between the classes. A filled-in triangle indicates virtual inheritance between the parent and its derived classes; an open triangle illustrates nonvirtual inheritance.

Figure 1.1 Base class with several derived classes

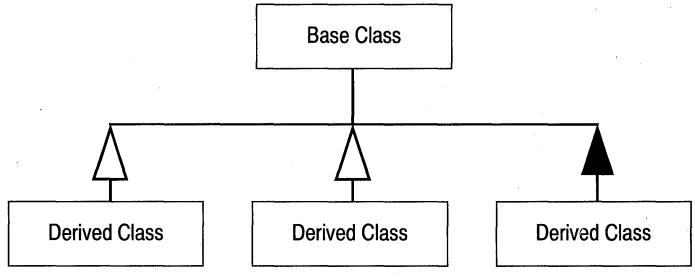


Figure 1.2 ObjectWindows hierarchy

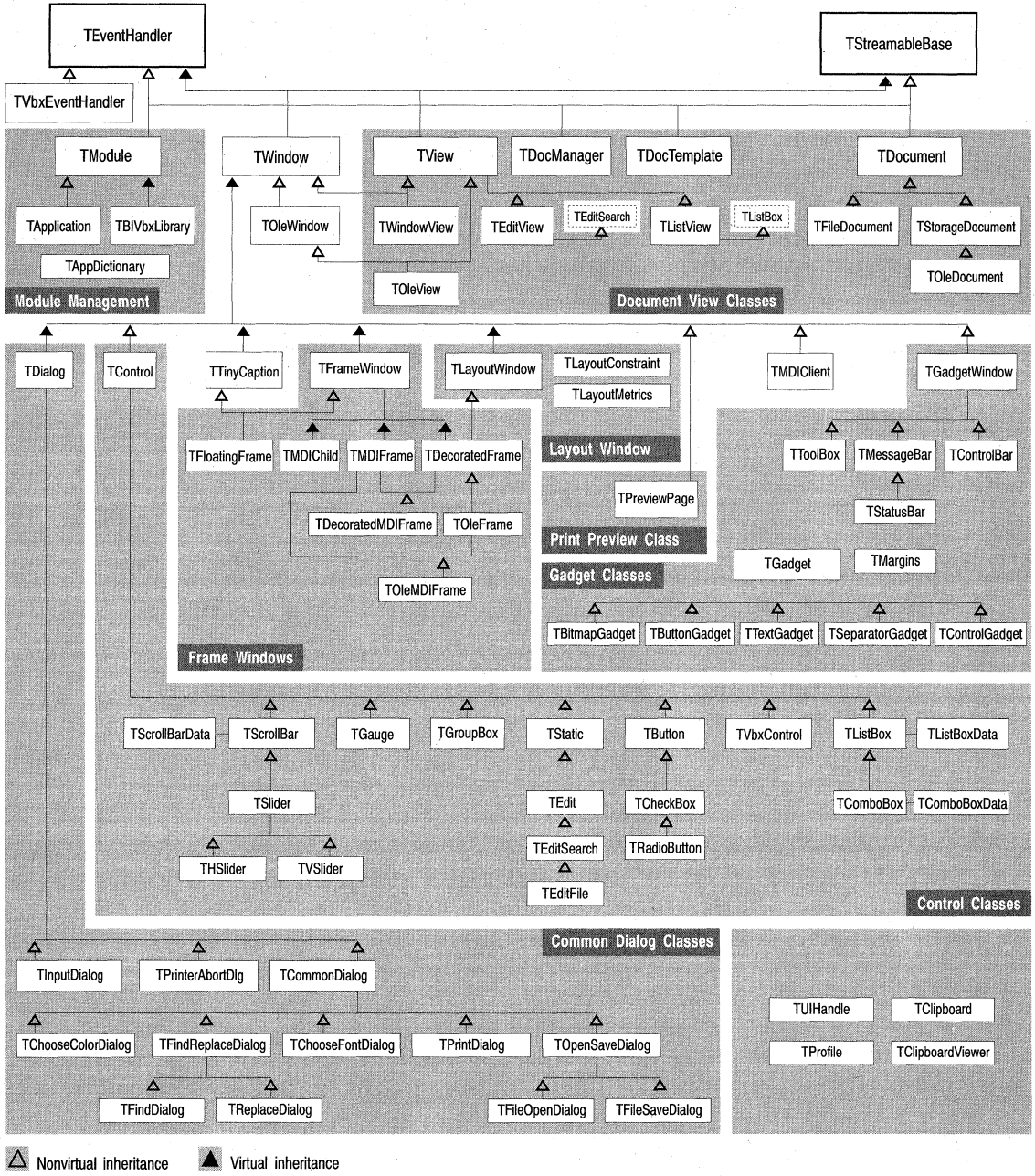
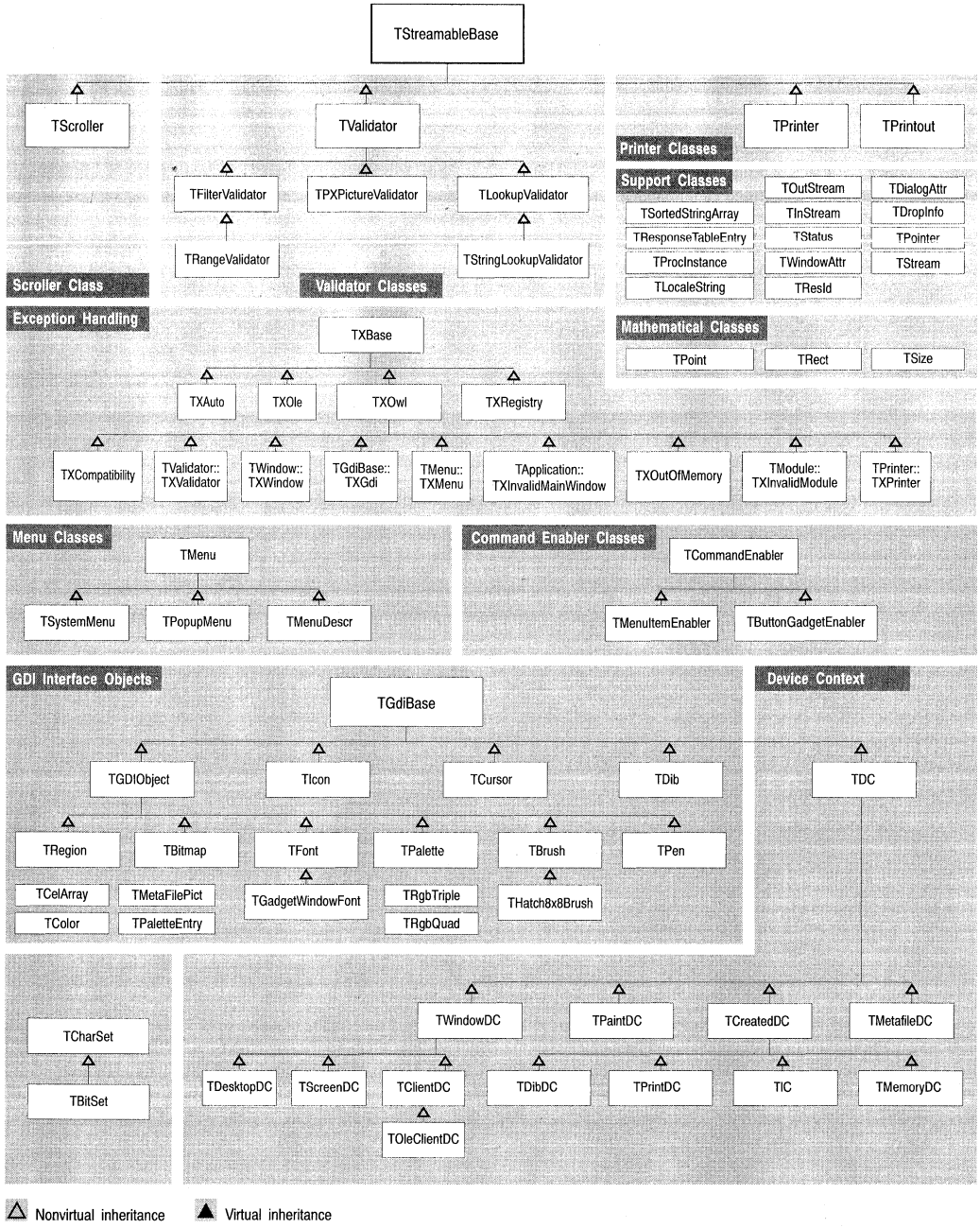


Figure 1.3 ObjectWindows hierarchy



Using ObjectWindows classes

The ObjectWindows hierarchy includes a forest of classes that you can use, modify, or derive from in order to create your own application. This section describes these groups of classes and how you can use them to build your application. These classes, which are displayed on the ObjectWindows hierarchy chart, can be divided into the following groups.

- Base classes
- Window management classes
  - Frame windows
  - Decorated windows
  - Common dialogs
  - Controls
  - Gadgets
  - Menus
- Module and application management classes
  - Command enabling
  - Doc/view
  - Print and print/preview
- Graphics classes
  - GDI objects
  - Device contexts
- Validators
- Exception handling classes
- Support classes

Base classes

TEventHandler, *TStreamableBase*, and *TGdiBase* are important base classes. All ObjectWindows classes are derived from one or more of these classes. Classes that inherit from *TEventHandler* are able to respond to window messages. Classes that inherit from *TStreamableBase* support streaming, that is their objects can write to and read from streams. Almost all of the ObjectWindows classes are derived from *TStreamableBase*. You can use multiple inheritance to derive a class from both *TEventHandler* and *TStreamableBase*. Classes that inherit from *TGDIBase*, a private base class, support GDI drawing objects such as pens, brushes, fonts, and bitmaps.

- *TEventHandler* sends messages to the appropriate message handler.
- *TStreamableBase* provides support for C++ streaming and persistence.
- *TGdiBase* is the root class for all derived GDI classes that support Windows' GDI library.

Window management classes

Derived from *TEventHandler and TStreamableBase, TWindow* is the parent class for all window classes. It represents the functionality common to all windows, whether they are dialog boxes, controls, multiple document interface (MDI) windows, or layout windows. One of the fundamental ObjectWindows classes that implements OLE functionality, *TOleWindow* provides support for embedding objects in a compound document application.

Frame windows

A frame window, which is actually an application's main window, has the ability to contain other client windows and also support UI elements such as menus and icons. Serving as main windows of MDI-compliant applications, MDI frame windows manage multiple documents or windows in a single document (SDI) application. ObjectWindows also provides OLE support for both SDI and MDI applications. A floating frame window provides the same functionality but lets you position the window anywhere within the parent window.

- *TFrameWindow* adds special functionality designed to simplify the managment of main windows.
- *TFloatingFrame,* derived from *TFloatingFrame* and *TTinyCaption,* provides the functionality of a frame window enhanced with a tiny caption bar.
- *TMDIChild* defines the behavior of MDI child windows.
- *TMDIFrame* provides support for frame windows designed to be used as MDI windows.
- *TOleFrame* provides OLE support for the main window of an SDI application.
- *TOleMDIFrame* provides OLE support for the main window of an MDI application.

Mix-in window classes such as *TLayoutWindow* and *TClipboardViewer* add the special functionality of layout capabilities and clipboards to the main window classes. Use *TLayoutWindow* to design the placement of a window on the screen and *TClipboardViewer* to view the data shared between applications.

Decorated windows

Multiply inherited from *TFramewindow* and *TLayoutWindow,* decorated window classes let you add decorated control bars, and status bars to the frame of a window and adjust the child window to accommodate the placement of these decorations.

- *TDecoratedFrame* is basically a frame window with added decorations.
- *TDecoratedMDIFrame* is an MDI frame window that supports decorations.

Common dialogs

TDialog lets you create specialized windows referred to as dialog boxes. Dialog boxes typically ask users for information about fonts, colors, files, printing options, or searching and replacing text. Depending on their purpose, dialog boxes can be either

modal, those which prevent a user from selecting other windows, or modeless, those which permit a user to select other windows.

You can create your own customized dialog boxes or use one of the ObjectWindows classes that encapsulates Windows' common dialog boxes. The following common dialog classes are derived from *TCommonDialog* which is itself derived from *TDialog*, the base dialog box class.

- *TChooseFontDialog* objects represent modal dialog boxes allow font selection, style, point, size, and color.
- *TChooseColorDialog* objects represent modal dialog boxes that allow color selection and custom color adjustment.
- *TOpenSaveDialog* is the base class for modal dialog boxes that let you open and save a file under a specified name.
- *TPrintDialog* displays a modal print or a printer setup dialog box.
- *TFindReplaceDialog* is the base class for modeless dialog boxes that let you search for and replace text.

Controls

The control classes support standard Windows controls such as list boxes, combo boxes, group boxes, check boxes, scroll bars, buttons, radio buttons, edit controls, and static controls.

Although most windows come with scroll bars already installed, you can use *TScrollBar* to create a standalone vertical or horizontal scroll bar, for example, as a dialog box control.

Unlike standard Windows controls, ObjectWindows supports widgets, specialized controls written entirely in C++. The widget classes ObjectWindows provides include support for sliders, controls that are used for providing nonscrolling position information, and gauges, controls that provide duration or analog information about a particular process.

- TSlider defines the basic behavior of sliders.
- *THSlider* implements horizontal sliders.
- *TVSlider* implements vertical sliders.
- *TGauge* defines the basic behavior of gauge controls.

Gadgets

TGadget is the base class for several derived classes that support gadget objects that belong to a gadget window, have borders and margins, and their own style attributes. Derived from *TWindow*, *TGadgetWindow* maintains a list of gadgets, controls the display of the gadgets, and sends the necessary messages to the gadgets.

Additional gadget classes derived from *TGadgetWindow* such as *TToolBox*, *TMessageBar*, *TStatusBar*, and *TControlBar* manipulate gadgets in different ways so that you can enhance a bar or tool box attached to a frame window.

- *TToolBox* lets you place a set of gadgets in a matrix of columns and rows.

- *TMessageBar* implements a message bar with one text gadget.
- *TStatusBar* lets you include multiple text gadgets and different border styles in a status bar.
- *TControlBar* implements a control bar that provides a set of buttons on a bar in a frame window.

Menus

TMenu and its derived classes let you construct, modify, and create menu objects. The classes derived from *TMenu* include

- *TPopupMenu* lets you add a popup menu to an existing window or popup menu.
- *TSystemMenu* creates a system menu object.

Module management classes

Derived from *TModule*, *TApplication* supplies functionality common to all ObjectWindows applications. Classes derived from *TApplication* have the ability to create instances of a class, create main windows, and process messages. *TModule* defines behavior shared by both library (DLL) and application modules. Virtually derived from *TModule*, *TBiVbxLibrary* lets you add Visual Basic (VBX) controls to your application.

Doc/View classes

Doc/View class support the Doc/View model, a system in which data is contained in and accessed through a document object, and displayed and manipulated through a view object. Any number of views can be associated with a particular document type. Various classes control the flow of information within this system. Several classes also provide support for OLE's compound document and compound file structure within the Doc/View model.

TDocManager is the base class designed to handle documents, templates, messages and so on.

- *TDocument* is an abstract base class that serves as an interface between the document, its views and its document manager.
- *TStorageDocument* supports OLE's compound file structure and lets you create compound documents with embedded objects.
- *TOleDocument* implements the document half of an OLE-enabled Doc/View application.
- *TView* is the base class that displays the document's data and gets user input.
- *TListView* supports views for list boxes.
- *TOleView* supports the view half of an OLE-enabled Doc/View application.

Printer classes

TPrinter, *TPrintout*, and *TPreviewPage* provide various functions that make it easy for you to set up a printer dialog box, view a document in a print preview window, and print a document.

- *TPrinter* represents the physical printer device.
- *TPrintout* represents the physical printed document sent to the printer
- *TPreviewPage* displays a page of a document in a print preview window.

Graphics classes

ObjectWindows GDI classes encapsulate Windows' Graphics Device Interface (GDI) to make it easier to use device context (DC) classes and GDI objects. The GDI library supports device independent drawing operations using DIBS (device independent bitmaps).

DC classes

Instead of drawing directly on a device (like the screen or a printer), you can use GDI classes to draw on a bitmap using a device context (DC). A *device context* is a structure that contains information about the drawing attributes (pens, brushes, text color, and so on) of a particular device. DC classes support a variety of device context operations.

- *TDC* is the root class for GDI DC wrapper classes.
- *TWindowDC* and its derived classes such as *TClientDC* and *TScreenDC* provide access to the area owned by a window.
- *TCreatedDC* and its derived classes provide access to various DCs that are created and deleted such as memory and print DCs.

GDI classes

ObjectWindows graphics library contains several classes that you can use to create DIBS, brushes, palettes, pens, and other drawing tools.

- *TGdiBase* is the private base class from which *TGDIObject*, *TIcon*, *TCursor*, and *TDib* are derived.
- *TGDIObject* is a base class for several other GDI classes that support drawing tools.
- *TDib* encapsulates the creation of structures containing format and palette information.
- *TCursor* encapsulates GDI cursor objects.

Validator classes

TValidator forms the base class for several ObjectWindows classes that encapsulate validation objects. The following derived classes make it easy for you to add data validation to your applications.

- *TFilterValidator* and its derived class, *TRangeValidator*, check an input field as the user types data into the field in order to determine the validity of the entered data.
- *TPXPictureValidator* compares user input with a picture of a data format.
- *TLookupValidator* compares a string typed by a user with a list of acceptable values.

Exception handling classes

Exception handling classes provide various functions that help you write error-free ObjectWindows applications. *TXBase* is the base class for all ObjectWindows and ObjectComponents classes. Derived from the *TXBase* class, *TXOwl* is the base class for the following ObjectWindows exception classes:

- *TXCompatibility* is included for backward compatibility with ObjectWindows 1.0 code.
- *TXOutOfMemory* describes exceptions that arise from out of memory conditions.
- Nested exception classes such as *TXInvalidMainWindow*, *TXInvalidModule*, *TXWIndow*, *TXMenu*, *TXValidator*, *TXGdi*, and *TXPrinter* describe specific error conditions such as those that occur when a main window, a module, a menu object, a validator object, a GDI object, or a printer device context is invalid.

Command enabling classes

Although several ObjectWindows classes process commands, there are three classes specifically devoted to enabling and disabling the commands available to an application.

- *TCommandEnabler* is the base class from which *TButtonGadgetEnabler* and *TMenuItemEnabler* are derived.
- *TButtonGadgetEnabler* enables and disables button gadgets.
- *TMenuItemEnabler* enables and disables menu options and places check marks by menu options.

ObjectSupport library classes

ObjectSupport classes provide various services that help you design your ObjectWindows application. For example, the class *TLocaleString* localizes OLE registration information required for containers and servers. These classes include the following groups:

- Mathematical classes such as *TPoint*, *TSize*, and *TRect* that define screen coordinates and properties of rectangles.
- Registration and localization classes such as *TRegList* and *TLocaleString* simplify the process of registering OLE containers and servers.
- Document template classes that make it easier to design Doc/View applications.

The following table lists the files included in the Object Support Library (..\OSL directory).

Table 1.1 Summary of the ObjectSupport library files

File name	Class definition	Use
defs.h		Contains common definitions, including windows.h definitions, and deals with BOOL data types.
doctpl.h	TDocTemplate	Creates the Doc/View classes.
	TDocTemplateT<D,V>	Registers the associated document and view classes.
except.h	TXBase	Base exception-handling class for ObjectWindows and ObjectComponents classes.
geometry.h	TDropInfo	Supports file name drag and drop operations.
	TPoint, TSize, TRect	Mathematical classes.
	TPointer	Provides exception-safe pointer manipulation.
	TProcInstance	A Win16 support class.
	TResId	A resource ID.
locale.h	TLocaleString	Localizable substitute for char*.
	TRegItem	An item for the system registry.
	TRegList	List of registration items.

ObjectWindows Libraries

The following tables list the ObjectWindows static and dynamic libraries, their uses, and the operating system under which the library is available. These files are in your library directory.

The name of the OWLWx.LIB file varies, depending on several factors—whether you are building a small, medium, or large memory model application or a WIN16 or WIN32 application. For example, if the application is built for a 16-bit, small memory model, the name of the library file is OWLWS.LIB. If you're building a flat model WIN32 application, the name of the library file is OWLWF.LIB where "F" indicates a flat model application. If runtime diagnostics are enabled, ObjectWindows adds "D" to the name of the libary.

Different versions of these files are included on your installation disk. If the diagnostic files are not shipped, you can build these files by adding the switch –DDIAGS to the ObjectWindows makefile located in your ..\EXAMPLES subdirectory.

Table 1.2 Summary of static libraries

File name	Application	Use
OWLWS.LIB	Win16	16-bit small model
OWLWM.LIB	Win16	16-bit medium model
OWLWL.LIB	Win16	16-bit large model
OWLWI.LIB	Win16	16-bit import library for OWL250.DLL
OWLDWS.LIB	Win16	16-bit diagnostic small model

Table 1.2 Summary of static libraries (continued)

File name	Application	Use
OWLDWM.LIB	Win16	16-bit diagnostic medium model
OWLDWL.LIB	Win16	16-bit diagnostic large model
OWLDWI.LIB	Win16	16-bit diagnostic import library
OWLWIU.LIB	Win16	16-bit large static for user .DLL
OWLWLU.LIB	Win16	16-bit import static for user .DLL
OWLWF.LIB	Win32, Win32s	32-bit library
OWLWFI.LIB	Win32, Win32s	32-bit import library for OWL250F.DLL
OWLDWF.LIB	Win32, Win32s	32-bit diagnostic library
OWLDWFI.LIB	Win32, Win32s	32-bit diagnostic import library

The dynamic-link library (DLL) versions of ObjectWindows are contained in the \BIN subdirectory of the installation. The following table lists the DLL names and uses, and the operating system under which each library is available.

Table 1.3 Summary of dynamic link libraries

File name	Application	Use
OWL250.DLL	Win 16	16-bit dynamic library
OWL250F.DLL	Win 32	32-bit dynamic library
OWL250D.DLL	Win 16	Diagnostic version of 16-bit dynamic library
OWL250DF.DLL	Win 32	Diagnostic version of 32-bit dynamic library

Compiler options for building and using ObjectWindows libraries

You need to use different compiler options depending on whether you are building or using ObjectWindows DLLs or static libraries. Unless you specify otherwise, ObjectWindows makes several assumptions about the default values for system platforms and memory models. That is, ObjectWindows assumes that the platform is win16 unless MODEL is explicitly set to "f," in which case ObjectWindows assumes that the platform is Win32. The default MODEL setting is "d," where "d" indicates that you are building the DLL version of an library.

The following table lists the combinations of SYSTEM and MODEL settings you can use to build the specified target applications.

Table 1.4 Target applications and compiler options

For this target application:	Use SYSTEM=	Use MODEL=
16-bit Windows small model static version	WIN16	s
16-bit Windows medium model static version	WIN16	m
16-bit Windows compact model static version	WIN16	c
16-bit Windows large model static version	WIN16	l
16-bit Windows large model DLL	WIN16	d
32-bit Windows static version	WIN32	f
32-bit Windows DLL	WIN32	d

Building ObjectWindows libraries

If you are building ObjectWindows DLLs and libraries, you need to use several predefined macros. For example, defining the make macro USERDLL builds ObjectWindows for use in a user DLL and adds the suffix, "U" to the name of the library. The preprocessor macro _BUILDOWLDLL, which must be defined to build the ObjectWindows DLL, sets the values for the _OWLCLASS, _OWLDATA, _OWLFUNC macros.

The following table lists the make options you need to use if you are building either 16- or 32-bit ObjectWindows. You can specify the system model as either s (small), m (medium), l (large), f (flat), or d (DLL). The make options you set are then responsible for generating the specified preprocessor macro, which, in turn, generates the indicated values for the _OWLCLASS, _OWLDATA, and _OWLFUNC macros and builds the appropriate library. For an example of how these settings are used, see the makefile in the \SOURCE\OWL directory or owldefs.h in the INCLUDE\OWL directory.

Table 1.5 Summary of options for building an ObjectWindows static or dynamic library

If you are building an:	Use these Make options:	Preprocessor macro	_OWLCLASS _OWLDATA _OWLFUNC	Libraries
ObjectWindows DLL				
16-bit EXE	MODEL = d	_BUILDOWLDLL	All are defined as __export.	OWLWI.LIB
32-bit EXE or DLL	MODEL = d –DWIN32	_BUILDOWLDLL	All are defined as __export	OWLWFI.LIB
	MODEL = d –D USERDLL	_BUILDOWLDLL	All are defined as __export.	OWLWIU.LIB
16-bit DLL				
ObjectWindows static library				
16-bit EXE	MODEL = s or m or l	Nothing	Nothing	OWLWS.LIB OWLWM.LIB OWLWL.LIB
32-bit EXE or DLL	MODEL = f –DWIN32	Nothing	Nothing	OWLWF.LIB
16-bit DLL	MODEL = l –DUSERDLL	Nothing	Nothing	OWLWLU.LIB

Using ObjectWindows libraries

This table lists the make options you need to specify if you are using either 16- or 32-bit ObjectWindows applications. You can specify the memory model as either s (small), m (medium), l (large), f (flat), or d(DLL). (Keep in mind that the make options SYSTEM=WIN32 and –DWIN32 are the same.) The make options you set are then responsible for generating the specified preprocessor macro, which, in turn, generates the indicated values for the _OWLCLASS, _OWLDATA, and _OWLFUNC macros and

builds the appropriate library. For an example of how these settings are used, see MAKEFILE.GEN in the \OWL\EXAMPLES directory.

Table 1.6 Summary of options for using an ObjectWindows static or dynamic library

If you are using an:	Use these Make options:	Preprocessor macro	_OWLCLASS _OWLDATA _OWLFUNC	Libraries
ObjectWindows DLL				
16-bit EXE	MODEL = d	_OWLDLL	All are defined as __import	OWLWI.LIB
32-bit DLL	MODEL = d –DWIN32	_OWLDLL	All are defined as __import	OWLWFI.LIB
16-bit DLL	MODEL = d	_OWLDLL	All are defined as __import	OWLWIU.LIB
ObjectWindows static library				
16-bit EXE	MODEL = s or m or l	Nothing	Nothing	OWLWS.LIB OWLWM.LIB OWLWL.LIB
32-bit EXE or DLL	MODEL = f –DWIN32	Nothing	Nothing	OWLWF.LIB
16-bit DLL	MODEL = l	Nothing	Nothing	OWLWLU.LIB

The following table lists the makefile and compiler options for the _OWLFARVTABLE macro, which moves ObjectWindows virtual function tables (vtables) out of the DGROUP of the data segment and stores them in the code segment.

Table 1.7 Compile options for _OWLCLASS macro

Use this compile option:	Define in your makefile:	With this result:
_OWLFARVTABLE _BIDSFARVTABLE _RTLFARVTABLE	OWLFARVTABLE	Adds _huge to the _OWLCLASS class modifier when static models are used.
_FASTTHIS	Doesn't apply	Adds _fastthis to the _OWLCLASS macro.

The ObjectWindows header files

Header files contain prototype declarations for class functions, and definitions for data types and symbolic constants.

Table 1.8 Summary of header files

File name	Class definition	Use
Directory of \INCLUDE\OWL		
appdict.h	TAppDictionary	Contains a set of assoications between an application and a process ID.
applicat.h	TApplication	Controls the basic behavior of all ObjectWindows applications.

Table 1.8 Summary of header files (continued)

File name	Class definition	Use
bitmapga.h	TBitmapGadget	A set including but no more than 256 items managed by bits.
bitset.h	TBitSet	Sets or clears one or more bits.
	TCharSet	A set of characters
button.h	TButton	Creates different types of button controls.
buttonga.h	TButtonGadget	Creates button gadgets that can be clicked on or off.
celarray.h	TCelArray	Creates an array of bitmap cels.
checkbox.h	TCheckBox	Represents a check box control.
chooseco.h	TChooseColor	Represents modal dialog boxes that allow color selection.
choosefo.h	TChooseFont	Represents modal dialog boxes that allow font selection.
clipboar.h	TClipboard	Contains functions that control how Clipboard data is handled.
clipview.h	TClipboardViewer	Registers a TClipboardViewer as a Clipboard viewer.
color.h	TColor	Contains functions used to simplify standard Windows color operations.
combobox.h	TCombobox	Creates combo boxes or combo box controls in a window, and class TComboBoxData, which is used to transfer data between combo boxes.
commdial.h	TCommonDialog	Abstract base class for TCommonDialog objects.
compat.h		Defines functions and constants used internally by ObjectWindows.
control.h	TControl	Used to create control objects in derived classes.
controlb.h	TControlBar	Implements a control bar that provides mnemonic access for its button gadgets.
controlg.h	TControlGadget	Allows controls to be placed in a gadget window.
dc.h	TBandInfo, TClientDC, TCreatedDC, TDC, TDesktopDC, TDibDC TIC, TMemoryDC, TMetaFileDC, TPaintDC TPrintDC, TScreenDC, TWindowDC	GDI DC wrapper classes that create DC objects.
decframe.h	TDecoratedFrame	Creates a client window into which decorations can be placed.
decmdifr.h	TDecoratedMDIFrame	Creates a frame object that supports decorated child windows.
dialog.h	TDialog	Creates modal and modeless dialog box interface elements.
	TDialogAttr	Holds the dialog box element's attributes.
dispatch.h		Defines dispatch functions designed to crack Windows messages.
docmanag.h	TDocManager	Creates a document manager object that manages the documents and templates.
doctpl.h	TDocTemplate	Creates the templates.
	TDocTemplateT	Registers the associated document and view classes
docview.h	TDocument, TView, TWindowView, TStream, TInStream, TOutStream	Create, destroy, and send messages about document views. Define streams for documents.

Table 1.8 Summary of header files (continued)

File name	Class definition	Use
edit.h	TEdit	Creates an edit control interface element.
editfile.h	TEditFile	Creates a file editing window.
editsear.h	TEditSearch	Creates an edit control that responds to search and replace commands.
editview.h	TEditView	View wrapper for TEdit.
eventhan.h	TEventHandler	Used to derive class capable of handling messages.
except.h	TXBase	Base exception-handling class for ObjectWindows and ObjectComponents classes.
	TXOwl	
	TXCompatibility	Base exception-handling class for ObjectWindows classes.
	TXOutOfMemory	Included for backward compatibility.
	TStatus	Describes an out-of-memory exception.
		Describes a status exception.
filedoc.h	TFileDocument	Opens and closes document views.
findrepl.h	TFindDialog, TFindReplaceDialog::	These classes create and define the attributes of modeless dialog boxes that respond to search and replace commands.
floatfra.h	TFloatingFrame	Implements a floating frame within a parent window.
framewin.h	TFrameWindow	Controls window-specific behavior such as keyboard navigation and command processing.
gadget.h	TGadget	Creates gadget objects that belong to a gadget window and have specified attributes.
gadgetwi.h	TGadgetWindow	Maintains a list of tiled gadgets for a window.
	TGadgetWindowFont	Defines the font used in gadget windows.
	TSeparatorGadget	Creates a separator between gadgets.
gauge.h	TGauge	Establishes the behavior of gauge controls.
gdibase.h	TGdiBase	Abstract base class for all GDI classes.
gdiobjec.h	TGdiObject	Base GDI class.
	TPen, TBrush, TFont TPalette, TBitmap, TIcon, TCursor, TDib, TRegion	These classes create specified GDI objects.
geometry.h	TPoint, TSize, TRect	Mathematical classes.
	TDropInfo	Supports file name drag and drop operations.
	TProcInstance	A Win16 support class.
	TPointer	Provides exception-safe pointer manipulation.
	TResId	Creates a resource ID.
groupbox.h	TGroupBox	Creates a group box object that represents a group box element in Windows.
inputdia.h	TInputDialog	Creates a generic dialog box that accepts text.
layoutco.h	TLayoutConstraint	Creates layout constraints.
layoutwi.h	TLayoutMetrics	Contains the layout constraints used to define the layout metrics for a window.
	TLayoutWindow	Provides layout options for a window.
listbox.h	TListBox	Creates a list box object.
	TListBoxData	Used to transfer the contents of a list box.

Table 1.8 Summary of header files (continued)

File name	Class definition	Use
listview.h	TListView	Provides views for list boxes.
locale.h	TLocaleString	Localizable substitute for char*.
mdi.h	TMDIClient	Manages MDI child windows.
	TMDIFrame	The main window of MDI-compliant applications.
mdichild.h	TMDIChild	Defines the behavior of MDI child windows.
menu.h	TMenu, TPopupMenu,	Create menu objects.
	TSystemMenu	Base menu class.
	TMenuDescr	A menu bar with groups.
messageb.h	TMessageBar	Implements a message bar.
metafile.h	TMetaFilePict	A wrapper class used with TMetaFileDC.
module.h	TModule	Defines the basic behavior for ObjectWindows libraries and applications.
oledoc.h	TOleDocument	Implements the Document half of the Doc/View pair.
olefacto.h	TAutoFactory<>	Template class that supports component create callbacks for automated OLE-enabled applications.
olefacto.h	TOleFactory<>	Template class that supports component create callbacks for Doc/View and non-Doc/View OLE-enabled applications.
oleframe.h	TOleFrame	Supports OLE-enabled main windows for SDI applications.
olemdifr.h	TOleMDIFrame	Supports OLE-enabled main windows for MDI applications.
oleview.h	TOleView	Supports the View half of the Doc/View pair.
olewindo.h	TOleClientDC	Translates between two different coordinate systems.
opensave.h	TOpenSave	Base class for modal open and save dialog boxes.
owlall.h		Include file for all of the ObjectWindows classes.
owlcore.h		Include file for the core ObjectWindows classes.
owldefs.h		Includes definitions of macros used by all ObjectWindows programs.
owlpch.h		Contains definitions of macros, data, and functions used by ObjectWindows.
preview.h	TPreviewPage	Displays a document page in a print preview window.
	TPrintPreviewDC	Maps printer device coordinates to screen coordinates.
printdia.h	TPrintDialog	Displays a modal print or print setup dialog box.
printer.h	TPrinter	Represents the printer device.
	TPrintout	Represents the printed document.
	TPrinterAbortDlg	Represents the printer-abort dialog box.
radiobut.h	TRadioButton	Create a radio button control.
scrollba.h	TScrollBar	Represents a vertical or horizontal scroll bar control.
	TScrollBarData	Contains the values of the thumb position on the scroll bar.
scroller.h	TScroller	Implements automatic window scrolling.
signatur.h		Defines the message cracking signature templates used by ObjectWindows event-handling functions.

Table 1.8 Summary of header files (continued)

File name	Class definition	Use
slider.h	TSlider	Defines the basic behavior of sliders.
	THSlider	A horizontal slider.
	TVSlider	A vertical slider.
static.h	TStatic	Create a static control in a window.
statusba.h	TStatusBar	Constructs a status bar.
stgdoc.h	TStorageDocument	Supports compound file structure mechanisms.
textgadg.h	TTextGadget	Construct a text gadget object.
tinycapt.h	TTinyCaption	Produces a smaller caption bar for a window.
toolbox.h	TToolBox	Creates a toolbox object with a specified number of rows and columns.
uihandle.h	TUIHandle	Defines and draws UI handles.
validate.h	TValidator,	Base validator class.
	TPXPictureValidator	Picture validator.
	TFilterValidator,	Filter validator.
	TRangeValidator,	Range validatator.
	TLookupValidator,	Lookup validation.
	TStringLookupValidator	String validation.
vbxctl.h	TVbxControl	Interface for VBX controls.
	TVbxEventHandler	Handles events from VBX controls.
	TBIVbxLibrary.	Loads and initializes BIVBXxx.DLL.
version.h		Defines the internal version number of the ObjectWindows library.
window.h	TWindow	Provides window-specific behavior and encapsulates many of the Windows API functions.
windowev.h		Defines event handlers and response table macros for Windows messages.

The ObjectWindows resource files

The ObjectWindows resource files define resource and command IDs.

Table 1.9 Summary of resource files

File name	Use
Directory of \INCLUDE\OWL	
docview.rh	Defines resource and command IDs to use with docview.h and docview.rc.
edit.rh	Defines command IDs to use with edit.h.
editfile.rh	Defines resource and command IDs to use in editfile.rc and editfile.h.
editsear.rh	Defines resource and command IDs to use in editsear.rc and editsear.h.
editview.rh	Defines accelerator and menu IDs to use with TEditView.
except.rh	Defines string resource IDs to use with except.h and except.rc.
inputdia.rh	Defines resource IDs to use with inputdia.rc and inputdia.h.

Table 1.9 Summary of resource files (continued)

File name	Use
listview.rh	Defines resource and command IDs to use with listview.h.
locale.rh	Defines localization resource IDs.
mdi.rh	Defines resource and command IDs to use with mdi.h.
oleview.rh	Defines resource IDs to use with OLE-enabled views.
printer.rh	Defines resource IDs to use with printer.rc and printer.h.
slider.rh	Defines resource IDs to use with slider.h.
statusba.rh	Defines resource IDs to use with statusba.h
validate.rh	Defines resources to use with TValidator and derived classes.
window.rh	Defines command IDs to use with window.h.

The ObjectWindows data types

ObjectWindows data types have been updated to use more portable type names. The following table lists the Windows API type names, the underlying C type definitions, and the new ObjectWindows type names. To ensure that these new types are used correctly, be sure to include the ObjectWindows header files before any Windows files, such as windows.h, in your application files. The new C++ type, which maps a nonzero value to TRUE, lets you assign an integer to a **bool** type. You can then compare this Boolean value to TRUE .

Table 1.10 New ObjectWindows data types

Windows	C definition	New type	Description
	char	int8	Used when 8 bit signed value is needed
BYTE	unsigned char	uint8	Always 8 bits
WORD	unsigned short	uint16	Always 16 bits
int	int	int	16 or 32 bits depending on the platform
UINT	unsigned int	uint	16 or 32 bits depending on the platform
LONG	long	long, int32	Long (Could be 64 bits on some platforms)
ULONG	unsigned long	ulong, int32	Long (Could be 64 bits on some platforms)
	long	int32	Always 32 bits
DWORD	unsigned long	uint32	Always 32 bits
BOOL	int	bool	New C++ type if available; otherwise, emulated using an enum.
TRUE	1	true	
FALSE	0	false	

Chapter

2

ObjectWindows library reference

This chapter alphabetically lists the ObjectWindows classes, data members, member functions, macros, constants, and data types. The header file that defines each entry is listed opposite the entry name. Class members are grouped according to their access specifiers, either public or protected. Within these categories, data members, then constructors (and the destructor, if one exists), and member functions are listed alphabetically.

Because many of the properties of the classes in the hierarchy are inherited from base classes, only data members and member functions that are *new* or *redefined* for a particular class are listed. Private members are not listed. If any response table entries exist, they are also listed. The cross-referenced entries provide additional information about how to use the specified entry. The first sample entry illustrates this format.

To find information about a particular inherited member function, use the inheritance diagram included at the beginning of each class. The inheritance diagram shows the virtual overridden functions that form the interface of the class, excluding *TEventHandler* and *TStreamableBase*, from which all classes are inherited. For a list of all the inherited as well as overridden virtual functions, see the online Help.

TBird class [sample] bird.h

Type definitions

This section alphabetically lists all typedefs and enums defined within a class.

typedef unsigned short TOwlId
typedef unsigned short TOwlId;
This text explains what this typdef contains, and how you use it.

See also Related data members, member functions, classes, constants, and types

Public data members

This section alphabetically lists all public data members and their declarations, and explains how they are used.

anOwlBeak
anOwlType anOwlBeak;
anOwlBeak is a data member that holds information about this sample class. This text explains what *anOwlBeak* contains, and how you use it.

See also Related data members, member functions, classes, constants, and types

anOwlWing
anOwlType anOwlWing;
anOwlWing is another public data member.

Public constructor and destructor

This section lists any public constructors and destructor for this class. Classes can have more than one constructor; they never have more than one destructor.

Constructor
TBird(anOwlType aParameter);
Constructor for a new sample class; sets the *anOwlBeak* data member to *aParameter*.

Destructor
~TBird;
Destructor for a new sample class; destroys the *TBird* object.

Public member functions

This section alphabetically lists all public member functions that are either newly defined for this class or that are redefined inherited member functions. The description includes the purpose of each parameter and the return value of the function. If a function overrides a virtual base class function, the text specifies this:

The inline keyword isn't provided because it doesn't affect usage.

EvGetDlgCode
UINT OwlHoot();
Responds to WM_GETDLGCODE messages.

OwlHoot
void OwlHoot();
The *OwlHoot* member function causes the sample class to perform some action. This function overrides the function *OwlHoot* in its base class, *TParent*.

See also *TParent::OwlHoot*

OwlSleep
virtual int OwlSleep(int index);

The *OwlSleep* function performs another action and overrides the function *OwlSleep* in its base class, *TParent*.

See also TParent::OwlSleep

Protected data members

This section alphabetically lists all protected data members and their declarations, and explains how they are used.

anOwlFeather
anOwlType anOwlFeather;
anOwlFeather is a protected data member that holds information about this sample class.

See also Related data members, member functions, classes, constants, and types

Protected constructors

Constructor
TBird(anOwlType bParameter);
If the class has a protected constructor, it is listed here.

Protected member functions

This section lists all protected member functions.

OwlCry
BOOLEAN OwlCry;
The *OwlCry* member function causes the sample class to perform some action.

See also TSomethingElse::OwlCry

ZatsIt
virtual int ZatsIt(int index);
The *ZatsIt* function performs a particular function in class *TBird*.

Response table entries

The *TBird* response table contains this predefined macro for the EV_xxxx messages and calls this member function:

Response table entry	Member function
EV_WM_GETDLGCODE	EVGetDlgCode

BF_xxxx button flag constants

checkbox.h

Check box and radio button objects use the button flag constants to indicate the state of a selection box.

Table 2.1 Button flag constants

Constant	Meaning
BF_CHECKED	Item is checked.
BF_GRAYED	Item is grayed.
BF_UNCHECKED	Item is unchecked.

See also TCheckbox::GetCheck, TCheckbox::SetCheck

BN_xxxx button message constants

windows.h

Mouse and radio button objects use the button message constants to indicate the state of a button.

Table 2.2 Button message constants

Constant	Meaning
BN_CLICKED	Message sent when the user clicks a button
BN_DISABLE	Message sent when a button is disabled
BN_DOUBLECLICKED	Message sent when the user double-clicks a button
BN_HILITE	Message sent when the user highlights a button
BN_PAINT	Message sent when a button needs to be repainted
BN_UNHILITE	Message sent when the highlighting needs to be removed from a button

See also TRadioButton::BNClicked

CBN_xxxx combo box message constants

windows.h

Combo box objects use these message constants to indicate the state of the combo box.

Table 2.3 Combo box message constants

Constant	Meaning
CBN_CLOSEUP	Message sent when the list box of a combo box has closed
CBN_DBLCLK	Message sent when the user double-clicks a text string in the combo box
CBN_DROPDOWN	Message sent when the list box of a combo box drops down
CBN_EDITCHANGE	Message sent when the user changes text in the edit control portion of a combo box
CBN_EDITUPDATE	Message sent when edited text is going to be displayed
CBN_ERRSPACE	Message sent when the combo box is out of memory
CBN_KILLFOCUS	Message sent when the combo box loses the input focus

Table 2.3 Combo box message constants (continued)

Constant	Meaning
CBN_SELENDCANCEL	Message sent when the user's initial selection needs to be cancelled because the user has selected another control or closed the dialog box.
CBN_SELENDOK	Message sent if the user's selection is valid
CBN_SETFOCUS	Message sent when the combo box receives the input focus

See also TComboBox

CM_xxxx edit constants

window.rh

These command-based member functions are invoked in response to a particular edit menu selection or command.

Table 2.4 Command-based constants

Constant	Member function	Menu equivalent
CM_EDITCLEAR	TEdit::CMEditClear	Edit | Clear
CM_EDITCOPY	TEdit::CMEditCopy	Edit | Copy
CM_EDITCUT	TEdit::CMEditCut	Edit | Cut
CM_EDITDELETE	TEdit::CMEditDelete	Edit | Delete
CM_EDITPASTE	TEdit::CMEditPaste	Edit | Paste
CM_EDITUNDO	TEdit::CMEditUndo	Edit | Undo
CM_EXIT	TWindow::CmExit	File | Exit

See also TEdit::CMEditClear, TEdit::CMEditCopy, TEdit::CMEditCut, TEdit::CMEditDelete, TEdit::CMEditPaste, TEdit::CMEditUndo, TWindow::CmExit

CM_xxxx edit file constants

docview.rh

These command-based member functions are invoked in response to open, close, print, and save commands.

Table 2.5 Command-based constants

Constant	Member function	Menu equivalent
CM_FILECLOSE		File | Close
CM_FILENEW	TEditFile::CmFileNew	File | New
CM_FILEOPEN	TEditFile::CmFileOpen	File | Open
CM_FILEPRINT		File | Print
CM_FILEPRINTERSETUP		File | PrinterSetup
CM_FILEREVERT	TDocManager::CmFileRevert	File | Revert
CM_FILESAVE	TEditFile::CmFileSave	File | Save
CM_FILESAVEAS	TEditFile::CmFileSaveAs	File | Save As
CM_VIEWCREATE	TDocManager::CmViewCreate	File | View Create

See also TEditFile::CmFileNew, TEditFile::CmFileOpen, TDocManager::CmFileRevert, TEditFile::CmFileSave, TEditFile::CmFileSaveAs

CM_xxxx edit replace constants

editsear.rh

These command-based member functions are invoked when the corresponding find and replace command is received.

Table 2.6 Command-based constants

Constant	Member function	Menu equivalent
CM_EDITFIND	TEditSearch::CMEditFind	Edit | Find
CM_EDITFINDNEXT	TEditSearch::CMEditFindNext	Edit | Find | Next
CM_EDITREPLACE	TEditSearch::CMEditReplace	Edit | Replace

See also TEditSearch::CMEditFind, TEditSearch::CMEditFindNext, TEditSearch::CMEditReplace

CM_xxxx edit view constants

oleview.rh

These command-based view functions are invoked in response to menu and accelerator key commands. The Edit | Verbs selection refers to one of the OLE-specific menu commands, such as Edit or Open.

Table 2.7 Command-based constants

Constant	Member function	Menu equivalent
CM_EDITPASTESPECIAL	TOleWindow::CMEditPasteSpecial	Paste | Special
CM_EDITPASTELINK	TOleWindow::CMEditPasteLink	Paste | Link
CM_EDITINSERTOBJECT	TOleWindow::CMEditInsertObject	Insert | Object
CM_EDITLINKS	TOleWindow::CMEditLinks	Edit | Links
CM_EDITOBJECT	TOleWindow::CMEditObject	Edit | Object
CM_EDITFIRSTVERB	TOleWindow::EvCommandEnable	Edit | Verbs
CM_EDITLASTVERB	TOleWindow::EvCommandEnable	Edit | Verbs
CM_EDITCONVERT	TOleWindow::CMEdit	Edit | Convert

CM_xxxx MDI constants

mdi.rh

These MDI functions are invoked when the corresponding MDI command message is received.

Table 2.8 Command message constants

Constant	Member function	Menu equivalent
CM_ARRANGEICONS	TMDIClient::CmArrangeIcons	Window | Arrange Icons
CM_CASCADECHILDREN	TMDIClient::CmCascadeChildren	Window | Cascade
CM_CLOSECHILDREN	TMDIClient::CmCloseChildren	Window | Close All
CM_CREATECHILD	TMDIClient::CmCreateChild	
CM_TILECHILDREN	TMDIClient::CmTileChildren	Window | Tile
CM_TILECHILDRENHORIZ	TMDIClient::CmTileChildren	Window | Tile

See also TMDIClient::CmArrangeIcons, TMDIClient::CmCascadeChildren, TMDIClient::CmCloseChildren, TMDIClient::CmCreateChild, TMDIClient::CmTileChildren

DECLARE_RESPONSE_TABLE macro

eventhan.h

Declares a response table in the class definition. To handle events for a class, you need to both declare a response table with this macro and define the response table using one of the DEFINE_RESPONSE_TABLE macros. For example, to declare a response table, use the following declaration, where the single parameter, *Class*, represents the name of the current class:

```
DECLARE_RESPONSE_TABLE(Class);
```

ObjectWindows' response tables define the relationship between a window message and a corresponding event-handling function. The description of *TEventHandler* has more information about how ObjectWindows associates a response table entry with the appropriate function.

See also DEFINE_RESPONSE_TABLE macros, END_RESPONSE_TABLE macro, TEventHandler class

DEFINE_APP_DICTIONARY macro

appdict.h

This macro defines an *AppDictionary* reference and object as needed for use in component DLLs and EXEs. Unless a user dictionary is specified, the macro defines the dictionary as *OwlAppDictionary*, which is a globally exported *TAppDictionary*. The macro is defined as follows:

```
# define DEFINE_APP_DICTIONARY(AppDictionary)
```

See also TAppDictionary

DEFINE_DOC_TEMPLATE_CLASS macro

doctpl.h

Creates a document template. Takes three arguments: the name of the document class that holds the data, the name of the view class that displays the data, and the name of the template class, and then associates the document with one or more views. The following example illustrates how you can associate document and view classes with new template classes.

```
DEFINE_DOC_TEMPLATE_CLASS(TFileDocument, TListView, ListTemplate);
DEFINE_DOC_TEMPLATE_CLASS(TFileDocument, TEditView, EditTemplate);
```

See also TDocTemplate

DEFINE_RESPONSE_TABLE macros

eventhan.h

Defines a response table. Takes one plus *x* number of arguments: one is the name of the class that is defining the response table, and *x* is the immediate base class as well as any virtual base classes. Use the END_RESPONSE_TABLE macro to end the definition for the response table. Between the DEFINE_RESPONSE_TABLE macro and the END_RESPONSE_TABLE macro, insert the message response entries for the messages you want the class to handle. For example,

```
DEFINE_RESPONSE_TABLE1(TMyClass, TWindow)
  EV_WM_PAINT,
  EV_WM_LBUTTONDOWN,
END_RESPONSE_TABLE;
```

In this example, EV_WM_PAINT and EV_WM_LBUTTONDOWN illustrate the message response entries for the class *TMyClass* derived from *TWindow*. These macros call the corresponding event-handling functions, *EvPaint* and *EvLButtonDown*, respectively. Note that response tables are sometimes defined, but have no entries. In such cases, the base class's response table entries are searched for the appropriate event-handling function. You can also associate more than one message with an event-handling function.

The following table shows the form the DEFINE_RESPONSE_TABLE macro takes depending on the number of base classes.

Number of base classes	Macro
0	DEFINE_RESPONSE_TABLE(Class)
1	DEFINE_RESPONSE_TABLE(Class, Base)
2	DEFINE_RESPONSE_TABLE2(Class, Base1, Base2)
3	DEFINE_RESPONSE_TABLE3(Class, Base1, Base2, Base3)

See also DECLARE_RESPONSE_TABLE macros, END_RESPONSE_TABLE macro, TEventHandler class

DLGC_xxxx dialog control message constants

windows.h

Indicate the kind of input that the dialog manager needs to process. Returned by *EvGetDlgCode*.

Table 2.9 Dialog control message constants

Constant	Meaning
DLGC_BUTTON	Indicates a button
DLGC_DEFPUSHBUTTON	Indicates a default button
DLGC_HASSETSEL	Indicates a range of characters in an edit control
DLGC_RADIOBUTTON	Indicates a radio button control message
DLGC_STATIC	Indicates a static control
DLGC_UNDEFPUSHBUTTON	Indicates a non-default push button control
DLGC_WANTALLKEYS	Indicates all keyboard input
DLGC_WANTARROWS	Indicates the direction keys
DLGC_WANTCHARS	Indicates all key code messages
DLGC_WANTMESSAGE	Indicates all keyboard input can be passed on to the control.
DLGC_WANTTAB	Indicates the TAB key

See also TButton::EvGetDlgCode, TCheckBox::EvGetDlgCode, TEdit::EvGetDlgCode, TListView::EvGetDlgCode, TSlider::EvGetDlgCode

dmxxxx document manager mode constants

docmanag.h

Used by TDocManager to indicate if a document supports single or multiple open documents, and to indicate if it has file menu IDs.

Constant	Meaning
dmMenu	Sets IDs for file menu.
dmMDI	Supports multiple open documents.
dmNoRevert	Disables the File | Revert menu command.
dmSaveEnable	Enables File | Save menu command.
dmSDI	Does not support multiple open documents.

See also TDocManager::TDocManager

dnxxxx document message constants

docmanag.h

Used by TDocManager to indicate that a document or view has been created or closed. You can set up response table entries for these messages using the EV_OWLVIEW or EV_OWLDOCUMENT macros.

Constant	Meaning
dnCreate	A new document or view has been created.
dnClose	A document or view has been closed.

See also TDocManager::TDocManager

dtxxxx document template constants

locale.h

Used by *TDocument* and *TDocTemplate* to create templates. Several constants are equivalent to the OFN_xxxx constants defined by Windows in commdlg.h.

Constant	Windows equivalent	Meaning
dtAutoDelete		Deletes the document when the last view is deleted.
dtAutoOpen		Opens a document upon creation.
dtCreatePrompt	(OFN_CREATEPROMPT)	Prompts the user before creating a document that does not currently exist.
dtDynRegInfo		Used to register a container or server for OLE 2 support. Indicates that the registration information table is dynamic not static.
dtFileMustExist	(OFN_FILEMUSTEXIST)	Lets the user enter only existing file names in the File Name entry field. If an invalid file name is entered, causes a warning message to be displayed.
dtHidden		Hides the template from the user's selection.
dtHideReadOnly	(OFN_HIDEREADONLY)	Hides the read-only check box.
dtNewDoc		Creates a new document with no path specified.
dtNoAutoView		Does not automatically create the default view type.
dtNoReadOnly	(OFN_NOREADONLYRETURN)	Returns the specified file as writeable.
dtNoTestCreate	(OFN_NOTESTFILECREATE)	Does not perform document-creation tests. The file is created after the dialog box is closed. If the application sets this flag, there is no check against write protection, a full disk, an open drive door, or network protection. For certain network environments, this flag should be set.

Constant	Windows equivalent	Meaning
dtOverwritePrompt	(OFN_OVERWRITEPROMPT)	When the Save As dialog box is displayed, asks the user if it's OK to overwrite the file.
dtPathMustExist	(OFN_PATHMUSTEXIST)	Allows only valid document paths to be entered. If an invalid path name is entered, causes a warning message to be displayed.
dtProhibited	(OFN_ALLOWMULTISELECT)	Doesn't support these specified
	(OFN_ENABLEHOOK)	Windows options
	(OFN_ENABLETEMPLATE)	
	(OFN_ENABLETEMPLATEHANDLE)	
dtReadOnly	(OFN_READONLY)	Checks the read-only check box when the dialog box is created.
dtRegisterExt		Used to register a container or server for OLE 2 support. Registers an extension with this application.
dtSelected		Indicates the last selected template.
dtSingleUse		Indicates that the document is to be registered as a singl use document.
dtSingleView		Provides only a single view for each document.
dtUpdateDir		Updates the directory with the directory specified in the dialog box.

See also TDocTemplate::GetFlags, TLocaleString, TDocument class

END_RESPONSE_TABLE macro

eventhan.h

END_RESPONSE_TABLE;

Indicates the end of a response table. For each class that contains a response table, add this macro to the class definition.

See also DEFINE_RESPONSE_TABLE macro

EN_xxxx edit message constants

windows.h

Indicate the state of an edit control in various situations: after a user has changed text, when the edit control receives the input focus, and so on.

Table 2.10 Edit message constants

Constant	Meaning
EN_CHANGE	Message sent when the display is updated after changes have been made to the edit control
EN_ERRSPACE	Message sent when the edit control is out of memory
EN_HSCROLL	Message sent when the user clicks the horizontal scroll bar
EN_KILLFOCUS	Message sent when the edit control is losing the input focus

Table 2.10 Edit message constants (continued)

Constant	Meaning
EN_MAXTEXT	Message sent when the text insertion is truncated
EN_SETFOCUS	Message sent when the edit control receives input focus
EN_UPDATE	Message sent when the edit control is going to display revised text
EN_VSCHOLL	Message sent when the user clicks the vertical scroll bar

EV_xxxx macros

windowev.h

Create response table entries that match events to member functions.

Macro	Meaning
EV_CHILD_NOTIFY(id,notifyCode,method)	Handles child ID notifications (for example, button, edit control, list box, combo box, and scroll bar notification messages) at the child's parent window. Passes no arguments.
EV_CHILD_NOTIFY_ALL_CODES	Passes all notifications to the response function and passes the notification code in as an argument.
EV_CHILD_NOTIFY_AND_CODE(id, notifyCode, method)	Handles child ID notifications at the child's parent window and passes the notification code as an argument.
EV_COMMAND(id, method)	Handler for menu selections, accelerator keys, and push buttons.
EV_COMMAND_AND_ID(id, method)	Handler for multiple commands using a single response function. Passes the menu ID in as an argument.
EV_COMMAND_ENABLE(id, method)	Enables and disables commands such as buttons and menu items.
EV_MESSAGE(message, method)	General purpose macro for Windows WM_xxxx messages.
EV_NOTIFY_AT_CHILD(notifyCode, method)	Handles all child ID notifications at the child window.
EV_OWLDOCUMENT(id, method)	Handles new document notifications.
EV_OWLNOTIFY(id, method)	Generic document handler.
EV_OWLVIEW(id, method)	Handles view notifications.
EV_REGISTERED(str, method)	Handles registered MSG messages.
EV_VIEWNOTIFY	Sends a notification message from the document to the views.

Factory template classes

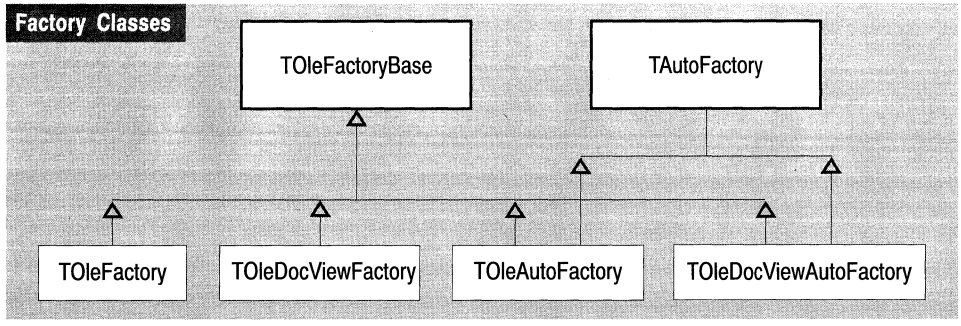
olefacto.h

The factory template classes create callback code for both automated and non automated Doc/View and non-Doc/view ObjectWindows applications. Use these factory template classes to make objects for embedding and linking. (That is, when an application object needs to be embedded within another container, the callback function is responsible for creating the embedded object.) Depending on the template arguments

passed to the factory class, you obtain different kinds of factories designed to create the object you need. All the templetized classes, however, assume that the application is using a global *AppDictionary* (the application's dictionary), and a global *Registrar* (the *TOcRegistrar* pointer that manages registering the application in the database).

ObjectWindows includes several factory template classes, divided into two main categories: those designed for Doc/View applications and those designed for non-Doc/View applications. Although all these classes contain the same functions, they are designed to create different types of objects.

The hierarchy chart illustrates the inheritance relationship among these classes.



Each class takes three arguments: the application class, the automation class, and the Doc/View class. The arguments indicate whether or not the application is a Doc/View application and whether or not the application is automated. The factory classes and their definitions include the following four classes.

Factory class for Doc/View, non-automated, OLE components

```
template <class T> class TOleDocViewFactory
: public TOleFactoryBase<T, TOleFactoryDocView<T,
  TOleFactoryNoAuto<T>>>{};
```

Factory class for Doc/View, automated OLE components

```
template <class T> class TOleDocViewAutoFactory
: public TOleFactoryBase<T, TOleFactoryDocView<T,
  TOleFactoryAuto<T>>>{};
```

Factory class for non-Doc/View, non-automated, OLE components

```
template <class T> class TOleFactory
: public TOleFactoryBase<T, TOleFactoryNoDocView<T,
  TOleFactoryNoAuto<T>>>{};
```

Factory class for non-Doc/View, automated OLE components

```
template <class T> class TOleAutoFactory
```

```
  : public TOleFactoryBase<T, TOleFactoryNoDocView<T,
    TOleFactoryAuto<T>>>{};
```

For either a Doc/View or a non-Doc/View application, you need to register your application in your *OwlMain* function. The argument to *TOcRegistrar* (in this case, *TOleFactory<TDrawApp>*) constructs an object and converts it to a *TComponentFactory* type, using the operator *TComponentFactory* to create a function pointer. In reality, the object is never created because all the factory class's functions are static.

Pass your application object derived from *TApplication* as the parameter to *TOleFactory*, as the following code from STEP15.CPP illustrates:

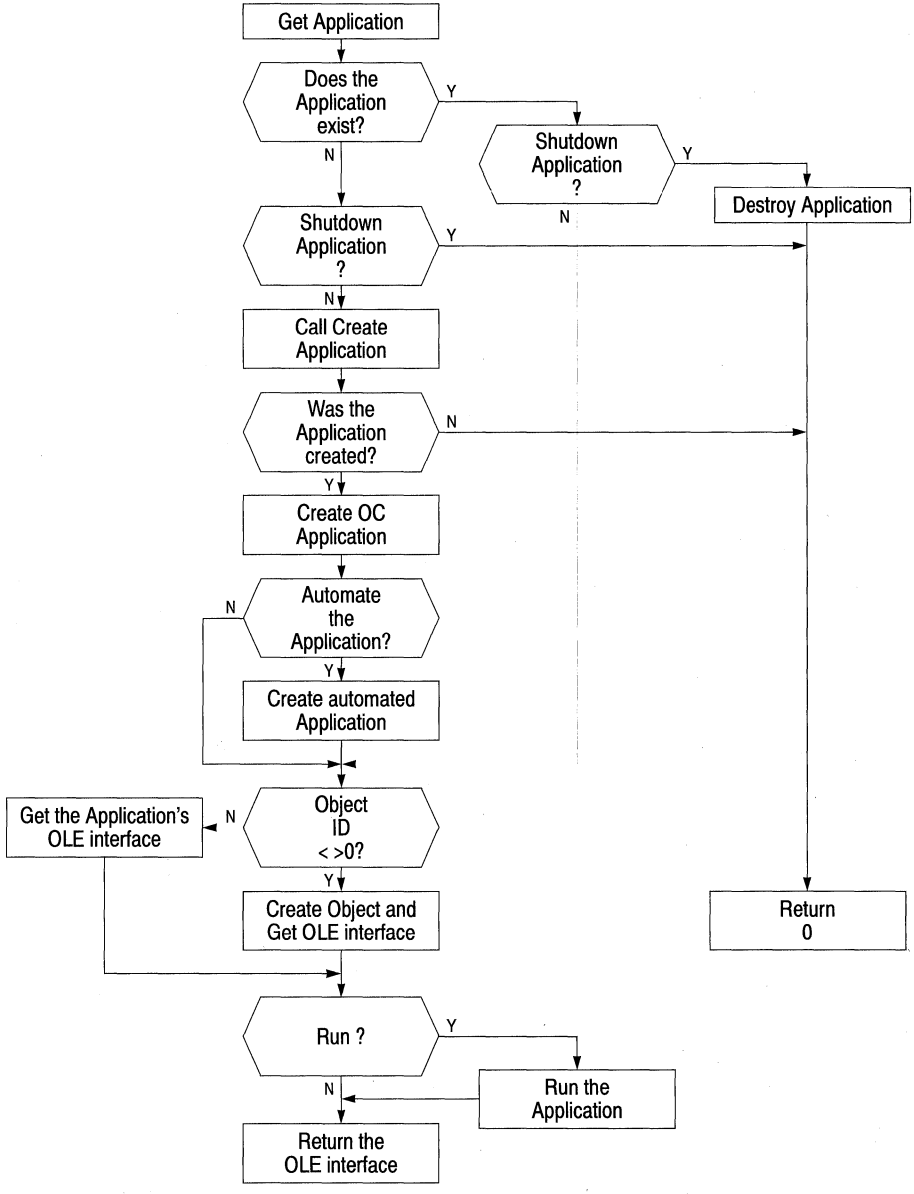
```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
  try {
    Registrar = new TOcRegistrar(AppReg, TOleFactory<TDrawApp>(),
                                 TApplication::GetCmdLine());
    if (Registrar->IsOptionSet(TOcCmdLine::AnyRegOption))
      return 0;
//If this is a normal exe server, run the application now; otherwise,
// wait until the factory is called.
    return Registrar->Run();
}
```

In general, these are the steps each factory class follows in the default callback code:

1. The factory gets the application. This is the application object derived from *TApplication*. For a DLL server, there can be several instances of the object. Using *TAppDictionary::GetApplication*, the factory verifies whether or not there is an entry in the application dictionary for an application object for the current process.
2. If the application does not exist, the factory creates the application object and tests to see if the application was created successfully before creating its corresponding *TOcApp* object. If the shutdown option flag is set, it then exits. (If the application has already been destroyed, the shutdown flag is set.)
3. If the factory is passed a shutdown option flag (one of the *TOcAppMode* enum values), it then shuts down the application and calls the factory's *DestroyApp* function to destroy the application.
4. If the application is automated, the factory creates a corresponding automation object.
5. If the object ID is not zero, the factory creates the object and gets the OLE interface. Otherwise, it gets the application's OLE interface. At this point, the Doc/View and non-Doc/View processes differ because they need to create different types of objects. If the application is automated, the factory creates an automated helper object.
6. The factory checks to see if the option flag *amRun* (one of the *TOcAppMode* enum values) is set, and, if so, it runs the application. This occurs if the application either was built as an .EXE or is a DLL running as an .EXE. If the *amRun* flag is not set and the application is an in-proc DLL server and should not be running, the factory just starts the application.

7 The factory returns either the OLE interface for the object or 0 if no interface was requested or if an error occurred.

The following diagram illustrates this process.



The factory can be called back to walk through this process several times:

1 On the first callback, the factory creates the application, and if the *amRun* flag is set, it enters a message loop.

2 On the second callback, OLE calls the factory to automate or embed or link an object. In the case of an embedded and/or linked object, this pass can occur one or more times. (In the case of an automation object, however, this second pass occurs only once because any subsequent requests pass through the automated application itself.)

3 On the final callback, the factory shuts down the application.

See also
TAutoFactory class, TAutoFactory::DestroyApp, TComponentFactory typedef, TOcAppMode enum, TOcRegistrar class, TOleFactoryBase class, TOleFactoryBase::DestroyApp, TOleFactoryBase::TComponentFactory

GetApplicationObject function

appdict.h

```
TApplication* GetApplicationObject(unsigned pid = 0);
```

This global function is included mainly for backward compatibility with previous ObjectWindows applications. To find the application object associated with a process ID, *GetApplicationObject* calls *TAppDictionary::GetApplication* on an application dictionary. Used by EXEs or DLLs statically linking ObjectWindows, *GetApplicationObject* can return 0.

See also GetWindowPtr function, TAppDictionary::GetApplication

GetWindowPtr function

window.h

```
TWindow* GetWindowPtr(HWND hWnd);
```

This global function is included mainly for backward compatibility with previous ObjectWindows applications. First, *GetWindowPtr* calls the global function *GetApplicationObject* to find the application. Then, calls *TApplication*'s *GetWindowPtr* to get the *TWindow* pointer associated with the window.

See also GetApplicationObject function, TApplication::GetWindowPtr

ID_xxxx file constants

inputdia.rh

Resource and control IDs for the input dialog box.

Constant	Meaning
IDD_INPUTDIALOG	Resource ID number for the input dialog box
ID_INPUT	Control ID for the user input
ID_PROMPT	Control ID for the static text

See also TInputDialog::SetUpWindow

ID_xxxx printer constants

printer.rh

Resource and control IDs for the printer abort dialog box.

Constant	Meaning
IDD_ABORTDIALOG	Resource ID number for the abort dialog box.
ID_TITLE	Control ID for the selected printer driver.
ID_DEVICE	Control ID for the selected printer.
ID_PAGE	ID number for the page number text control.
ID_PORT	Control ID for the selected printer port.

IDA_xxxx accelerator ID constants

editfile.rh

Resource ID for accelerator keys.

Constant	Meaning
IDA_EDITFILE	Resource ID for accelerator keys.

IDA_xxxx OLE accelerator ID constants

oleview.rh

Resource ID for accelerator keys.

Constant	Meaning
IDA_OLEVIEW	Resource ID for accelerator keys for OLE enabled applications.

IDM_xxxx menu ID constants

editfile.rh

Resource ID for menu selections.

Constant	Meaning
IDM_EDITFILE	Resource ID for menu selections.

IDM_xxxx OLE menu ID constants

oleview.rh

Menu IDs for OLE-enabled applications.

Constant	Meaning
IDM_OLEPOPUP	OLE enabled application pop-up menu
IDM_OLEVIEW	OLE enable application view menu selection

IDS_xxxx edit view ID constants

oleview.rh

String constants used to respond to edit view commands.

Constant	Meaning
IDS_EDITOBJECT	Edit the object
IDS_EDITCONVERT	Convert the object
IDS_EXITSERVER	Exit the server application

IDS_Mode constants

statusba.rh

Resource and command IDs to use with TStatusBar.

Constant	Meaning
IDS_MODES	String resource to define mode On indicators
IDS_MODESOFF	String resource to define mode Off indicators

IDS_xxxx document string ID constants

docview.rh

String IDs that define resource IDs used to determine the status of the document.

Constant	Displays these messages
IDS_DOCCHANGED	If the document has been changed, displays the message, "Do you want to save the changes?"
IDS_DOCLIST	Document is a document type.
IDS_DOCMANAGERFILE	This is a document manager file.
IDS_DUPLICATEDOC	This is a duplicate document.
IDS_NODOCMANAGER	There is no document manager.
IDS_NOMEMORYFORVIEW	Not enough memory to view the document
IDS_NOTCHANGED	The document has not been changed.
IDS_READERROR	Error while reading the file
IDS_UNTITLED	The file is untitled.
IDS_UNABLECLOSE	Document manager is unable to close the document.
IDS_UNABLEOPEN	Document manager is unable to open the document.
IDS_UNTITLED	Document is untitled.
IDS_VIEWLIST	Document is a view type.

IDS_xxxx edit file ID constants

editfile.rh

String constants used by edit and file classes to display information about files.

Constant	Meaning
IDS_FILECHANGED	The text in the file has changed. Do you want to save the changes?
IDS_FILEFILTER	Use this filter to display text files.
IDS_UNABLEREAD	Unable to read the file from the disk.
IDS_UNABLEWRITE	Unable to write the file to the disk.
IDS_UNTITLEDFILE	The default window title unless the file is being edited.

IDS_xxxx exception message constants

except.rh

General and application exception message constants. The following list groups the constants according to message type:

Constant	Meaning
IDS_INVALIDMAINWINDOW	Invalid MainWindow
IDS_INVALIDMODULE	Invalid module specified for window
IDS_NOAPP	No application object
IDS_OKTORESUME	Resume in spite of error
IDS_OWLEXCEPTION	Unknown exception
IDS_OUTOFMEMORY	Out of memory
IDS_UNHANDLEDXMSG	Unhandled xmsg error
IDS_UNKNOWNERROR	Unknown error
IDS_UNKNOWNEXCEPTION	Unknown exception error
Owl 1 compatibility messages:	
IDS_INVALIDCHILDWINDOW	Invalid child window
IDS_INVALIDCLIENTWINDOW	Invalid client window
IDS_INVALIDWINDOW	Invalid window
TXWindow messages:	
IDS_CHILDCREATEFAIL	Child create fail for window
IDS_CHILDREGISTERFAIL	Child class registration fails for window
IDS_CLASSREGISTERFAIL	Class registration fails for window
IDS_LAYOUTCOMPLETE	Layout window failure
IDS_LAYOUTBADRELWIN	Layout window failure
IDS_MENUFAILURE	Menu creation failure
IDS_PRINTERERROR	Printer error
IDS_VALIDATORSYNTAX	Validator syntax error
IDS_WINDOWCREATEFAIL	Create fail for window
IDS_WINDOWEXECUTEFAIL	Execute fail for window

Constant	Meaning
GDI messages:	
IDS_GDIALLOCFAIL	GDI allocate failure
IDS_GDICREATEFAIL	GDI creation failure
IDS_GDIDELETEFAIL	GDI object delete failure
IDS_GDIDESTROYFAIL	GDI object destroy failure
IDS_GDIFAILURE	GDI failure
IDS_GDIFILEREADFAIL	GDI file read failure
IDS_INVALIDDIBHANDLE	Invalid DIB handle
IDS_GDIRESLOADFAIL	GDI resource load failure

IDS_xxxx listview ID constants

listview.rh

Define operations performed on views. These include clearing the document, inserting a new line, copying text to the Clipboard, and so on.

Constant	Meaning
IDS_LISTVIEW	Resource ID for listview constants.

IDS_xxxx printer string ID constants

printer.rh

Constants used by printer classes to determine the printer status.

Constant	String displayed
IDS_PRNCANCEL	Printing is canceled.
IDS_PRNERRORCAPTION	Printer error occurred.
IDS_PRNERRORTEMPLATE	Document was not printed.
IDS_PRNGENERROR	Error encountered during printing.
IDS_PRNMGRABORT	Printing aborted in Print Manager.
IDS_PRNON	Printer is on.
IDS_PRNOUTOFDISK	Out of disk space.
IDS_PRNOUTOFMEMORY	Out of memory.

IDS_xxxx validator ID constants

validate.rh

Defines several constants used by validator classes to determine the validator status.

Constant	Meaning
IDS_VALPXPCONFORM	Item doesn't conform to correct picture format.
IDS_VALINVALIDCHAR	Character isn't one of the valid entries.

Constant	Meaning
IDS_VALNOTINRANGE	Entry isn't within the specified range.
IDS_VALNOTINLIS	String isn't found in the list of valid entries.

IDW_MDICLIENT constant

framewin.h

IDW_MDICLIENT
Child ID constant used to identify MDI client windows.

IDW_MDIFIRSTCHILD constant

framewin.h

IDW_FIRSTMDICHILD
Child ID constant used to identify the first MDI client window.

LangXxxx ID constants

locale.h

A language ID is 16-bit number representing a language. In ObjectComponents, a language ID is a value of type *TLangId*.

A language ID is composed of two parts. The ten low-order bits identify a language and the six high-order bits identify a dialect or sub-language. For example, a language ID can specify simply French, or make use of the upper bits to designate Belgian French, Swiss French, or Canadian French.

ObjectComponents defines constants to represent three common IDs:

Constant	Meaning
LangSysDefault	The default language set for the system.
LangUserDefault	The default language set for a particular user (which can differ from the system setting on multi-user systems.)
LangNeutral	A neutral setting signifying no particular locale. An application that receives this value uses its own default setting.

See also TLangId typedef, TLocaleString

LBN_xxxx list box message constant

windows.h

Indicate changes in the status of the list box.

Constant	Meaning
LBN_DBLCLK	Message sent when the user double-clicks a string in a list box
LBN_ERRSPACE	Message sent when the list box is out of memory
LBN_KILLFOCUS	Message sent when the list box is losing the input focus

Constant	Meaning
LBN_SELCANCEL	Message sent when the user cancels the selection in a list box
LBN_SELCHANGE	Message sent when the user changes the selection in a list box
LBN_SETFOCUS	Message sent when the list box receives the input focus.

ImParent constant — layoutco.h

```
#define ImParent 0
```

LmParent is used to construct layout metrics (for example, edge and size constraints).

See also TLayoutConstraint struct

LongMulDiv function — scroller.h

```
long LongMulDiv(long mul1, long mul2, long div1);
```

TScroller uses this function to convert horizontal range values (*XRange*) from the scroll bar to horizontal scroll values (*XScrollValue*) and vice versa, or to convert vertical range values (*YRange*) from the scroll bar to vertical scroll values (*YScrollValue*) and vice versa.

See also TScroller

MAX_RSRC_ERROR_STRING constant — except.h

```
const int MAX_RSRC_ERROR_STRING = 255;
```

Maximum number of characters possible for an error message.

MB_Xxxx message constants — windows.h

Contain information about the style and behavior of a message dialog box. You can use these constants to determine the kinds of information displayed in a message dialog box.

Constant	Meaning
MB_OK	The message dialog box contains an OK push button.
MB_OKCANCEL	The message dialog box contains Cancel and OK push buttons.
MB_ABORTRETRYIGNORE	The message dialog box contains Abort, Retry, and Ignore push buttons.
MB_YESNOCANCEL	The message dialog box contains Yes, No and Cancel push buttons.
MB_YESNO	The message dialog box contains Yes and No push buttons.
MB_RETRYCANCEL	The message dialog box contains Retry and Cancel push buttons.
MB_ICONHAND	The message dialog box contains a stop sign icon.
MB_ICONQUESTION	The message dialog box contains a question mark.
MB_ICONEXCLAMATION	The message dialog box contains an exclamation mark.

Constant	Meaning
MB_ICONASTERISK	The message dialog box contains an icon consisting of a lowercase letter i.
MB_ICONINFORMATION	The message dialog box contains an icon consisting of a lowercase letter i.
MB_ICONSTOP	The message dialog box contains a stop sign icon.
MB_ICONHAND	The message dialog box contains a stop-sign icon.
MB_DEFBUTTON1	The first button is the default button in the message dialog box. This is always the case unless MB_DEFBUTTON2 or MB_DEFBUTTON3 is specified.
MB_DEFBUTTON2	The second button is the default button in the message dialog box.
MB_DEFBUTTON3	The third button is the default button in the message dialog box.
MB_APPLMODAL	Before continuing to work in this window, the user must answer the message dialog box. However, the user can work in other windows. MB_APPLMODAL is the default unless MB_SYSTEMMODAL, MB_TASKMODAL, or MB_NOFOCUS is specified.
MB_SYSTEMMODAL	Before continuing to work in this window, the user must answer the message dialog box. Unless the application indicates MB_ICONHAND, the message box does not become modal until after it is created. The owning window and other windows can continue to receive messages. You can use MB_SYSTEMMODAL to notify the user of serious errors, such as lack of sufficient memory to run an application, that must be taken care of immediately.
MB_TASKMODAL	Before continuing to work in this window, the user must answer the message dialog box. However, the user can work in other windows. Unlike MB_APPLMODAL, all top-level windows in the current task are disabled. Use this constant when the calling application does not have a window handle available, and you want to prevent input to other windows in the current application without actually preventing the other applications from executing.

See also TDocument::PostError, TDocManager::PostDocError, TWindow::MessageBox

NBits function

color.h

```
uint16 NBits(long colors);
```

Returns the bit count corresponding to the given color count.

See also NColors, TColor class

NColors function

color.h

```
int NColors(uint16 bitCount);
```

Returns the color count corresponding to the given bit count, or –1 if the bit count is not supported by Windows. Bit counts currently supported are 1, 4, 8, and 24.

See also NBits, TColor class

ofxxxx document open enum

docview.h

Defines the document and open sharing modes used for constructing streams and storing data. Any constants that have the same functionality as those used by OLE 2.0 docfiles are indicated in the following table; for example, STGM_TRANSACTED, STGM_CONVERT, STGM_PRIORITY, and STGM_DELETEONRELEASE.

Although files are typically used for data storage, databases or spreadsheets can also be used. I/O streams rather than DOS use these bit values. Documents open the object used for storage in one of the following modes:

Constant	Meaning
ofParent	A storage object is opened using the parent's mode.
ofRead	A storage object is opened for reading.
ofWrite	A storage object is opened for writing.
ofReadWrite	A storage object is opened for reading and writing.
ofAtEnd	Seek to end-of-file when opened originally.
ofAppend	Data is appended to the end of the storage object.
ofTruncate	An already existing file is truncated.
ofNoCreate	Open fails if file doesn't exist.
ofNoReplace	Open fails if file already exists.
ofBinary	Data is stored in a binary, not text, format. Carriage returns are not stripped.
ofIosMask	All of the above bits are used by the ios class.
ofTransacted	Changes to the storage object are preserved until the data is either committed to permanent storage or discarded. (STGM_TRANSACTED)
ofPreserve	Backs up previous storage data using before creating a new storage object with the same name. (STGM_CONVERT)
ofPriority	Supports temporary, efficient reading before opening the storage. (STGM_PRIORITY)
ofTemporary	The storage or stream is automatically destroyed when it is destructed. (STGM_DELETEONRELEASE)

See also TStream, TInStream, TOutStream

pfxxxx property attribute constants

docview.h

Define document and view property attributes. Documents, views, and applications use these attributes to determine how to process a document or view.

Constant	Meaning
pfGetText	Property is accessible in a text format.
pfGetBinary	Property is accessible as a native nontext format.
pfConstant	Property can't be changed for the object instance.
pfSettable	Property can be set as a native format.
pfUnknown	Property is defined but unavailable for the object.

Constant	Meaning
pfHidden	Property should be hidden from the user during normal browsing.
pfUserDef	Property has been user-defined at run time.

See also TDocument, TView

_BUILDOWLDLL macro

owldefs.h

_BUILDOWLDLL
Used internally to control values for the _OWLCLASS, _OWLDATA, and _OWLFUNC macros. This macro is defined when the user's module is built as a DLL. It must be defined and included in ObjectWindows makefiles to build the ObjectWindows DLL.

See also _OWLDLL macro

_OWLCLASS macro

owldefs.h

_OWLCLASS
Used internally by ObjectWindows to modify an entire class for use in a DLL. It is the ObjectWindows version of _RTLCLASS adjusted to export and import WIN32 DLLs.

For static WIN16 and WIN32 , the default models are used. When ObjectWindows is being built, this macro evaluates to **_export** for WIN16 and WIN32 DLLs. For WIN32 DLLs, this macro evaluates to **_import** and performs necessary operations for WIN32 DLLs. For WIN16 DLL use, this macro evaluates to **_import**, which is essentially the same as **_huge**.

_OWLDATA macro

owldefs.h

_OWLDATA
The ObjectWindows version of _RTLDATA adjusted to export and import WIN32 DLLs for ObjectWindows. _OWLDATA modifies a specific data declaration.

For static WIN16 and WIN32, the default models are used. When ObjectWindows is being built, this macro evaluates to **_export** for WIN16 and WIN32 DLLs. For WIN32 DLLs, this macro evaluates to **_import** and performs necessary operations for WIN32 DLLs. For WIN16 DLLs, this macro evaluates to nothing.

_OWLDLL macro

owldefs.h

_OWLDLL
_OWLDLL, which is automatically defined if _RTLDLL is turned on, controls values for the _OWLCLASS, _OWLDATA, and _OWLFUNC macros. It is also automatically defined if the user _OWLDLL module is used as a DLL from another user module. It

must be defined if you are writing ObjectWindows applications or DLLs that use DLLs. This macro can also be turned on by a makefile.

_OWLFAR macro — owldefs.h

_OWLFAR
The macro _OWLFAR is the ObjectWindows version of _RTLFAR adapted to promote far data pointers in DLLs for ObjectWindows.

_OWLFARVTABLE macro — owldefs.h

_OWLFARVTABLE
Moves the ObjectWindows virtual function tables (vtables) out of the DGROUP of the data segment and stores them in the code segment. Use this macro in conjunction with the _OWLCLASS macro to add the **_huge** option when static models are compiled.

_OWLFASTTHIS macro — owldefs.h

_OWLFASTTHIS
The macro _OWLFASTTHIS causes **_fastthis** to be added to the _OWLCLASS macro so that all ObjectWindows classes use the fastthis calling convention for passing the **this** parameter in the registers. This macro, which has the same effect as using the **–po** compiler option, applies to 16-bit code only.

_OWLFUNC macro — owldefs.h

_OWLFUNC
The ObjectWindows function version of _RTLFUNC adapted to export and import functions for building WIN32 DLLs for ObjectWindows. _OWLFUNC modifies a specific member or global function for use in a DLL.

For static WIN16 and WIN32 DLLs, the default models are used. When ObjectWindows is being built, this macro evaluates to **_export** for WIN16 and WIN32 DLLs. For WIN32 DLLs, this macro evaluates to **_import** and performs necessary operations for WIN32 DLLs. For WIN16 DLL use, this macro evaluates to nothing.

OWLGetVersion function — owldefs.h

unsigned short far _OWLFUNC OWLGetVersion();
Returns the version number of the ObjectWindows library. The version number is represented as an unsigned short.

SB_Xxxx scroll bar constants

windows.h

The following constants define scroll bar modes:

Constant	Meaning
SB_BOTH	Displays or hides the horizontal and vertical scroll bars for a window.
SB_CTL	Displays or hides a scroll bar's control button.
SB_HORIZ	Displays or hides the horizontal scroll bars for a window.
SB_VERT	Displays or hides the vertical scroll bar for a window.

shxxxx document sharing enum

docview.h

The following file-sharing modes are available when opening document streams.

Constant	Meaning
shCompat	Used for noncompliant applications, but should be avoided if possible.
shNone	DENY_ALL functionality.
shRead	DENY_WRITE functionality.
shWrite	DENY_READ functionality.
shReadWrite	DENY_NONE functionality
shDefault	Use stream implementation default value.
shMask	Mask for file-sharing bits.

TActionFunc typedef

window.h

```
typedef void(*TActionFunc)(TWindow* win, void* param);
```

Passes a function pointer to *TWindow::ForEach*.

See also TWindow::ForEach

TActionMemFunc typedef

window.h

```
typedef void(TWindow::*TActionMemFunc)(TWindow* win, void* param);
```

Passes a member function pointer to *TWindow::ForEach*.

See also TWindow::ForEach

TAnyPMF typedef

dispatch.h

```
typedef void(GENERIC::*TAnyPMF)();
```

A generic pointer to a member function.

TAnyDispatcher typedef

dispatch.h

typedef LRESULT(*TAnyDispatcher)(GENERIC&, TAnyPMF, WPARAM, LPARAM);

A message dispatcher type. All message dispatcher functions conform to this type and take four parameters:

- A reference to an object
- A pointer to the member function in which the signature varies according to the cracking that the function performs
- WPARAM
- LPARAM

TAppDictionary class

appdict.h

A *TAppDictionary* is a dictionary of associations between a process ID and an application. A process ID identifies a process, that is, a program (including all of its affiliated code, data, and system resources) that is loaded into memory and ready to execute. A *TAppDictionary* supports global application lookups using the global *GetApplicationObject* function or *TAppDictionary*'s *GetApplication* function. If you do not define an application dictionary, ObjectWindows provides a default, global application dictionary that is exported. In fact, for EXEs, this global application dictionary is automatically used.

TAppDictionary includes a *TEntry* struct, which stores the process ID and the corresponding application associated with the ID. The public member functions add, find, and remove the entries in the appplication dictionary.

If you are statically linking ObjectWindows, you do not have to explicitly create an application dictionary because the default global ObjectWindows application dictionary is used. However, when writing a DLL component that is using ObjectWindows in a DLL, you do need to create your own dictionary. To make it easier to define an application dictionary, ObjectWindows includes a macro DEFINE_APP_DICTIONARY, which automatically creates or references the correct dictionary for your application.

Although this class is transparent to most users building EXEs, component DLLs need to create an instance of a *TApplication* class for each task that they service. This kind of application differs from an EXE application in that it never runs a message loop. (All the other application services are available, however.)

Although a component may consist of several DLLs, each with its own *TModule*, the component has only one *TApplication* for each task. A *TAppDictionary*, which is used for all servers (including DLL servers) and components, lets users produce a complete, self-contained application or component. By using a *TAppDictionary*, these components can share application objects.

When 16-bit ObjectWindows is statically linked with an EXE or under Win32, with per-instance data, the *TAppDictionary* class is implemented as a wrapper to a single

application pointer. In this case, there is only one *TApplication* that the component ever sees.

To build a component DLL using the ObjectWindows DLL, a new *TAppDictionary* object must be constructed for that DLL. These are the steps an application must follow in order to associate the component DLL with the *TAppDictionary*, the application, and the window class hierarchy:

1 Use the DEFINE_APP_DICTIONARY macro to construct an instance of *TAppDictionary*. Typically, this will be a static global in one of the application's modules (referred to as "AppDictionary"). The DEFINE_DICTIONARY macro allows the same code to be used for EXEs and DLLs.

```
TAppDictionary AppDictionary;
```

2 Construct a generic *TModule* and assign it to the global *::Module*. This is the default provided in the ObjectWindows' *LibMain* function.

```
LibMain(...)
   ::Module  = new TModule(0, hInstance);
```

3 When each *TApplication* instance is constructed, pass a pointer to the *TAppDictionary* as the last argument. This ensures that the application will insert itself into this dictionary. In addition, for 16 bit DLLs, the *gModule* argument needs to be supplied with a placeholder value because the *Module* construction has already been completed at this point, as a result of the process performed in step 2.

```
TApplication* app = new TMyApp(..., app, AppDictionary);
```

4 If the Doc/View model is used, supply the application pointer when constructing the *TDocManager* object.

```
SetDocManager(new TDocManager(mode, this));
```

5 When a non-parented window (for example, the main window) is being constructed, pass the application as the module.

```
SetMainWindow(new TFrameWindow(0, "", false, this));
```

See also

TApplication::GetWindowPtr, TWindow::GetWindowPtr, DEFINE_APP_DICTIONARY macro

Type definitions

TEntry

```
struct TEntry {
 unsigned Pid;
 TApplication* App;
};
```

An application dictionary entry that associates a process ID (*Pid*) with an application (*App*). The dictionary is indexed by (*Pid*) and can have only 1 entry per process ID.

See also TAppDictionary::TEntry struct, TAppDictionary::Iterate

TEntryIterator

typedef void(*TEntryIterator)(TEntry&);

A dictionary iterator function pointer type that receives a reference to an entry. You can supply a function of this type to the *Iterate* function to iterate through the entries in the dictionary.

See also TAppDictionary::TEntryIterator typedef

Public constructor and destructor

Constructor

TAppDictionary();

Constructs a *TAppDictionary* object.

Destructor

~TAppDictionary();

Destroys the *TAppDictionary* object and calls *DeleteCondemned* to clean up the condemned applications.

Public member functions

Add

void Add(TApplication* app, unsigned pid = 0);

Adds an application object (*app*) and corresponding process ID to this dictionary. The default ID is the current process's ID.

See also TAppDictionary::Remove

Condemn

void Condemn(TApplication* app);

Marks an application in this dictionary as condemned by zeroing its process ID so that the application can be deleted later when *DeleteCondemned* is called.

See also TAppDictionary::DeleteCondemned

DeleteCondemned

bool DeleteCondemned();

Deletes all condemned applications and their associated process IDs from the dictionary. If no applications remain in the dictionary, *DeleteCondemned* returns **true**.

See also TAppDictionary::Condemn

GetApplication

TApplication* GetApplication(unsigned pid = 0);

Looks up and returns the application associated with a given process ID. The default ID is the ID of the current process. If no application is associated with the process ID, *GetApplication* returns 0.

Iterate

void Iterate(TEntryIterator iter);

Iterates through a list of entries in the application dictionary, calling the *iter* callback function for each entry.

See also TAppDictionary::TEntryIterator

Remove

Form 1 `void Remove(TApplication* app);`

Searches for the dictionary entry using the specified application (*app*). Then, removes a given application and process ID entry from this dictionary, but does not delete the application.

Form 2 `void Remove(unsigned pid);`

Searches for the dictionary entry using the specified process ID (*pid*). Then, removes a given application and its associated process ID entry from this dictionary, but does not delete the application.

See also TAppDictionary::Add

TApplication class

applicat.h

Derived from *TModule*, *TApplication* acts as an object-oriented stand-in for an application module. *TApplication* and *TModule* supply the basic behavior required of an application. *TApplication* member functions create instances of a class, create main windows, and process messages.

Public data members

HAccTable

`HACCEL HAccTable;`

Included to provide backward compatibility, *HAccTable* holds a handle to the current accelerator table being used by the application. New applications should instead use the accelerator table handle *TWindowAttr*::AccelTable for each window object in the application.

See also TWindow::LoadAcceleratorTable, TWindowAttr struct

hPrevInstance

`HINSTANCE hPrevInstance;`

Contains the handle of the previously executing instance of the Windows application. If *hPrevInstance* is 0, there was no previously executing instance when this instance began execution. Under Win32, this value is always 0.

nCmdShow

`int nCmdShow;`

Indicates how the main window is to be displayed (either maximized or as an icon). These correspond to the *WinMain* parameter *nCmdShow*. *nCmdShow* can contain one of the following constants:

Constant	Meaning
SW_SHOWDEFAULT	Shows the default SW_xxxx command.
SW_HIDE	Hides the window.
SW_MINIMIZE	Minimizes the specified window.
SW_SHOW	Activates a window using current size and position.
SW_SHOWMAXIMIZED	Displays a maximized window.
SW_SHOWMINIMIZED	Displays a minimized window.
SW_SHOWNA	Displays window in its current state.
SW_SHOWNOACTIVATE	Displays the window as an icon.
SW_SHOWNORMAL	Displays a window in its original size and position.
SW_SHOWSMOOTH	Shows a window by updating it in a bitmap and then copying the bits to the screen.

Type definitions

xs

```
enum {xsUnknown, xsBadCast, xsBadTypeid, xsMsg, xsAlloc, xsOwl};
```

These bit flags define the types of exceptions that are caught and suspended. *TApplication::SuspendThrow* and *TApplication::QueryThrow* return the values of these bit flags.

The following table shows the xs exception enum constants:

Constant	Meaning
xsUnknown	Unknown exception
xsBadCast	*Bad_cast* exception
xsBadTypeid	*Bad_typeid* exception
xsMsg	Any exception derived from *xmsg*
xsAlloc	*xalloc* exception
xsOwl	*TXOwl* exception

See also TXOwl, TApplication::QueryThrow, TApplication::SuspendThrow

Public constructor and destructor

Constructor

Form 1

```
TApplication(const char far* name = 0, TModule*& gModule = ::Module, TAppDictionary* appDict = 0);
```

This *TApplication* constructor creates a new *TApplication* object named *name*. You can use *gModule* to specify the global module pointer that points to this application. The *appDict* parameter specifies which dictionary this application will insert itself into. To override the default ObjectWindows *TAppDictionary* object, pass a pointer to a user-supplied *appDict* object.

Form 2 TApplication(const char far* name, HINSTANCE hInstance, HINSTANCE hPrevInstance, const char far* cmdLine, int cmdShow, TModule*& gModule = ::Module, TAppDictionary* appDict = 0);

This *TApplication* constructor creates a *TApplication* object with the application name (*name*), the application instance handle (*instance*), the previous application instance handle (*prevInstance*), the command line invoked (*cmdLine*), and the main window show flag (*cmdShow*). The *appDict* parameter specifies which dictionary this application will insert itself into. To override the default ObjectWindows *TAppDictionary* object, pass a pointer to a user-supplied *appDict* object.

If you want to create your own *WinMain*, use this constructor because it provides access to the various arguments provided by *WinMain*. You can use *gModule* to specify the global module pointer that points to this application.

Destructor

~TApplication();

~TApplication destroys the *TApplication* object.

See also TApplication::nCmdShow, TModule, TAppDictionary

Public member functions

BeginModal

int BeginModal(TWindow* window, int flags = MB_APPLMODAL);

BeginModal is called to begin a modal window's modal message loop. After determining which window to disable, saves the current status of the window, disables the window, calls *MessageLoop*, and then reenables the window when the message loop is finished. The flags determine how *BeginModal* treats the window. *flags* can be one of the following values:

Constant	Meaning
MB_APPLMODAL	The window to be disabled (which is usually an ancestor of the modal window) is identified by window. If window is 0, no window is disabled.
MB_SYSTEMMODAL	The window to become system modal is identified by window.
MB_TASKMODAL	All top-level windows are disabled, and window is ignored. *BeginModal* returns –1 if an error occurs.

See also TWindow

BWCCEnabled

bool BWCCEnabled();

Indicates if the Borland Custom Controls library (BWCC) is enabled. Returns **true** if BWCC is enabled and **false** if BWCC is disabled.

CanClose

virtual bool CanClose();

Returns **true** if it's OK for the application to close. By default, *CanClose* calls the *CanClose* member function of its main window and returns **true** if both the main window and the document manager (*TDocManager*) can be closed. If any of the *CanClose* functions return **false**, the application doesn't close.

This member function is seldom redefined; closing behavior is usually redefined in the main window's *CanClose* member function, if needed.

See also TWindow::CanClose, TDocManager

Condemn

```
void Condemn(TWindow* win);
```

Performs window cleanup.

Ctl3dEnabled

```
bool Ctl3dEnabled() const;
```

Returns **true** if the Microsoft 3-D Controls Library DLL is enabled. This DLL gives controls a three-dimensional look and feel.

See also TApplication::EnableCtl3d, TApplication::EnableCtl3dAutosubclass

EnableBWCC

```
void EnableBWCC(bool enable = true, uint Language = 0);
```

Loads and registers BWCC.DLL if you are running 16-bit applications or BWCC32.DLL if you are running 32-bit applications. By default, BWCC is enabled. To disable BWCC, set *enable* to **false**.

See also TDialog

EnableCtl3d

```
void EnableCtl3d(bool enable = true);
```

Enables or disables the use of the CTL3D DLL. If *enable* is **true**, *EnableCtl3d* loads and registers the CTL3D.DLL if it's not already enabled.

See also TApplication::Ctl3dEnabled, TApplication::EnableCtl3dAutosubclass

EnableCtl3dAutosubclass

```
void EnableCtl3dAutosubclass(bool enable);
```

Enables or disables CTL3D's use of autosubclassing if CTL3D is already enabled using *Ctl3dEnabled*. If autosubclassing is enabled, any non-ObjectWindows dialog boxes have a 3-D effect. The common dialog classes and *TDocManager* use this function to turn on autosubclassing before creating a non-ObjectWindows dialog box to make it three-dimensional and to turn off autosubclassing immediately after the dialog box is destroyed.

See also TDialog::EvCtlColor, TApplication::EnableCtl3d, TApplication::Ctl3dEnabled

EndModal

```
void EndModal(int result);
```

EndModal is called to end a modal window's modal message loop. Sets *result* to –1 if an error occurs.

Find

```
bool Find(TEventInfo &, TEqualOperator = 0);
```

Because *TApplication* has no event table itself, it defers event handling to the DocManager. If a DocManager has been installed, *Find* calls *TDocManager* to handle events.

See also TEventHandler::TEventInfo

GetBWCCModule

TModule* GetBWCCModule() const;

Returns a pointer to the enabled BWCC module.

GetCtl3dModule

TModule* GetCtl3dModule() const;

Returns a pointer to the enabled Ctl3d module.

GetDocManager

TDocManager* GetDocManager();

Returns a pointer to the document manager object that invoked the application.

See also TApplication::SetDocManager, TDocManager

GetMainWindow

TFrameWindow* GetMainWindow();

Returns a pointer to the application's main window.

See also TApplication::SetMainWindow, TFrameWindow

GetWindowPtr

TWindow* GetWindowPtr(HWND hWnd) const;

Retrieves a *TWindow* pointer associated with the handle to a window (hWnd). Allows more than one application to share the same HWND.

See also TWindow::GetWindowPtr

GetWinMainParams

void GetWinMainParams();

Initializes a static instance of an application. ObjectWindows *OwlMain* uses this function to support static application instances.

See also TApplication::SetWinMainParams

MessageLoop

virtual int MessageLoop();

Operates the application's message loop, which runs during the lifetime of the application. Queries for messages; if one is received, *MessageLoop* processes it by calling *ProcessAppMsg*. If the query returns without a message, *MessageLoop* calls *IdleAction* to perform some processing during the idle time. *MessageLoop* calls *PumpWaitingMessages* to get and dispatch waiting messages. *MessageLoop* can be broken if *BreakMessageLoop* is set by *EndModal*.

See also TApplication::BreakMessageLoop, TApplication::IdleAction, TApplication::ProcessAppMsg, TApplication::PumpWaitingMessages

PostDispatchAction

void PostDispatchAction();

If *TApplication's* message loop is not used, this function should be called after each message is dispatched

PreProcessMenu

virtual void PreProcessMenu(HMENU hmenu);

Your application can call *PreProcessMenu* to process the main window's menu before it is displayed.

See also TDocmanager::EvPreProcessMenu, TMenu::TMenu

ProcessAppMsg

virtual bool ProcessAppMsg(MSG& msg);

Checks for any special processing that is required for modeless dialog box, accelerator, and MDI accelerator messages. Calls the virtual *TWindow::PreProcessMsg* function of the window receiving the message. If your application does not create modeless dialog boxes, does not respond to accelerators, and is not an MDI application, you can improve performance by overriding this member function to return **false**.

See also TWindow::PreProcessMsg, MSG struct

PumpWaitingMessages

bool PumpWaitingMessages();

Called by *MessageLoop*, *PumpWaitingMessages* processes and dispatches all waiting messages until the queue is empty. It also sets *BreakMessageLoop* when a WM_QUIT message is received.

See also TApplication::MessageLoop, TApplication::BreakMessageLoop

QueryThrow

int QueryThrow();

QueryThrow tests to see if an exception is suspended and returns one or more of the bit flags in the *xs exception status enum*.

See also xs exception status enum

ResumeThrow

void ResumeThrow();

ResumeThrow checks and rethrows suspended exceptions. Call this function any time you reenter ObjectWindows code from exception-unsafe code where an exception could have been thrown.

Run

virtual int Run();

Initializes the instance, calling *InitApplication* for the first executing instance and *InitInstance* for all instances. If the initialization is successful, *Run* calls *MessageLoop* and runs the application. If exceptions are thrown outside the message loop, *Run* catches these exceptions.

If an error occurs in the creation of a window, *Run* throws a *TXWindow* exception. If *Status* is assigned a nonzero value (which ObjectWindows uses to identify an error), a *TXCompatibility* exception is thrown.

See also TApplication::InitApplication, TApplication::InitInstance, TApplication::MessageLoop, TXWindow, TXCompatibility

SetWinMainParams

static void SetWinMainParams(HINSTANCE hInstance, HINSTANCE hPrevInstance, const char far* cmdLine, int cmdShow);

ObjectWindows default *WinMain* function calls *SetMainWinParams* so that *TApplication* can store the parameters for future use. To construct an application instance, *WinMain* calls the *OwlMain* function that's in the user's code. As it's being constructed, the application instance can fill in the parameters using those set earlier by *SetMainWinParams*.

See also TApplication::GetWinMainParams

SuspendThrow

Form 1 `void SuspendThrow(xalloc& x);`
This version of *SuspendThrow* saves xalloc exception information.

Form 2 `void SuspendThrow(xmsg& x);`
This version of *SuspendThrow* saves xmsg exception information.

Form 3 `void SuspendThrow(TXOwl& x);`
This version of *SuspendThrow* saves a copy of a *TXOwl* exception.

Form 4 `void SuspendThrow(int);`
This version of *SuspendThrow* sets the *xs exception status* bit flags to the specified exception, for example *Bad_cast* or *Bad_typeid*.

See also xs exception status enum, TXOwl

Uncondemn

`void Uncondemn (TWindow* win);`
Removes condemned children from the list of condemned windows.

See also TWindow

Protected data members

BreakMessageLoop

`bool BreakMessageLoop;`
Causes the current modal message loop to break and terminate. If the current modal message loop is the main application, and your program sets *BreakMessageLoop*, your main application terminates.

See also TApplication::EndModal, TApplication::MessageLoop, TApplication::PumpWaitingMessages

MessageLoopResult

`int MessageLoopResult;`
MessageLoopResult is set by a call to *EndModal*. It contains the value that is returned by *MessageLoop* and *BeginModal*.

See also TApplication::BeginModal, TApplication::EndModal, TApplication::MessageLoop

Protected member functions

IdleAction

virtual bool IdleAction(long idleCount);

ObjectWindows calls *IdleAction* when no messages are waiting in the queue to be processed. You can override *IdleAction* to do background processing. However, the default action is to give the main window a chance to do idle processing as long as *IdleAction* returns true. *idleCount* specifies the number of times *IdleAction* has been called between messages.

See also TFrameWindow::IdleAction

InitApplication

virtual void InitApplication();

ObjectWindows calls *InitApplication* to initialize the first instance of the application. For subsequent instances, this member function is not called.

The following sample program calls *InitApplication* the first time an instance of the program begins.

```
class TTestApp : public TApplication {
   public:
     TTestApp(): TApplication("Instance Tester")
       {strcpy(WindowTitle, "Additional Instance");}
   protected:
      char WindowTitle[20];
   void InitApplication() {strcpy(WindowTitle, "First Instance");}
   void InitMainWindow() {MainWindow = new TFrameWindow(0, WindowTitle);}
};
static TTestApp App;
```

InitInstance

virtual void InitInstance();

Performs each instance initialization necessary for the application. Unlike *InitApplication*, which is called for the first instance of an application, *InitInstance* is called whether or not there are other executing instances of the application. *InitInstance* calls *InitMainWindow*, and then creates and shows the main window element by *TWindow::Create* and *TWindow::Show*. If the main window can't be created, a *TXInvalidMainWindow* exception is thrown.

If you redefine this member function, be sure to explicitly call *TApplication::InitInstance*.

See also TApplication::InitApplication, TApplication::InitMainWindow, TApplication::Run, TModule::MakeWindow, TWindow::Show

InitMainWindow

virtual void InitMainWindow();

By default, *InitMainWindow* constructs a generic *TFrameWindow* object with the name of the application as its caption. You can redefine *InitMainWindow* to construct a useful main window object of *TFrameWindow* (or a class derived from *TFrameWindow*) and store it in *MainWindow*. The main window must be a top-level window; that is, it must be derived from *TFrameWindow*. A typical use is

```
virtual void TMyApp_InitMainWindow(){
  SetMainWindow(TMyWindow(NULL, Caption));
}
```

InitMainWindow can be overridden to create your own custom window.

SetDocManager

TFrameWindow* SetDocManager(TDocManager* docManager);

Sets a pointer to the document manager object that invoked the application.

See also TApplication::GetDocManager, TDocManager, TFrameWindow

SetMainWindow

TFrameWindow* SetMainWindow(TFrameWindow* window);

Sets up a new main window and sets the window's WM_MAINWINDOW flag.

See also TApplication::GetMainWindow, TFrameWindow

TermInstance

virtual int TermInstance(int status);

Handles the termination of each executing instance of an ObjectWindows application.

TApplication::TXInvalidMainWindow class

applicat.h

A nested class, *TXInvalidMainWindow* describes an exception that results from an invalid Window. This exception is thrown if there is not enough memory to create a window or a dialog object. *InitInstance* throws this exception if it can't initialize an instance of an application object.

Public constructor

Constructor

TXInvalidMainWindow();

Constructs a *TXInvalidMainWindow* object with a default IDS_INVALIDMAINWINDOW message.

Public member functions

Clone

virtual TXOwl* Clone();

Makes a copy of the exception object. *Clone* must be implemented in any class derived from *TXOwl*.

Throw

virtual void Throw();

Throws the exception object. *Throw* must be implemented in any class derived from *TXOwl*.

TAutoFactory<> class

olefacto.h

A template class, *TAutoFactory<>* creates callback code for ObjectWindows classes. The application class is passed as the argument to the template. By itself, *TAutoFactory<>* does not provide linking or embedding support for ObjectWindows automated applications.

Although *TAutoFactory<>* simplifies the process of creating the callback function, you can write your own callback function or provide alternate implementation for any or all of *TAutoFactory<>*'s functions.

See also

TComponentFactory typedef, TOcRegistrar class, TOleFactoryBase class

Public member functions

Create

static IUnknown* Create(IUnknown* outer, uint32 options, uint32 id);

Create is a *TComponentFactory* callback function that creates or destroys the application or creates objects. If an application object does not already exist, *Create* creates a new one. The *outer* argument points to the OLE2 *IUnknown* interface with which this object aggregates itself. If outer is 0, the new object is not aggregated, or it will become the main object.

The *options* argument indicates the application's mode while it is running. The values for *options* are either set from the command line or set by ObjectComponents. They are passed in by the "Registrar" to this callback. The application looks at these flags to know how to operate, and the factory callback looks at them to know what to do. For example, a value of *amExeMode* indicates that the server is running as an .EXE either because it was built as an .EXE or because it is a .DLL that was launched by an .EXE stub and is now running as an executable program. See *TOcAppMode* enum for a description of the possible values for the *options* argument.

If the application already exists, *Create* returns the application's OLE interface and registers the options from *TOcAppMode* enum which contains OLE-related flags used in the application's command line. For example, the *amAutomation flag* tells the server to register itself as a single-user application. (In general, these flags tell the application whether it has been run as a server, whether it needs to register itself, and so on.)

The *id* argument, which is not used for *TAutoFactory*, is always 0.

See also TAutoFactory::DestroyApp, TOcAppMode enum

CreateApp

static T* CreateApp(uint32 options);

CreateApp creates a new automated application. By default, it creates a new application of template type *T* with no arguments. The *options* argument is one of the *TOcAppMode* enum values, for example, *amRun*, *amAutomation*, and so on that indicate the application's mode when running.

See also TAutoFactory::DestroyApp, TOcAppMode enum

DestroyApp

static void DestroyApp(T* app);

Destroys the previously created application referred to in *app*.

operator TComponentFactory

operator TComponentFactory();

Converts the object into a pointer to the factory. ObjectComponents uses this pointer to create the automated object.

See also TAutoFactory::CreateApp

TBandInfo struct

dc.h

An ObjectWindows **struct**, *TBandInfo* is used to pass information to a printer driver that supports banding. *TBandInfo* is declared as follows:

```
struct TBandInfo {
bool HasGraphics;
bool HasText;
TRect GraphicsRect;
};
```

HasGraphics is **true** if graphics are (or are expected to be) on the page or in the band; otherwise, it is **false**.

HasText is **true** if text is (or is expected to be) on the page or in the band; otherwise, it is **false**.

GraphicsRect defines the bounding region for all graphics on the page.

See also TPrintDC::BandInfo, TPrintDC::NextBand

TBitmap class

gdiobjec.h

TBitmap is the GDI bitmap class derived from *TGdiObject*. *TBitMap* can construct a bitmap from many sources. *TBitmap* objects are DDBs (device-dependent bitmaps), which are different from the DIBs (device-independent bitmaps) represented by *TDib* objects.

Public constructors

Constructors

Form 1 TBitmap(HBITMAP handle, TAutoDelete autoDelete = NoAutoDelete);

Creates a *TBitmap* object and sets the *Handle* data member to the given borrowed *handle*. The *ShouldDelete* data member defaults to false, ensuring that the borrowed handle will not be deleted when the C++ object is destroyed.

Form 2 TBitmap(const TClipboard& clipboard);

Creates a *TBitmap* object with values from the given Clipboard.

Form 3 TBitmap(const TBitmap& bitmap);
Creates a copy of the given *bitmap* object.

Form 4 TBitmap(int width, int height, uint8 planes=1, uint8 bitCount=1, void far* bits=0);
Creates a bitmap object from *bitCount* bits in the *bits* buffer with the given *width, height,* and *planes* argument values.

Form 5 TBitmap(const BITMAP far* bitmap);
Creates a bitmap object with the values found in the given *bitmap* structure.

Form 6 TBitmap(const TDC& Dc, int width, int height, bool discardable = false);
Creates a bitmap object for the given device context with the given argument values.

Form 7 TBitmap(const TDC& Dc, const TDib& dib, uint32 usage=CBM_INIT);
Creates a bitmap object for the given device context with the given *dib* and *usage* argument values.

Form 8 TBitmap(const TMetaFilePict& metaFile, TPalette& palette, const TSize& size);
Creates a bitmap object from the given *metaFile* using the given *palette* and *size* arguments.

Form 9 TBitmap(const TDib& dib, const TPalette* palette = 0);
Creates a bitmap object from the given *dib* and *palette* arguments. A working palette constructed from the DIB's color table is used if no palette is supplied. The default system palette can also be passed using *&TPalette::GetStock(TPalette::Default);*

Form 10 TBitmap(HINSTANCE instance, TResID resID);
Creates a bitmap object for the given application instance from the given resource.

See also TClipBoard::GetClipboardData, TDC, TDib, TGdiObject::Handle, TGdiObject::ShouldDelete, TPalette, BITMAP struct

Public member functions

BitsPixel
uint8 BitsPixel() const;
Returns the number of bits per pixel in this bitmap.

See also TBitmap::GetObject

GetBitmapBits
uint32 GetBitmapBits(uint32 count, void far* bits) const;
Copies up to *count* bits of this bitmap to the buffer *bits*.

GetBitmapDimension
bool GetBitmapDimension(TSize& size) const;
Retrieves the size of this bitmap (width and height, measured in tenths of millimeters) and sets it in the *size* argument. Returns **true** if the call is successful; otherwise returns **false**.

See also TSize

GetObject
bool GetObject(BITMAP far& bitmap) const;

Retrieves data (width, height, and color format) for this bitmap and sets it in the given *BITMAP* structure. To retrieve the bit pattern, use *GetBitmapBits*.

See also TBitMap::GetBitmapBits, BITMAP struct

Height

int Height() const;

Returns the height of this bitmap.

See also TBitmap::GetObject

operator <<

TClipboard& operator <<(TClipboard& clipboard, TBitmap& bitmap);

Copies the given *bitmap* to the given *clipboard* argument. Returns a reference to the resulting Clipboard, which allows normal chaining of **<<**.

See also TClipboard

operator HBITMAP()

operator HBITMAP() const;

Typecasting operator. Converts this bitmap's *Handle* to type *HBITMAP* (the data type representing the handle to a physical bitmap).

Planes

uint8 Planes() const;

Returns the number of planes in this bitmap.

See also TBitmap::GetObject

SetBitmapBits

uint32 SetBitmapBits(uint32 count, const void far* bits);

Copies up to *count* bits from the *bits* buffer to this bitmap.

SetBitmapDimension

bool SetBitmapDimension(const TSize& size, TSize far* oldSize=0);

Sets the size of this bitmap from the given *size* argument (width and height, measured in tenths of millimeters). The previous size is set in the *oldSize* argument. Returns **true** if the call is successful; otherwise returns **false**.

See also TSize

ToClipboard

void ToClipboard(TClipboard& clipboard);

Copies this bitmap to the given Clipboard.

See also TClipboard::SetClipboardData

Width

int Width() const;

Returns the width of this bitmap.

See also TBitmap::GetObject

Protected constructor

Constructor
TBitMap();
Protected constructor for a *TBitmap* object.

Protected member functions

Create
void Create(const TDib& dib, const TPalette &palette);
void Create(const TBitmap &src);
Creates a bitmap handle from the given argument objects.

See also TDib, TPalette

Operators

operator <<
TClipboard& operator <<(TClipboard& clipboard, TBitmap& bitmap);
Copies the given *bitmap* to the given *clipboard* argument. Returns a reference to the resulting Clipboard, which allows normal chaining of <<.

See also TClipboard

operator HBITMAP()
operator HBITMAP() const;
Typecasting operator. Converts this bitmap's *Handle* to type *HBITMAP* (the data type representing the handle to a physical bitmap).

TBitmapGadget class

bitmapga.h

Derived from *TGadget*, *TBitmapGadget* is a simple gadget that can display an array of bitmap images one at a time.

Public constructor and destructor

Constructor
TBitmapGadget(TResId bmpResId, int id, TBorderStyle borderStyle, TResId bitmapName, int numImages, int startImage);
Constructs a *TBitmapGadget* and sets the current image to the beginning image (*startImage*) in the array of images. Then, sets the border style to the current *TGadget* border style and *numImages* to the number of images in the array.

Destructor
~TBitmapGadget();
Deletes the array of images.

Public member functions

SelectImage

int SelectImage(int imageNum, bool immediate);

Determines the current image and repaints the client area if the image has changed. Updates the client area if the image has changed.

SysColorChange

void SysColorChange();

When the system colors have been changed, *SysColorChange* is called by the gadget window's *EvSysColorChange* so that bitmap gadgets can be rebuilt and repainted.

Protected member functions

GetDesiredSize

void GetDesiredSize(TSize& size);

Calls *TGadget::GetDesiredSize*, which determines how big the bitmap gadget can be. The gadget window sends this message to query the gadget's size. If shrink-wrapping is requested, *GetDesiredSize* returns the size needed to accommodate the borders and margins. If shrink-wrapping is not requested, it returns the gadget's current width and height. *TGadgetWindow* needs this information to determine how big the gadget needs to be, but it can adjust these dimensions if necessary. If *WideAsPossible* is **true**, then the width parameter (*size.cx*) is ignored.

Paint

void Paint(TDC& dc);

Paints the gadget's border and the contents of the bitmap.

See also TGadget::Paint

SetBounds

void SetBounds(TRect& r);

Calls *TGadget::SetBounds* and passes the dimensions of the bitmap gadget. *SetBounds* informs the control gadget of a change in its bounding rectangle.

See also TGadget::SetBounds

TBitSet class

bitset.h

TBitSet sets or clears a single bit or a group of bits. You can use this class to set and clear option flags and to retrieve information about a set of bits. The class *TCharSet* performs similar operations for a string of characters.

Public constructors

Constructors

Form 1 TBitSet();

Constructs a *TBitSet* object.

Form 2 TBitSet(const TBitSet& bs);
Constructs a *TBitSet* object as a copy of another *TBitSet*.

Public member functions

DisableItem

Form 1 void DisableItem(int item);
Clears a single bit at *item*.

Form 2 void DisableItem(const TBitSet& bs);
Clears the set of bits enabled in *bs*.

EnableItem

Form 1 void EnableItem(int item);
Sets a single bit at *item*.

Form 2 void EnableItem(const TBitSet& bs);
Sets the set of bits enabled in *bs*.

Has

int Has(int item);
Is nonzero if *item* is in the set of bits.

IsEmpty

int TBitSet::IsEmpty();
Is nonzero if the set is empty; otherwise, is 0.

operator +=

Form 1 TBitSet& operator +=(int item);
Calls *EnableItem* to set a bit in the copied set. Returns a reference to the copied *TBitSet* object.

Form 2 TBitSet& operator +=(const TBitSet& bs);
Calls *EnableItem* to set the bits enabled in *bs*. Returns a reference to the copied *TBitSet* object.

operator –=

Form 1 TBitSet& operator –=(int item);
Calls *DisableItem* to clear a bit in the set. Returns a reference to the copied *TBitSet* object.

Form 2 TBitSet& operator –=(const TBitSet& bs);
Calls *DisableItem* to clear the bits enabled in *bs*. Returns a reference to the copied *TBitSet* object.

operator &=

TBitSet& operator &=(const TBitSet&);
ANDs all the bits in the copied set and returns a reference to the copied *TBitSet* object.

operator |=

TBitSet& operator |=(const TBitSet&);
ORs all of the bits in the copied set and returns a reference to the copied *TBitSet* object.

operator ~

TBitSet operator ~(const TBitSet&);

Returns the set of bits that is the opposite of a specified set of bits. For example, if the set of bits is 01010101, the returned set is 10101010. Returns a reference to the copied *TBitSet* object.

TBIVbxLibrary class

vxbctl.h

Defined in vbxctl.h and virtually derived from *TModule*, *TBIVbxLibrary* handles loading and initializing of BIVBX10.DLL. If you want to use VBX controls, construct a *TBIVbxLibrary* object with the same scope as your application. For example,

```
int OwlMain(int, char**)
{
  TBIVbxLibrary vbxLib;
  return TTestApp().Run();
}
```

Public constructor and destructor

Constructor

TBIVbxLibrary();

Constructs a *TBIVbxLibrary* object.

Destructor

~TBIVbxLibrary();

Destroys a *TBIVbxLibrary* object.

TBrush class

gdiobjec.h

The GDI Brush class is derived from *TGdiObject*. *TBrush* provides constructors for creating solid, styled, or patterned brushes from explicit information. It can also create a brush indirectly from a borrowed handle.

Public constructors

Constructors

Form 1 TBrush(HBRUSH handle, TAutoDelete autoDelete = NoAutoDelete);

Creates a *TBrush* object and sets the *Handle* data member to the given borrowed *handle*. The *ShouldDelete* data member defaults to **false**, ensuring that the borrowed handle will not be deleted when the C++ object is destroyed.

Form 2 TBrush(TColor color);

Creates a solid *TBrush* object with the given color. To save a brush creation this constructor uses a cache that can detect any color that matches a stock color.

Form 3 TBrush(TColor color, int style);

Creates a hatched *TBrush* object with the given style and color.

Form 4 TBrush(const TBitmap& pattern);
Creates a patterned *TBrush* object with the given pattern.

Form 5 TBrush(const TDib& pattern);
Creates a patterned *TBrush* object with the given DIB pattern.

Form 6 TBrush(const LOGBRUSH far* logBrush);
Creates a *TBrush* object with values from the given *logBrush*.

See also TBitmap, TColor, TDib, TGdiObject::Handle, TGdiObject::ShouldDelete, LOGBRUSH struct

Public member functions

GetObject

bool GetObject(LOGBRUSH far& logBrush) const;
Retrieves information about this brush object and places it in the given *LOGBRUSH* structure. Returns **true** if the call is successful; otherwise returns **false**.

See also TGdiObject::GetObject, LOGBRUSH struct

operator HBRUSH()

operator HBRUSH() const;
Typecasting operator. Converts this brush's *Handle* to type HBRUSH (the data type representing the handle to a physical brush).

UnrealizeObject

bool UnrealizeObject();
Directs the GDI to reset the origin of this brush the next time it is selected. Returns **true** if call is successful; otherwise returns **false**.

TButton Class

button.h

TButton is an interface class that represents a pushbutton interface element. You must use a *TButton* to create a button control in a parent *TWindow*. You can also use a *TButton* to facilitate communication between your application and the button controls of a *TDialog*. This class is streamable.

There are two types of pushbuttons: regular and default. Regular buttons have a thin border. Default buttons (which represent the default action of the window) have a thick border.

Public data members

IsDefPB

bool IsDefPB;

Indicates whether the button is to be considered the default pushbutton. Used for owner-draw buttons, *IsDefPB* is set by a *TButton* constructor based on BS_DEFPUSHBUTTON style setting.

Public constructors

Constructors

Form 1 `TButton(Window *parent, int Id, const char far* text, int X, int Y, int W, int H, bool isDefault = false, TModule* module = 0);`

Constructs a button object with the supplied parent window (*parent*), control ID (*Id*), associated text (*text*), position (*X, Y*) relative to the origin of the parent window's client area, width (*W*), and height (*H*). If *IsDefault* is **true**, the constructor adds BS_DEFPUSHBUTTON to the default styles set for the *TButton* (in *Attr.Style*). Otherwise, it adds BS_PUSHBUTTON.

Form 2 `TButton(TWindow* parent, int resourceID, TModule* module = 0);`

Constructs a *TButton* object to be associated with a button control of a *TDialog*. Calls *DisableTransfer* to exclude the button from the transfer mechanism because there is no data to be transferred.

The *resId* parameter must correspond to a button resource that you define.

See also TControl::TControl

Protected data member

IsCurrentDefPB

`bool IsCurrentDefPB;`

Indicates whether the current button is the default pushbutton.

Protected member functions

BMSetStyle

`LRESULT BMSetStyle(WPARAM, LPARAM);`

Because a button can't have both owner-drawn and pushbutton styles, *BMSetStyle* keeps track of the style if the button is owner-drawn and the system tries to set the style to BS_DEFPUSHBUTTON. *BMSetStyle* sets *IsCurrentDefPB* to **true** if the button style is BS_DEFPUSHBUTTON.

EvGetDlgCode

`uint EvGetDlgCode(MSG far*);`

Responds to WM_GETDLGCODE messages that are sent to a dialog box associated with a control. *EvGetDlgCode* allows the dialog manager to intercept a message that would normally go to a control and then ask the control if it wants to process this message. If not, the dialog manager processes the message. The *msg* parameter indicates the kind of message, for example a control, command, or button message, sent to the dialog box manager.

EvGetDlgCode returns a code that indicates how the button is to be treated. If this is the currently used pushbutton, *EvGetDlgCode* returns either DLGC_DEFPUSHBUTTON or DLGC_UNDEFPUSHBUTTON.

See also DLGC_xxxx dialog control message constants

GetClassName

char far* GetClassName();

Overrides *TWindow*'s *GetClassName* function. If BWCC is enabled, returns the name of *TButton*'s registration class, "BUTTON_CLASS"; if BWCC isn't enabled, returns the name "BUTTON."

SetupWindow

void SetupWindow();

Overrides *TWindow's SetupWindow* function. If the button is the default pushbutton and an owner-drawn button, *SetupWindow* sends a DM_SETDEFID message to the parent window.

Response table entries

Response table entry	Member function
EV_WM_GETDLGCODE	*EVGetDlgCode*
EV_MESSAGE (BM_SETSTYLE, BMSetStyle)	*BMSetStyle*

TButtonGadget class

buttonga.h

Derived from *TGadget, TButtonGadgets* represent buttons that you can click on or off. You can also apply attributes such as color, style, and shape (notched or unnotched) to your button gadgets.

In general, button gadgets are classified as either command or attribute buttons. Attribute buttons include radio buttons (which are considered exclusive), or check boxes (which are nonexclusive). The public data member, *TType*, enumerates these button types.

TButtonGadget contains several functions that let you change the style of a button. Use *SetAntialiasEdges* to turn antialiasing on and off, *SetNotchCorners* to control corner notching, and *SetShadowStyle* to change the style of the button shadow.

TButtonGadget objects respond to mouse events in the following manner: when a mouse button is pressed, the button is pressed; when the mouse button is released, the button is released. Commands can be entered only when the mouse button is in the "up" state. When the mouse is pressed, *TButtonGadget* objects capture the mouse and reserve all mouse messages for the current window. When the mouse button is up, button gadgets release the capture for the current window. The public data member, *TState*, enumerates the three button states.

Type definitions

TShadowStyle

enum TShadowStyle;

Enumerates button shadow styles—either single (1) or double (2) shadow borders.

TState

enum TState;

TState enumerates the three button positions during which the button can be pressed: up (0), down (1), and an indeterminate state (2). A nonzero value indicates a highlighted button.

TType

enum TType;

Enumerates the types of buttons: command, exclusive, or nonexclusive.

Public constructor and destructor

Constructor

TButtonGadget(TResId bmpResId, int id, TType type = Command, bool enabled = false, TState state = Up, bool repeat = false);

Constructs a *TButtonGadget* object using the specified bitmap ID, button gadget ID, and type, with enabled set to **false** and in a button-up state. The button isn't enabled—its initial state before command enabling occurs.

Destructor

~TButtonGadget();

Deconstructs a *TButtonGadget* object.

See also TButtonGadget::TState

Public member functions

CommandEnable

void CommandEnable();

Enables the button gadget to capture messages. Calls *SendMessage* to send a WM_COMMAND_ENABLE message to the gadget window's parent, passing a *TCommandEnable: EvCommandEnable* message for this button.

GetButtonState

TState GetButtonState();

Returns the state of the button. If 0, the button is up, if 1, the button is down, if 2, the state is indeterminate.

See also TButtonGadget::TState

GetButtonType

TType GetButtonType();

Returns the button type as 1 if the button is a command, 2 if exclusive, or 3 if nonexclusive.

SetAntialiasEdges

SetAntialiasEdges(bool anti = true);

Turns the antialiasing of the button bevels on or off. By default, antialiasing is on.

SetButtonState

void SetButtonState(TState);

Sets the state of the button. If the state has changed, the button is exclusive, and is in the down state, checks that the button is exclusive, sets *State*, and calls *Invalidate* to mark the changed area of the gadget for repainting.

See also TButtonGadget::TState

SetNotchCorners

void SetNotchCorners(bool NotchCorners = true);

By default, *SetNotchCorners* implements notched corners for buttons. To repaint the frame of the button if the window has already been created, call *InvalidateRect* with the *Bounds* rectangle.

See also TButtonGadget::Invalidate, TGadget::InvalidateRect, TGadget::Paint

SetShadowStyle

void SetShadowStyle(TShadowStyle);

Sets the button style to a shadow style which, by default, is DoubleShadow. Sets the left and top borders to 2 and the right and bottom borders to *ShadowStyle* + 1.

SysColorChange

void SysColorChange();

SysColorChange responds to an *EvSysColorChange* message forwarded by the owning *TGadgetWindow* by setting the dither brush to zero. If a user-interface bitmap exists, *SysColorchange* deletes and rebuilds it to get the new button colors.

Protected data members

AntialiasEdges

bool AntialiasEdges;

Is true if antialiasing is turned on.

BitmapOrigin

TPoint BitmapOrigin;

Points to the x and y coordinates of the bitmap used for this button gadget.

CelArray

TCelArray* CelArray;

The array of cels used by this button gadget.

NotchCorners

bool NotchCorners;

Initialized to 1, *NotchCorners* is 1 if the button gadget has notched corners or 0 if it doesn't have notched corners.

Pressed
bool Pressed;
Initialized to 1, *Pressed* is 1 if the button is released or 0 if it isn't released.

See also TButtonGadget::Activate, TButtonGadget::BeginPressed, TButtonGadget::CancelPressed

Repeat
bool Repeat;
Initialized to 1, *Repeat* stores the repeat count for keyboard events.

ResId
TResId ResId;
Holds the resource ID for this button gadget's bitmap.

ShadowStyle
TShadowStyle ShadowStyle;
Holds the shadow style for the button—1 for single and 2 for double.

State
TState State;
Holds the state of the button—either up, down, or indeterminate.

Type
TType Type;
Holds the type of the button—either command, exclusive, or nonexclusive.

Protected member functions

Activate
virtual void Activate(TPoint& p);
Invoked when the mouse is in the "up" state, *Activate* sets *Pressed* to **false**, changes the state for attribute buttons, and paints the button in its current state. To do this, it calls *CancelPressed*, posts a WM_COMMAND message to the gadget window's parent, and sends menu messages to the gadget window's parent.

See also TButtonGadget::Pressed

BeginPressed
virtual void BeginPressed(TPoint& p);
When the mouse button is pressed, *Beginpressed* sets *Pressed* to **true**, paints the pressed button, and sends menu messages to the gadget window's parent.

See also TButtonGadget::Pressed

BuildCelArray
virtual void BuildCelArray();
Builds a cel array using the resource bitmap as the base glyph. Any existing cel array should be deleted if a replacement is built.

See also TCelArray

CancelPressed

virtual void CancelPressed(TPoint& p);

When the mouse button is released, *CancelPressed* sets *Pressed* to **false**, paints the button, and sends menu messages to the gadget window's parent.

See also TButtonGadget::Pressed

GetDesiredSize

void GetDesiredSize(TSize& size);

Stores the width and height (measured in pixels) of the button gadget in size. Calls *TGadget's GetDesiredSize* to calculate the relationship between one rectangle and another.

GetGlyphDib

virtual TDib* GetGlyphDib();

Supplies the *glyphdib*. You can override this function to get a different *dib*, or to change the attributes of the *dib*, such as the colors, and so on.

Invalidate

void Invalidate();

If a button is pressed or the state of the button is changed, *Invalidate* invalidates (marks for repainting) the changed area of the gadget. *Invalidate* only invalidates the area that changes. To repaint the entire gadget, call *TGadget::InvalidateRect* and pass the rectangle's boundaries.

See also TButtonGadget::TState, TGadget::InvalidateRect

LButtonDown

void LButtonDown(uint modKeys, TPoint& p);

Overrides *TGadget* member function and responds to a left mouse button click by calling *BeginPressed*.

See also TButtonGadget::BeginPressed

LButtonUp

void LButtonUp(uint modKeys, TPoint& p);

Overrides *TGadget* member functions and responds to a release of the left mouse button by calling *Activate*.

See also TButtonGadget::Activate

MouseEnter

void MouseEnter(uint modKeys, TPoint& p);

Called when the mouse enters the boundary of the button gadget. *modKeys* indicates the virtual key information and can be any combination of the following values: MK_CONTROL, MK_LBUTTON, MK_MBUTTON, MK_RBUTTON, or MK_SHIFT. *p* indicates where the mouse entered the button gadget.

MouseLeave

void MouseLeave(uint modKeys, TPoint& p);

Called when the mouse leaves the boundary of the button gadget. *modKeys* indicates the virtual key information and can be any combination of the following values:

MK_CONTROL, MK_LBUTTON, MK_MBUTTON, MK_RBUTTON, or MK_SHIFT. *p* indicates the place where the mouse left the button gadget.

MouseMove

void MouseMove(uint modKeys, TPoint& p);

Calls *TGadget::MouseMove* in response to the mouse being dragged. If the mouse moves off the button, *MouseMove* calls *CancelPressed*. If the mouse moves back onto the button, *MouseMove* calls *BeginPressed*.

See also TButtonGadget::BeginPressed, TButtonGadget::CancelPressed

Paint

void Paint(TDC& dc);

Gets the width and height of the window frame (in pixels), calls *GetImageSize* to retrieve the size of the bitmap, and sets the inner rectangle to the specified dimensions. Calls *TGadget::PaintBorder* to perform the actual painting of the border of the control. Before painting the control, *Paint* determines whether the corners of the control are notched, and then calls *GetSysColor* to see if highlighting or shadow colors are used. Paint assumes the border style is plain. Then, *Paint* draws the top, left, right, and bottom of the control, adjusts the position of the bitmap, and finishes drawing the control using the specified embossing, fading, and dithering.

ReleaseGlyphDib

virtual void ReleaseGlyphDib(TDib* glyph);

Releases the *glyph dib* depending on how it was obtained by *GetGlyphDib*.

SetBounds

void SetBounds(TRect& r);

Gets the size of the bitmap, calls *TGadget::SetBounds* to set the boundary of the rectangle, and centers the bitmap within the button's rectangle.

See also TGadget::SetBounds

TButtonGadgetEnabler class — buttonga.cpp

Derived from *TCommandEnabler*, *TButtonGadgetEnabler* serves as a command enabler for button gadgets. The functions in this class modify the text, check state, and appearance of a button gadget.

Public constructor

Constructor

TButtonGadgetEnabler(HWND hWndReceiver, TButtonGadget* g)

Constructs a *TButtonGadgetEnabler* for the specified gadget. *hWndReceiver* is the window receiving the message.

Protected data member

gadget
TButtonGadget* gadget;
The button gadget being enabled or disabled.

Public member functions

Enable
void Enable(bool enable);
Overrides *TCommandEnable::Enable*. Enables or disables the keyboard, mouse input and appearance of the corresponding button gadget.

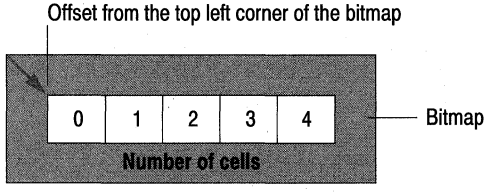
SetCheck
void SetCheck(int state)
Overrides *TCommandEnable::SetCheck*. Changes the check state of the corresponding button gadget.

SetText
void SetText(const char far*)
Overrides *TCommandEnable::SetText*. Changes the text of the corresponding button gadget.

TCelArray class

celarray.h

TCelArray is a horizontal array of cels (a unit of animation) created by slicing a portion of or an entire bitmap into evenly sized shapes. Gadgets such as buttons can use a *TCelArray* to save resource space. *TCelArray's* functions let you control the dimensions of each cel and determine if the cel can delete the bitmap.



Public constructors and destructor

Constructors

Form 1 TCelArray(TBitmap* bmp, int numCels, TSize celSize = 0, TPoint Offset = 0, TAutoDelete = AutoDelete);
Constructs a *TCelArray* from a bitmap by slicing the bitmap into a horizontal array of cels of a specified size. If *autoDelete* is **true**, *TCelArray* can automatically delete the bitmap. The *ShouldDelete* data member defaults to **true**, ensuring that the handle will be deleted when the bitmap is destroyed.

Form 2 TCelArray(TDib& dib, int numCels);

Constructs a *TCelArray* from a DIB (Device Independent Bitmap) by slicing the DIB into a horizontal array of evenly sized cels.

Form 3 `TCelArray(const TCelArray& src);`
Constructs a *TCelArray* as a copy of an existing one. If the original *TCelArray* owned its bitmap, the constructor copies this bitmap; otherwise, it keeps a reference to the bitmap.

Destructor

`virtual ~TCelArray();`
If *ShouldDelete* is **true** (the default value), the bitmap is deleted. If *ShouldDelete* is **false**, no action is taken.

Public member functions

CelSize

`TSize CelSize() const;`
Returns the size in pixels of each cell.

CelOffset

`TPoint CelOffset(int cels);`
Returns the position of the upper left corner of a given cel relative to the upper left corner of the bitmap.

CelRect

`TRect CelRect(int cel) const;`
Returns the upper left and lower right corner of a given cell relative to the upper left corner of the bitmap.

NumCels

`int NumCels() const;`
Returns the number of cels in the array.

Offset

`TPoint Offset() const;`
Returns the offset of the entire *CelArray*.

operator []

`TRect operator [](int cel) const;`
Returns *CelRect*.

operator =

`TCelArray& operator =(const TCelArray&);`
Returns *TCelArray*.

operator TBitmap&()

`operator TBitmap&();`
Returns a reference to the bitmap.

SetCelSize

`void SetCelSize(TSize size);`
Sets the size of each cel in the array.

SetOffset
void SetOffset(TPoint offs);
Sets the offset for the cels in the array.

SetNumCels
void SetNumCels(int numCels);
Sets the number of cels in the array.

Protected data members

Bitmap
TBitmap* Bitmap;
Points to the bitmap.

CSize
TSize CSize;
The size of a cell in the array.

Offs
TPoint Offs;
Holds the offset of the upper left corner of the cel array from the upper left corner of the bitmap.

NCels
int NCels;
The number of cells in the cel array.

ShouldDelete
bool ShouldDelete;
Is **true** if the destructor needs to delete the bitmap associated with the cel array.

TCharSet class

bitset.h

Derived from *TBitSet*, *TCharSet* sets and clears bytes for a group of characters. You can use this class to set or clear bits in a group of characters, such as the capital letters from "A" through "Z" or the lowercase letters from "a" through "z." The class *TBitSet* performs similar operations for a group of bits.

Public constructors

Constructors

Form 1 TCharSet();
Constructs a *TCharSet* object.

Form 2 TCharSet(const TCharSet&);
Copy constructor for a *TCharSet* object.

Form 3 TCharSet(const char far* str);

Constructs a string of characters.

Public member function

operator !=
int operator !=(const TBitSet& bs1, const TBitSet& bs2);
ORs all of the bits in the copied string and returns a reference to the copied *TCharSet* object.

TCheckBox class

checkbox.h

TCheckBox is a streamable interface class that represents a check box control. Use *TCheckBox* to create a check box control in a parent window. You can also use *TCheckBox* objects to more easily manipulate check boxes you created in a dialog box resource.

Two-state check boxes can be *checked* or *unchecked*; three-state check boxes have an additional *grayed* state. *TCheckBox* member functions let you easily control the check box's state. A check box can be in a group box (a *TGroupBox* object) that groups related controls.

Public data member

Group
TGroupBox* Group;
If the check box belongs to a group box (a *TGroupBox* object), *Group* points to that object. If the check box is not part of a group, *Group* is zero.

See also TGroupBox::TGroupBox

Public constructors

Constructors

Form 1 TCheckBox(TWindow* parent, int Id, const char far* title, int x, int y, int w, int h, TGroupBox* group = 0, TModule* module = 0);
Constructs a check box object with the specified parent window (*parent*), control ID (*Id*), associated text (*title*), position relative to the origin of the parent window's client area (*x*, *y*), width (*w*), height (*h*), associated group box (*group*), and owning module (*module*). Invokes the *TButton* constructor with similar parameters. Sets the check box's style to WS_CHILD | WS_VISIBLE | WS_TABSTOP | BS_AUTOCHECKBOX.

Form 2 TCheckBox(TWindow* parent, int resourceId, TGroupBox* group = 0, TModule* module = 0);
Constructs an associated *TCheckBox* object for the check box control with a resource ID of *resourceId* in the parent dialog boxSets *Group* to *group* then enables the data transfer mechanism by calling *EnableTransfer*.

See also TButton::TButton, TWindow::EnableTransfer

Public member functions

Check

void Check();

Forces the check box to be checked by calling *SetCheck* with the value of BF_CHECKED. Notifies the associated group box, if there is one, that the state was changed.

See also TCheckBox::GetCheckTCheckBox_GetCheck, TCheckBox::ToggleTCheckBox_Toggle, TCheckBox::UncheckTCheckBox_Uncheck, TGroupBox::SelectionChangedTGroupBox_SelectionChanged

GetCheck

uint GetCheck() const;

Returns the state of the check box.

TCheckBox check states

Check box state	Return value
Checked	BF_CHECKED
Unchecked	BF_UNCHECKED
Grayed	BF_GRAYED

See also TCheckBox::SetCheck

GetState

uint GetState() const;

Returns the check, focus, and highlight state of the check box.

See also TCheckBox::SetState

SetCheck

void SetCheck(uint check);

Forces the check box into the state specified by *check*. See the table in *GetCheck* for possible values of *check*.

See also TCheckBox::GetCheck

SetState

void SetState(uint state);

Sets the check, focus, and highlight state of the check box.

See also TCheckBox::GetState

SetStyle

void SetStyle(uint style, bool redraw);

Changes the style of the check box.

Toggle

void Toggle();

Toggles the check box between checked and unchecked if it is a two-state check box; toggles it between checked, unchecked, and gray if it is a three-state check box.

See also TCheckBox::SetCheck

Transfer

uint Transfer(void* buffer, TTransferDirection direction);

Overrides *TWindow::Transfer*. Transfers the check state of the check box to or from *buffer*, using the values specified in the table in *GetCheck*. If *direction* is *tdGetDate*, the check box state is transferred into the buffer. If *direction* is *tdSetData*, the check box state is changed to the settings in the transfer *buffer*.

Transfer returns the size of the transfer data in bytes. To get the size without actually transferring the check box, use *tdSizeData* as the *direction* argument.

Uncheck

void Uncheck();

Forces the check box to be unchecked by calling *SetCheck* with a value of BF_UNCHECKED. Notifies the associated group box, if there is one, that the state has changed.

See also TCheckBox::Check, TCheckBox::SetCheck, TCheckBox::Toggle

Protected member functions

BNClicked

void BNClicked();

Responds to notification message BN_CLICKED, indicating that the user clicked the check box. If *Group* isn't 0, *BNClicked* calls the group box's *SelectionChanged* member function to notify the group box that the state has changed.

See also TGroupBox::SelectionChanged

EvGetDlgCode

uint EvGetDlgCode(MSG far* msg);

Overrides *TButton's* response to the WM_GETDLGCODE message, an input procedure associated with a control that is not a check box, by calling *DefaultProcessing*. The *msg* parameter indicates the kind of message, for example a control, command, or check box message, sent to the dialog box manager.

EvGetDlgCode returns a code that indicates how the check box is to be treated.

See also TButton::EvGetDlgCode, TWindow::DefaultProcessing, DLGC_xxxx dialog control message constants

GetClassName

char far* GetClassName();

If BWCC is enabled, *TCheckBox* returns CHECK_CLASS. If BWCC is not enabled, returns "BUTTON."

Response table entries

Response table entry	Member function
EV_NOTIFY_AT_CHILD (BN_CLICKED, BNClicked)	*BNClicked*
EV_WM_GETDLGCODE	*EVGetDlgCode*

TChooseColorDialog class

chooseco.h

TChooseColorDialog objects represent modal dialog box interface elements that allow color selection and custom color adjustment. *TChooseColorDialog* can be made to appear modeless to the user by creating the dialog's parent as an invisible pop-up window and making the pop-up window a child of the main application window. *TChooseColorDialog* uses the *TChooseColor::TData* struct to initialize the dialog box with the user's color selection.

Public constructors

Constructor

```
TChooseColorDialog(TWindow* parent, TData& data, TResId templateID = 0, const char far* title = 0,
  TModule* module = 0);
```

Constructs a dialog box with specified parent window, data, resource identifier, window caption, and module ID. Sets the attributes of the dialog box based on info in the *TChooseColor::TData* structure.

See also TChooseColorDialog::TData

Public member function

SetRGBColor

```
void SetRGBColor(TColor color);
```

Sets the current RGB color for the open dialog box by sending a *SetRGBMsgId*. You can use *SetRGBColor* to send a message to change the current color selection.

Public data members

cc

```
CHOOSECOLOR cc;
```

Stores the length of the *TChooseColorDialog* structure, the window that owns the dialog box, and the data block that contains the dialog template. It also points to an array of 16 RGB values for the custom color boxes in the dialog box, and specifies the dialog-box initialization flags.

See also TChooseColorDialog::TData

Data

```
TData& Data;
```

Data is a reference to the *TData* object passed in the constructor.

See also TChooseColorDialog::TData

SetRGBMsgId

static uint SetRGBMsgId;

Contains the ID of the registered message sent by *SetRGBColor*.

Protected member functions

DialogFunction

bool DialogFunction(uint message, WPARAM, LPARAM);

Returns **true** if a message is handled.

See also TDialog::DialogFunction

DoExecute

int DoExecute();

If no error occurs, *DoExecute* copies flags and colors into *Data* and returns zero. If an error occurs, *DoExecute* returns the IDCANCEL with *Data.Error* set to the value returned from *CommDlgExtendedError*.

EvSetRGBColor

LPARAM EvSetRGBColor(WPARAM, LPARAM);

Responds to the message sent by *SetRGBColor* by forwarding the to the original class. This event handler is not in the response table.

Response table entries

The *TChooseColorDialog* response table has no entries.

TChooseColorDialog::TData class

chooseco.h

Defines information necessary to initialize the dialog box with the user's color selection.

Public data members

Color

TColor Color;

Specifies the color that is initially selected when the dialog box is created. Contains the user's color selection when the dialog box is closed.

CustColors

TColor* CustColors;

Points to an array of 16 colors.

Error

uint32 Error;

If the dialog box is successfully executed, *Error* is 0. Otherwise, it contains one of the following codes:

Constant	Meaning
CDERR_DIALOGFAILURE	Failed to create a dialog box.
CDERR_FINDRESFAILURE	Failed to find a specified resource.
CDERR_LOADRESFAILURE	Failed to load a specified resource.
CDERR_LOCKRESOURCEFAILURE	Failed to lock a specified resource.
CDERR_LOADSTRFAILURE	Failed to load a specified string.

Flags

```
uint32 Flags;
```

Flags can be a combination of the following values that control the appearance and functionality of the dialog box:

Constant	Meaning
CC_FULLOPEN	Causes the entire dialog box to appear when the dialog box is created.
CC_PREVENTFULLOPEN	Disables the "Define Custom Colors" push button.
CC_RGBINIT	Causes the dialog box to use the color specified in *rgbResult* as the initial color selection.
CC_SHOWHELP	Causes the dialog box to show the Help push button.

See also TChooseColorDialog::Data

TChooseFontDialog class

choosefo.h

A *TChooseFontDialog* represents modal dialog-box interface elements that create a system-defined dialog box from which the user can select a font, a font style (such as bold or italic), a point size, an effect (such as strikeout or underline), and a color. *TChooseFontDialog* can be made to appear modeless by creating the dialog's parent as an invisible pop-up window and making the pop-up window a child of the main application window. *TChooseFontDialog* uses the *TChooseFontDialog::TData* structure to initialize the dialog box with the user-selected font styles.

Public constructor

Constructor

```
TChooseFontDialog(TWindow* parent, TData& data, TResID templateID = 0, const char far* title = 0,
  TModule* module = 0);
```

Constructs a dialog box with specified data, parent window, resource identifier, window caption, and module ID. Sets the attributes of the dialog box based on the font information in the *TChooseFontDialog::TData* structure.

See also TChooseFontDialog::TData

Protected data members

cf

CHOOSEFONT cf;

Contains font attributes. *cf* is initialized using fields in the *TChooseFontDialog::TData* structure. It stores the length of the structure, the window that owns the dialog box and the data block that contains the dialog template. It also specifies the dialog-box initialization flags.

See also TChooseFontDialog::TData

Data

TData& Data;

Data is a reference to the *TData* object passed in the constructor.

See also TChooseFontDialog::TData

Protected member functions

CmFontApply

void CmFontApply();

Default handler for the third pushbutton (the Apply button) in the dialog box.

DialogFunction

bool DialogFunction(uint message, WPARAM, LPARAM);

Returns **true** if a dialog box message is handled.

See also TDialog::DialogFunction

DoExecute

int DoExecute();

If no error occurs, *DoExecute* copies the flag values and font information into *Ddata*, and returns IDOK or IDCANCEL. If an error occurs, *DoExecute* returns an error code from *TChooseFontDialog::TData* structure's *Error* data member.

See also TChooseFontDialog::TData

Response table entries

The *TChooseFontDialog* response table contains no entries.

TChooseFontDialog::TData class

choosefo.h

Defines information necessary to initialize the dialog box with the user's font selection.

Public data members

Color

TColor Color;

Indicates the font color that is initially selected when the dialog box is created; contains the user's font color selection when the dialog box is closed.

DC

HDC DC;

Handle to the device context from which fonts are obtained.

Error

uint32 Error;

If the dialog box is successfully executed, *Error* returns 0. Otherwise, it contains one of the following codes:

Constant	Meaning
CDERR_DIALOGFAILURE	Failed to create a dialog box.
CDERR_FINDRESFAILURE	Failed to find a specified resource.
CDERR_LOCKRESOURCEFAILURE	Failed to lock a specified resource.
CDERR_LOADRESFAILURE	Failed to load a specified resource.
CDERR_LOADSTRFAILURE	Failed to load a specified string.
CFERR_MAXLESSTHANMIN	The size specified in *SizeMax* is less than the size in *SizeMin*.
CFERR_NOFONTS	No fonts exist.

Flags

uint32 Flags;

Flags can be a combination of the following constants that control the appearance and functionality of the dialog box:

Constant	Meaning
CF_APPLY	Enables the display and use of the Apply button.
CF_ANSIONLY	Specifies that the *ChooseFontDialog* structure allows only the selection of fonts that use the ANSI character set.
CF_BOTH	Causes the dialog box to list both the available printer and screen fonts.
CF_EFFECTS	Enables strikeout, underline, and color effects.
CF_FIXEDPITCHONLY	Enables fixed-pitch fonts only.
CF_FORCEFONTEXIST	Indicates an error if the user selects a nonexistent font or style.
CF_INITTOLOGFONTSTRUCT	Uses the *LOGFONT* structure at which *LogFont* points to initialize the dialog controls.
CF_LIMITSIZE	Limits font selection to those between *SizeMin* and *SizeMax*.
CF_NOSIMULATIONS	Does not allow GDI font simulations.
CF_PRINTERFONTS	Causes the dialog box to list only the fonts supported by the printer that is associated with the device context.
CF_SCALABLEONLY	Allows only the selection of scalable fonts.
CF_SCREENFONTS	Causes the dialog box to list only the system-supported screen fonts.
CF_SHOWHELP	Causes the dialog box to show the Help button.

Constant	Meaning
CF_TTONLY	Enumerates and allows the selection of TrueType® fonts only.
CF_USESTYLE	Specifies that *Style* points to a buffer containing the style attributes used to initialize the selection of font styles.
CF_WYSIWYG	Allows only the selection of fonts available on both the printer and the screen.

FontType
uint16 FontType;
Font type or name.

LogFont
LOGFONT LogFont;
Attributes of the font.

PointSize
int PointSize;
Point size of the font.

SizeMax
int SizeMax;
Maximum size of the font.

SizeMin
int SizeMin;
Minimum size of the font.

See also TChooseFontDialog::Data

Style
char far* Style;
Style of the font such as bold, italic, underline, or strikeout.

TClientDC Class

dc.h

A DC class derived from *TWindowDC*, *TClientDC* provides access to the client area owned by a window.

See also
TOleClientDC

Public constructors

Constructor
TClientDC(HWND wnd);
Creates a *TClientDC* object with the given owned window. The data member *Wnd* is set to *wnd*.

See also TWindowDC::Wnd, TDC::TDC

TClipboard class

clipboar.h

TClipboard encapsulates and manipulates clipboard data. You can open, close, empty, and paste data in a variety of data formats between the Clipboard and the open window. An object on the clipboard can exist in a variety of clipboard formats, which range from bitmaps to text.

Usually, the window is in charge of manipulating clipboard interactions between the window and the clipboard. It does this by responding to messages sent between the clipboard owner and the application. The following ObjectWindows event-handling functions encapsulate these clipboard messages:

EvRenderFormat—Responds to a WM_RENDERFORMAT message sent to the clipboard owner if a specific clipboard format that an application has requested hasn't been rendered. After the clipboard owner renders the data in the requested format, it calls *SetClipboardData* to place the data on the clipboard.

EvRenderAllFormats—Responds to a message sent to the clipboard owner if the clipboard owner has delayed rendering a clipboard format. After the clipboard owner renders data in all of possible formats, it calls *SetClipboardData*.

The following example tests to see if there is a palette on the clipboard. If one exists, *TClipboard* retrieves the palette, realizes it, and then closes the clipboard.

```
  if (clipboard.IsClipboardFormatAvailable(CF_PALETTE)) {
    newPal = new TPalette(TPalette(clipboard));  // make a copy
    UpdatePalette(true);
  }

// TryDIB format first.

  if (clipboard.IsClipboardFormatAvailable(CF_DIB)) {
    newDib = new TDib(TDib(clipboard));        // make a copy
    newBitmap = new TBitmap(*newDib, newPal);  // newPal==0 is OK
  // try metafile 2nd
  //
  } else if (clipboard.IsClipboardFormatAvailable(CF_METAFILEPICT)) {
    if (!newPal)
      newPal = new TPalette((HPALETTE)GetStockObject(DEFAULT_PALETTE));
    newBitmap = new TBitmap(TMetaFilePict(clipboard), *newPal,
                            GetClientRect().Size());
:
// Gets a bitmap , keeps it, and sets up data on the clipboard.

  //
  delete Bitmap;
  Bitmap = newBitmap;

  if (!newDib)
    newDib = new TDib(*newBitmap, newPal);
#endif
```

```
delete Dib;
Dib = newDib;

delete Palette;
Palette = newPal ? newPal : new TPalette(*newDib);
Palette->GetObject(Colors);

PixelWidth  = Dib->Width();
PixelHeight = Dib->Height();
AdjustScroller();
SetCaption("(Clipboard)");

clipboard.CloseClipboard();
```

Public destructor

Destructor

~TClipboard();

Destroys the *TClipboard* object.

Public data members

DefaultProtocol

static const char* DefaultProtocol;

Points to a string that specifies the name of the protocol the client needs. The default protocol is "StdFileEditing," which is the name of the object linking and embedding protocol.

See also TClipboard::QueryCreate

Public member functions

CloseClipboard

void CloseClipboard()

If the Clipboard is closed (*IsOpen* is false), closes the Clipboard. Closing the Clipboard allows other applications to access the Clipboard.

See also TClipboard::OpenClipboard

CountClipboardFormats

int CountClipboardFormats() const;

Returns a count of the number of types of data formats the Clipboard can use.

See also TClipboard::RegisterClipboardFormat

EmptyClipboard

bool EmptyClipboard();

Clears the Clipboard and frees any handles to the Clipboard's data. Returns true if the Clipboard is empty, or false if an error occurs.

GetClipboard

static TClipboard& GetClipboard()

Returns a reference to the *TClipboard* object.

GetClipboardData

HANDLE GetClipboardData(uint format) const;

Retrieves data from the Clipboard in the format specified by *format*. The following formats are supported:

Value	Meaning
CF_BITMAP	Data is in a bitmap format.
CF_DIB	Data is memory.
CF_DIF	Data is in a Data Interchange Format (DIF).
CF_DSPMETAFILEPICT	Data is in a metafile format.
CF_DSPTEXT	Data is in a text format.
CF_METAFILEPICT	Data is in a metafile structure.
CF_OEMTEXT	Data is an array of text characters in OEM character set.
CF_OWNERDISPLAT	Data is in a special format that the application must display.
CF_PALETTE	Data is in a color palette format.
CF_PENDATA	Data is used for pen format.
CF_RIFF	Data is in Resource Interchange File Format (RIFF).
CF_SYLK	Data is in symbolic Link format (SYLK).
CF_TEXT	Data is stored as an array of text characters.
CF_TIFF	Data is in Tag Image File Format (TIFF).
CF_WAVE	Data is in a sound wave format.

See also TClipboard::SetClipboardData

GetClipboardFormatName

int GetClipboardFormatName(uint format, char far* formatName, int maxCount) const;

Retrieves the name of the registered format specified by *format* and copies the format to the buffer pointed to by *formatName*. *maxCount* specifies the maximum length of the name of the format. If the name is longer than *maxCount*, it is truncated.

See also TClipboard::CountClipboardFormats

GetClipboardOwner

HWND GetClipboardOwner() const;

Retrieves the handle of the window that currently owns the Clipboard, otherwise returns NULL.

GetClipboardViewer

HWND GetClipboardViewer() const;

Retrieves the handle of the first window in the Clipboard-view chain. Returns NULL if there is no viewer.

See also TClipboard::SetClipboardViewer

GetOpenClipboardWindow

HWND GetOpenClipboardWindow() const;

Retrieves the handle of the window that currently has the Clipboard open. If the Clipboard is not open, the return value is false. Once the Clipboard is opened, applications cannot modify the data.

GetPriorityClipboardFormat

int GetPriorityClipboardFormat(uint FAR * priorityList, int count) const;

Returns the first Clipboard format in a list. *priorityList* points to an array that contains a list of the Clipboard formats arranged in order of priority. See *GetClipboardData* for the clipboard formats.

See also TClipboard::GetClipboardData

IsClipboardFormatAvailable

bool IsClipboardFormatAvailable(uint format) const;

Indicates if the format specified in *format* exists for use in the Clipboard. See *GetClipBoardData* for a description of clipboard data formats.

The following code tests if the clipboard can support the specified formats:

```
void
TBmpViewWindow::CePaste(TCommandEnabler& ce)
{
  TClipboard& clipboard = OpenClipboard();
  ce.Enable(
    clipboard && (
      clipboard.IsClipboardFormatAvailable(CF_METAFILEPICT) ||
      clipboard.IsClipboardFormatAvailable(CF_DIB) ||
      clipboard.IsClipboardFormatAvailable(CF_BITMAP)
    )
  );
  clipboard.CloseClipboard();
```

See also TClipboard::GetClipboardData

OpenClipboard

HWND GetOpenClipboardWindow() const;

Retrieves the handle of the window that currently has the Clipboard open. If the Clipboard is not open, the return value is false. Once the Clipboard is opened, applications cannot modify the data.

See also TClipboard::CloseClipboard

bool

operator bool() const;

Checks handle. Should use *IsOk* instead.

QueryCreate

bool QueryCreate(const char far* protocol = DefaultProtocol, OLEOPT_RENDER renderopt = olerender_draw, OLECLIPFORMAT format = 0);

QueryCreate determines if the object on the Clipboard supports the specified protocol and rendering options. *DefaultProtocol* points to a string specifying the name of the

protocol the client application needs to use. *renderopt* specifies the client application's display and printing preference for the Clipboard object. *renderopt* is set to *olerender_draw*, which tells the client library to obtain and manage the data presentation. *format* specifies the Clipboard format the client application requests. The macros _OLE_H or _INC_OLE must be defined before this function can be used.

See also TClipboard::QueryLink

QueryLink

```
bool QueryLink(const char far* protocol = DefaultProtocol, OLEOPT_RENDER renderopt = olerender_draw,
  OLECLIPFORMAT format = 0);
```

QueryLink determines if a client application can use the Clipboard data to produce a linked object that uses the specified protocol and rendering options. See *TClipboard::QueryCreate* for a description of the parameters. The macros _OLE_H or _INC_OLE must be defined before this function can be used.

See also TClipboard::QueryCreate

RegisterClipboardFormat

```
uint RegisterClipboardFormat(const char far* formatName) const;
```

Registers a new Clipboard format. *formatName* points to a character string that identifies the new format. If the format can be registered, the return value indicates the registered format. If the format can't be registered, the return value is 0. Once the format is registered, it can be used as a valid format in which to render the data.

See also TClipboard::CountClipboardFormats, TClipboard::GetClipboardFormatName

SetClipboardData

```
HANDLE SetClipboardData(uint format, HANDLE handle);
```

Sets a handle to the block of data at the location indicated by *handle*. *format* specifies the format of the data block. The clipboard must have been opened before the data handle is set. format can be any one of the valid clipboard formats (for example, CF_BITMAP or CF_DIB). See *GetClipboardData* for a list of these formats. handle is a handle to the memory location where the data data is stored. If successful, the return value is a handle to the data; if an error occurs, the return value is 0. Before the window is updated with the clipboard data, the clipboard must be closed.

See also TClipboard::GetClipboardData

SetClipboardViewer

```
HWND SetClipboardViewer(HWND Wnd) const;
```

Adds the window specified by *Wnd* to the chain of windows that WM_DRAWCLIPBOARD notifies whenever the contents of the Clipboard change.

See also TClipboard::GetClipboardViewer

Protected data members

IsOpen

```
bool IsOpen;
```

Returns true if the Clipboard is open.

TheClipboard
static TClipboard TheClipboard;
Holds the current clipboard.

Protected constructor

Protected constructor
TClipboard();
Constructs a *TClipboard* object.

TClipboardViewer Class clipview.h

TClipboardViewer is a mix-in class that registers a *TClipboardViewer* as a Clipboard viewer when the user interface element is created, and removes itself from the Clipboard-viewer chain when it is destroyed.

Protected data member

HWndNext
HWND HWndNext;
Specifies the next window in the Clipboard-viewer chain.

Protected constructors

Constructors

Form 1 TClipboardViewer();
Constructs a *TClipboardViewer* object.

Form 2 TClipboardViewer(HWND hWnd, TModule* module = 0);
Constructs a *TClipboardViewer* object with a handle (*hWnd*) to the windows that will receive notification when the Clipboard's contents are changed.

Protected member functions

DoChangeCBChain
TEventStatus DoChangeCBChain(HWND hWndRemoved, HWND hWndNext);
Tests to see if the Clipboard has changed and, if so, *DoChangeCBChain* forwards this message.

DoDestroy
TEventStatus DoDestroy();
Removes the window from the Clipboard-viewer chain.

DoDrawClipboard
TEventStatus DoDrawClipboard();
Handles *EvDrawClipboard* messages.

EvChangeCBChain

void EvChangeCBChain(HWND hWndRemoved, HWND hWndNext);

Responds to a WM_CHANGECBCHAIN message. *hWndRemoved* is a handle to the window that's being removed. *hWndNext* is the window following the removed window.

EvDestroy

void EvDestroy();

Responds to a WM_DESTROY message when a window is removed from the Clipboard-viewer chain.

EvDrawClipboard

void EvDrawClipboard();

Responds to a WM_DRAWCLIPBOARD message sent to the window in the Clipboard-viewer chain when the contents of the Clipboard change.

SetupWindow

void SetupWindow();

Adds a window to the Clipboard-viewer chain.

See also TWindow::SetupWindow

Response table entries

Response table entry	Member function
EV_WM_CHANGECBCHAIN	EvChangeCbChain
EV_WM_DESTROY	EvDestroy
EV_WM_DRAWCLIPBOARD	EvDrawClipBoard

TColor Class

color.h

TColor is a support class used in conjunction with the classes *TPalette*, *TPaletteEntry*, *TRgbQuad*, and TRgbTriple to simplify all color operations. *TColor* has ten static data members representing the standard RGB COLORREF values, from *Black* to *White*. Constructors are provided to create *TColor* objects from COLORREF and RGB values, palette indexes, palette entries, and RGBQUAD and RGBTRIPLE values.

See the entries for *NBits* and *NColors* for a description of *TColor*-related functions.

Public constructors

Constructors

Form 1 TColor();

The default constructor sets *Value* to 0.

Form 2 TColor(COLORREF value);

Creates a *TColor* object with *Value* set to the given value.

Form 3 **TColor(long value);**
Creates a *TColor* object with *Value* set to the value defined in COLORREF.

Form 4 **TColor(int r, int g, int b);**
Creates a *TColor* object with *Value* set to RGB(*r,g,b*).

Form 5 **TColor(int r, int g, int b, int f);**
Creates a *TColor* object with *Value* set to RGB(*r,g,b*) with the flag byte formed from *f*.

Form 6 **TColor(int index);**
Creates a *TColor* object with *Value* set to *PALETTEINDEX(index)*.

Form 7 **TColor(const PALETTEENTRY far& pe);**
Creates a *TColor* object with *Value* set to
`RGB(pe.peRed, pe.peGreen, pe.peBlue)`

Form 8 **TColor(const RGBQUAD far& q);**
Creates a *TColor* object with *Value* set to
`RGB(q.rgbRed, q.rgbGreen, q.rgbBlue)`

Form 9 **TColor(const RGBTRIPLE far& t);**
Creates a *TColor* object with *Value* set to
`RGB(t.rgbtRed, t.rgbtGreen, t.rgbtBlue)`

See also COLORREF typedef, PALETTEENTRY struct, RGBQUAD struct, RGBTRIPLE struct, TColor::Value

Public data members

Black
static const TColor Black;
The static *TColor* object with fixed *Value* set by RGB(0, 0, 0).

Gray
static const TColor Gray;
Contains the static *TColor* object with fixed *Value* set by RGB(128, 128, 128).

LtBlue
static const TColor LtBlue;
Contains the static *TColor* object with the fixed *Value* set by RGB(0, 0, 255).

LtCyan
static const TColor LtCyan;
Contains the static *TColor* object with the fixed *Value* set by RGB(0, 255, 255).

LtGray
static const TColor LtGray;
Contains the static *TColor* object with the fixed *Value* set by RGB(192, 192, 192).

LtGreen
static const TColor LtGreen;
Contains the static *TColor* object with the fixed *Value* set by RGB(0, 255, 0).

LtMagenta
static const TColor LtMagenta;
Contains the static *TColor* object with the fixed *Value* set by RGB(255, 0, 255).

LtRed
static const TColor LtRed;
Contains the static *TColor* object with the fixed *Value* set by RGB(255, 0, 0).

LtYellow
static const TColor LtYellow;
Contains the static *TColor* object with the fixed *Value* set by RGB(255, 255, 0).

White
static const TColor White;
Contains the static *TColor* object with the fixed *Value* set by RGB(255, 255, 255).

Public member functions

Blue
uint8 Blue() const;
Returns the blue component of this color's *Value*.

See also TColor::Red, TColor::Green, COLORREF typedef

Flags
uint8 Flags() const;
Returns the *peFlags* value of this object's *Value*.

See also TPaletteEntry

GetSysColor
static TColor GetSysColor(int uiElement);
(Presentation Manager only) Returns the color of the given *uiElement*.

Green
uint8 Green() const;
Returns the green component of this color's *Value*.

See also TColor::Red, TColor::Blue, COLORREF typedef

operator ==
bool operator ==(const TColor& clrVal);
Returns true if this color's *Value* equals *clrValue*; otherwise returns false.

See also TColor::Value

operator COLORREF()
operator COLORREF() const;
Type-conversion operator that returns *Value*.

See also TColor::Value

Index

int Index() const;

Returns the index value corresponding to this color's *Value* by masking out the two upper bytes. Used when color is a palette index value.

See also TColor::Value, COLORREF typedef

PalIndex

TColor PalIndex() const;

Returns the palette index corresponding to this color's *Value*. The returned color has the high-order byte set to 1.

See also TColor::Value, TColor::Index, COLORREF typedef

PalRelative

TColor PalRelative() const;

Returns the palette-relative RGB corresponding to this color's *Value*. The returned color has the high-order byte set to 2.

See also TColor::Value, TColor::Rgb, COLORREF typedef

Red

uint8 Red() const;

Returns the red component of this color's *Value*.

See also

TColor::Blue, TColor::Green

Rgb

TColor Rgb() const;

Returns the explicit RGB color corresponding to this color's *Value* by masking out the high-order byte.

See also TColor::Value, COLORREF typedef

SetSysColors

static bool SetSysColors(unsigned nelems, const int uiElementIndices[], const TColor colors[]);

(Presentation Manager only) Sets groups of UI element colors. *nelems* indicates the number of element colors to change (and the size of the array parameters, *uiElementIndices* indicates which elements to change, and *colors* indicates what color to change the corresponding element to. Returns **true** if successful.

Protected data member

Value

COLORREF Value;

The color value of this *TColor* object. *Value* can have three different forms, depending on the application:

- Explicit values for RGB (red, green, blue)
- An index into a logical color palette

- A palette-relative RGB value

See also COLORREF typedef

TComboBox class

combobox.h

You can use *TComboBox* to create a combo box or a combo box control in a parent *TWindow*, or to facilitate communication between your application and the combo box controls of *TDialog*. *TComboBox* objects inherit most of their behavior from *TListBox*. This class is streamable.

There are three types of combo boxes: simple, drop down, and drop down list. These types are governed by the style constants CBS_SIMPLE, CBS_DROPDOWN, and CBS_DROPDOWNLIST. These constants, supplied to the constructor of a *TComboBox*, indicate the type of combo box element to create.

Public constructors

Constructors

Form 1 **TComboBox(TWindow* parent, int id, int x, int y, int w, int h, uint32 style, uint textLen, TModule* module = 0);**

Constructs a combo box object with the specified parent window (*parent*), control ID (*Id*), position (*x, y*) relative to the origin of the parent window's client area, width (*w*), height (*h*), style (*style*), and text length (*textLen*).

Invokes the *TListBox* constructor with similar parameters. Then sets *Attr.Style* as follows:

```
Attr.Style = WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP | CBS_SORT | CBS_AUTOHSCROLL |
 WS_VSCROLL | style;
```

One of the following combo box style constants must be among the styles set in *style*: CBS_SIMPLE, CBS_DROPDOWN, CBS_DROPDOWNLIST, CBS_OWNERDRAWFIXED, or CBS_OWNERDRAWVARIABLE.

Form 2 **TComboBox(TWindow* parent, int ResourceId, uint textLen = 0, TModule* module = 0);**

Constructs a default combo box with the given parent window control ID text length.

See also TComboBox::GetTextLen, TListBox::TListBox

Public data member

TextLen

uint TextLen;

Contains the length of the text in the combo box's associated edit control.

Public member functions

AddString

virtual int AddString(const char far* string);

Adds a string to an associated list part of a combo box. Returns the index of the string in the list. The first entry is at index zero. Returns a negative value if an error occurs.

Clear

```
void Clear();
```

Clears the text of the associated edit control.

ClearList

```
virtual void ClearList();
```

Clears out all associated entries in the associated list.

DeleteString

```
virtual int DeleteString(int index);
```

Deletes the string at the passed index in the associated list part of a combo box. Returns a count of the entries remaining in the list or a negative value if an error occurs.

DirectoryList

```
virtual int DirectoryList(uint attrs, const char far* fileSpec);
```

Fills the combo box with file names from a specified directory.

FindString

```
virtual int FindString(const char far* find, int indexStart) const;
```

Searches for a match beginning at the passed Index. If a match is not found after the last string has been compared, the search continues from the beginning of the list until a match is found or until the list has been completely traversed. Returns the index of the first string in the associated list part of a combo box or a negative value if an error occurs.

GetCount

```
virtual int GetCount() const;
```

Returns the number of entries in the associated list part of the combo box or a negative value if an error occurs.

GetDroppedControlRect

```
void GetDroppedControlRect(TRect& Rect) const;
```

For combo boxes, gets the screen coordinates of the dropped down list box.

GetDroppedState

```
bool GetDroppedState() const;
```

For drop down combo boxes, determines if a list box is visible.

GetEditSel

```
int GetEditSel(int &startPos, int &endPos);
```

Returns the starting and ending positions of the text selected in the associated edit control. Returns CB_ERR if the combo box has no edit control.

GetExtendedUI

```
bool GetExtendedUI() const;
```

Determines if the combo box has the extended user interface, which differs from the default user interface in the following ways:

- Displays the list box if the user clicks the static text field.

- Displays the list box if the user presses the *Down* key.
- Disables scrolling in the static text field if the item list is not visible.

Returns **true** if the combo box has the extended user interface; otherwise returns **false**.

See also TComboBox::SetExtendedUI

GetItemData

virtual uint32 GetItemData(int index) const;

Returns the 32-bit value associated with the combo box's item.

See also TListBox::GetItemData

GetItemHeight

int GetItemHeight(int index) const;

Returns the height in pixels of the combo box's list items. If an error occurs, returns a negative value.

See also TComboBox::GetItemData, TListBox::GetItemData

GetSelIndex

virtual int GetSelIndex() const;

Returns the index of the list selection or a negative value if none exists.

GetString

virtual int GetString(char far* str, int index) const;

Retrieves the contents of the string at the position supplied in *index* and returns it in *string*. *GetString* returns the string length or a negative value if an error occurs. The buffer must be large enough for the string and the terminating zero.

See also TListBox::GetString

GetStringLen

virtual int GetStringLen(int index) const;

Returns the string length (excluding the terminating zero) of the item at the position index supplied in *index*. Returns a negative value if an error occurs.

See also TListBox::GetStringLen

GetText

int GetText(char far* str, int maxChars) const;

Retrieves the number of characters in the edit or static portion of the combo box.

GetTextLen

int GetTextLen() const;

Returns the text length (excluding the terminating zero) of the edit control or static portion of the combo box.

HideList

void HideList();

Hides the drop down list of a drop down or drop down list combo box.

InsertString

virtual int InsertString(const char far* str, int index);

Inserts a string in the associated list part of a combo box at the position supplied in *Index*. Returns the index of the string in the list or a negative value if an error occurs.

See also TListBox::InsertString

SetEditSel

int SetEditSel(int startPos, int endPos);

Selects characters that are between *startPos* and *endPos* in the edit control of the combo box. Returns CB_ERR if the combo box does not have an edit control.

SetExtendedUI

int SetExtendedUI(bool extended);

If the combo box has the extended user interface, sets the extended user interface.

See also TComboBox::GetExtendedUI

SetItemData

virtual int SetItemData(int index, uint32 data);

Sets the 32-bit value associated with the *TComboBox's* item. If an error occurs, returns a negative value.

SetItemHeight

int SetItemHeight(int index, int height);

Sets the height of the list items or the edit control portion in a combo box. If the index or height is invalid, returns a negative value.

See also TComboBox::GetItemHeight

SetSelIndex

virtual int SetSelIndex(int index);

Selects a string of characters in a combo box. *index* specifies the index of the string of characters in the list box to select. If the index is 0, the first line in the list box is selected. If the index is –1, the current selection is removed. If an error occurs, a negative value is returned.

See also TComboBox::GetSelIndex

SetSelString

virtual int SetSelString(const char far* findStr, int indexStart);

Selects a string of characters in the associated list box and sets the contents of the associated edit control to the supplied string.

SetText

void SetText(const char far* string);

Selects the first string in the associated list box that begins with the supplied *string*. If there is no match, *SetText* sets the contents of the associated edit control to the supplied string and selects it.

ShowList

Form 1 void ShowList();

Shows the list of a drop down or drop down list combo box.

Form 2 void ShowList(bool show);

Shows or hides the drop down or drop down list combo box depending on the value of show. If show is **true**, shows the list; if show is **false**, hides the list.

See also TComboBox::HideList

Transfer

uint Transfer(void* buffer, TTransferDirection direction);

Transfers the items and selection of the combo box to or from a transfer buffer if *tdSetData* or *tdGetData*, respectively, is passed as the *direction*. *buffer* is expected to point to a *TComboBoxData* structure.

Transfer returns the size of a pointer to a *TComboBoxData*. To retrieve the size without transferring data, your application must pass *tdSizeData* as the *direction*.

You must use a pointer in your transfer buffer to these structures. You cannot embed copies of the structures in your transfer buffer, and you cannot use these structures as transfer buffers.

See also TComboBoxData, TWindow::Transfer, Window::SetupWindow

Protected member functions

GetClassName

virtual char far* GetClassName();

Returns the name of *TComboBox's* registration class, "ComboBox."

SetupWindow

void SetupWindow();

Sets up the window and limits the amount of text the user can enter in the combo box's edit control to the value of *TextLen* minus 1.

TComboBoxData class

combobox.h

An interface object that represents a transfer buffer for a *TComboBox*.

Public constructor and destructor

Constructor

TComboBoxData();

Constructs a *TComboBoxData* object, initializes *Strings* and *ItemDatas* to empty arrays, and initializes *Selection* and *SelIndex* to 0.

Destructor

~TComboBoxData();

Deletes *Strings*, *ItemDatas*, and *Selection*.

Public member functions

AddString

```
void AddString(const char *str, bool isSelected = false);
```

Adds the specified string to the array of Strings. If *IsSelected* is true, *AddString* deletes *Selection* and copies *string* into *Selection*.

AddStringItem

```
void AddStringItem(const char* str, uint32 itemData, bool isSelected = false);
```

Calls *AddItemData* to add the item data to the *ItemDatas* array, and calls *AddString* to add a string to the array of *Strings*.

Clear

```
void Clear();
```

Flushes the *Strings* and *ItemDatas* members. Resets the index and selected string values.

GetItemDatas

```
TDwordArray& GetItemDatas();
```

Returns the array of DWORDs to transfer to or from a combo box's associated list box.

GetSelCount

```
int GetSelCount() const;
```

Returns the number of items selected, either 0 or 1.

GetSelection

```
string& GetSelection();
```

Returns the currently selected string (the *Selection* data member) to transfer to or from a combo box.

GetSelIndex

```
int GetSelIndex();
```

Returns the index (the *SelIndex* data member) of the selected item in the strings array.

GetSelString

```
void GetSelString(char far* buffer, int bufferSize) const;
```

Copies the selected string into a buffer of the specified size. *bufferSize* includes the terminating 0.

GetSelStringLength

```
int GetSelStringLength() const;
```

Returns length of the currently selected string excluding the terminating 0.

GetStrings

```
TStringArray& GetStrings();
```

Returns the array of strings (the *Strings* data member) to transfer to or from a combo box's associated list box.

ResetSelections

```
void ResetSelections();
```

Resets the index of the selected item and the currently selected string.

Select
void Select(int index);
Selects the item at the given index.

SelectString
void SelectString(const char far* str);
Selects the selection string (*str*) and makes the matching *String* entry (if one exists) as selected.

Protected data members

ItemDatas
TDwordArray ItemDatas;
Array of DWORDs to transfer to or from a combo box's associated list box.

Selection
string Selection;
The currently selected string to transfer to or from a combo box.

SelIndex
int SelIndex;
Index of the selected item in the strings array.

Strings
TStringArray Strings;
Array of class string to transfer to or from a combo box's associated list box.

TCommandEnabler class

window.h

An abstract base class used for automatic enabling and disabling of commands, *TCommandEnabler* is a class from which you can derive other classes, each one having its own command enabler. For example, *TButtonGadgetEnabler* is a derived class that's a command enabler for button gadgets, and *TMenuItemEnabler* is a derived class that's a command enabler for menu items. Although your derived classes are likely to use only the functions *Enable*, *SetCheck*, and *GetHandled*, all of *TCommandEnabler's* functions are described so that you can better understand how ObjectWindows uses command processing. The following paragraphs explain the dynamics of command processing.

Handling command messages
Commands are messages of the windows WM_COMMAND type that have associated command identifiers (for example, CM_FILEMENU). When the user selects an item from a menu or a toolbar, when a control sends a notification message to its parent window, or when an accelerator keystroke is translated, a WM_COMMAND message is sent to a window.

Responding to command messages

A command is handled differently depending on which type of command a window receives. Menu items and accelerator commands are handled by adding a command entry to a message response table using the EV_COMMAND macro. The entry requires two arguments:

- A command identifier (for example, CM_LISTUNDO)
- A member function (for example, *CMEditUndo*)

Child ID notifications, messages that a child window sends to its parent window, are handling by using one of the notification macros defined in the header file windowev.h.

It is also possible to handle a child ID notification at the child window by adding an entry to the child's message response table using the EV_NOTIFY_AT_CHILD macro. This entry requires the following arguments:

- A notification message (for example, LBN_DBLCLK)
- A member function (for example, *CmEditItem*)

TWindow command processing

One of the classes designed to handle command processing, *TWindow* performs basic command processing according to these steps:

1. The member function *WindowProc* calls the virtual member function *EvCommand*.
2. *EvCommand* checks to see if the window has requested handling the command by looking up the command in the message response table.
3. If the window has requested handling the command identifier by using the EV_COMMAND macro, the command is dispatched.

TWindow also handles Child ID notifications at the child window level.

TFrameWindow command processing

TFrameWindow provides specialized command processing by overriding its member function *EvCommand* and sending the command down the command chain (that is, the chain of windows from the focus window back up to the frame itself, the original receiver of the command message).

If no window in the command chain handles the command, *TFrameWindow* delegates the command to the application object . Although this last step is theoretically performed by the frame window, it is actually done by *TWindow's* member function, *DefaultProcessing*.

Invoking EvCommand

When *TFrameWindow* sends a command down the command chain, it doesn't directly dispatch the command; instead, it invokes the window's *EvCommand* member function.This procedure gives the windows in the command chain the flexibility to handle a command by overriding the member function *EvCommand* instead of being limited to handling only the commands requested by the EV_COMMAND macro.

Handling command enable messages

Most applications expend considerable energy updating menu items and tool bar buttons to provide the necessary feedback indicating that a command has been enabled. In order to simplify this procedure, ObjectWindows lets the event handler that's going to handle the command make the decision about whether or not to enable or disable a command.

Although the WM_COMMAND_ENABLE message is sent down the same command chain as the WM_COMMAND event; exactly when the WM_COMMAND_ENABLE message is sent depends on the type of command enabling that needs to be processed.

Command enabling for menu items

TFrameWindow performs this type of command enabling when it receives a WM_INITMENUPOPUP message. It sends this message before a menu list appears. ObjectWindows then identifies the menu commands using the command IDs and sends requests for the commands to be enabled.

Note that because Object Windows actively maintains toolbars and menu items, any changes made to the variables involved in the command enabling functions are implemented dynamically and not just when a window is activated.

Command enabling for toolbar buttons

The type of command enabling is performed during idle processing (in the *IdleAction* function). See the diagram following the description of *TWindow::DefaultProcessing* for a graphical illustration of this process.

Creating specialized command enablers

Associated with the WM_COMMAND_ENABLE message is an object of the *TCommandEnabler* type. This family of command enablers includes specialized command enablers for menu items and toolbar buttons.

As you can see from *TCommandEnable's* class declaration, you can do considerably more than simply enable or disable a command using the command enabler. For example, you have the ability to change the text associated with the command as well as the state of the command.

Using the EV_COMMAND_ENABLE macro

You can use the EV_COMMAND_ENABLE macro to handle WM_COMMAND_ENABLE messages. Just as you do with the EV_COMMAND macro, you specify the command identifier that you want to handle and the member function you want to invoke to handle the message.

Automatically enabling and disabling commands

ObjectWindows simplifies enabling and disabling of commands by automatically disabling commands for which there are no associated handlers. *TFrameWindow*'s member function *EvCommandEnable* performs this operation, which involves completing a two pass algorithm.

1 The first pass sends a WM_COMMAND_ENABLE message down the command chain giving each window an explicit opportunity to do the command enabling.

2 If no window handles the command enabling request, then ObjectWindows checks to see whether any windows in the command chain are going to handle the command through any associated EV_COMMAND entries in their response tables. If there is a command handler in one of the response tables, then the command is enabled; otherwise it is disabled.

Because of this implicit command enabling or disabling, you do not need to (and actually should not) do explicit command enabling unless you want to change the command text, change the command state, or conditionally enable or disable the command.

If you handle commands indirectly by overriding the member function *EvCommand* instead of using the EV_COMMAND macro to add a response table entry, then ObjectWindows will not be aware that you are handling the command. Consequently, the command may be automatically disabled. Should this occur, the appropriate action to take is to also override the member function *EvCommandEnable* and explicitly enable the command.

Public constructor

Constructor

TCommandEnabler(uint id, HWND hWndReceiver = 0);

Constructs the *TCommandEnable* object with the specified command ID. Sets the message responder (*hWndReceiver*) to zero.

Type definitions

CheckState

enum CheckState{Unchecked, Checked, Indeterminate};

Enumerates the values for the check state of the command sender. This state applies to buttons, such as those used for tool bars or to control bar gadgets.

Public data members

Id

const uint Id;

Command ID for the enabled command.

Public member functions

Enable

virtual void Enable(bool enable = true);

Enables or disables the command sender. When *Enable* is called, it sets the *Handled* flag.

GetHandled

bool GetHandled();

Returns *Handled*, a flag value that shows if this command enabler has been handled, in which case *Handled* is **true**.

IsReceiver

bool IsReceiver(HWND hReceiver);

Returns **true** if *receiver* is the same as the message responder originally set up in the constructor.

SetCheck

virtual void SetCheck(int check) = 0;

Changes the *check* state of the command sender to either unchecked, checked, or indeterminate. This state applies to buttons, such as those used for toolbars or to control bar gadgets.

SetText

virtual void SetText(const char far* text) = 0;

Changes the text associated with a command sender. This applies, for example, to text associated with a menu item or text on a button.

Protected data members

Handled

bool Handled;

Is **true** if the command enabler has been handled.

HWndReceiver

const HWND HWndReceiver;

The message responder (the window) that receives the command.

TCommonDialog class

commdial.h

Derived from *TDialog*, *TCommonDialog* is the abstract base class for *TCommonDialog* objects. It provides the basic functionality for creating dialog boxes using the common dialog DLL. The ObjectWindows classes that inherit this common dialog functionality include

TChooseColorDialog—a modal dialog box that lets a user select colors for an application

TChooseFontDialog—a modal dialog box that lets a user select fonts for an application

TReplaceDialog—a modeless dialog box that lets a user specify a text selection to replace

TFindDialog—a modeless dialog box that lets a user specify a text selection to find

TFileOpenDialog—a modal dialog box that lets a user specify a file to open

TFileSaveDialog—a modal dialog box that lets a user specify a file to save

TPrintDialog—a modal dialog box that lets a user specify printer options

Each common dialog class uses a nested class, *TData*, that stores the attributes and user-specified data. For example, the *TChooseColorDialog::TData* class stores the color attributes the user selects in response to a prompt in a common dialog box. In fact, to create a common dialog box, you construct a *TData* object first, then fill in the data members of the *TData* object before you even construct the common dialog box object. Finally, you either execute a modal dialog box or create a modeless dialog box.

Public constructor

Constructor

```
TCommonDialog(TWindow* parent, const char far* title = 0, TModule* module = 0);
```

Invokes a *TWindow* constructor, passing the parent window *parent* and constructs a common dialog box which you can modify to suit your specifications. You can indicate the module ID (*title*) and window caption (*title*), which otherwise default to 0.

Public member functions

DoCreate

```
HWND DoCreate();
```

Called by *Create*, *DoCreate* creates a modeless dialog box. It returns 0 if unsuccessful.

See also TDialog::Create

DoExecute

```
int DoExecute();
```

Called by *Execute*, *DoExecute* executes a modal dialog box. If the dialog box execution is canceled or unsuccessful, *DoExecute* returns IDCANCEL.

See also TDialog::Execute

Protected data member

CDTitle

```
const char far* CDTitle;
```

CDTitle stores the optional caption displayed in the common dialog box.

See also TDialog::SetCaption

Protected member functions

CmHelp

```
void CmHelp();
```

Default handler for the *pshHelp* push button (the Help button in the dialog box).

CmOkCancel

```
void CmOkCancel();
```

Responds to a click on the dialog box's OK or Cancel button by calling *DefaultProcessing* to let the common dialog DLL process the command.

See also TDialog::CmCancel, TDialog::CmOk

EvClose

void EvClose();
Responds to a WM_CLOSE message by calling *DefaultProcessing* to let the common dialog DLL process the command.

See also TDialog::EvClose

SetupWindow

void SetupWindow();
Assigns the caption of the dialog box to *CDTitle* if *CDTitle* is nonzero.

See also TDialog::SetupWindow

Response table entries

Response table entry	Member function
EV_COMMAND(IDCANCEL, CmOkCancel)	CmOkCancel
EV_COMMAND(IDOK, CmOkCancel)	CmOkCancel
EV_WM_CLOSE	EvClose
EV_WM_CTLCOLOR	EvCtlColor

TCondFunc typedef

window.h

typedef bool (*TCondFunc) (TWindow *win, void *param);
Defines a member function type used by *TWindow's* function *FirstThat*.

See also TWindow::FirstThat

TCondMemFunc typedef

window.h

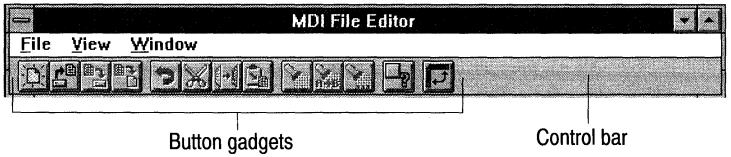
typedef bool (TWindow_TCondMemFunc) (*win, void *param);
Defines a member function type used by *TWindow's* function *FirstThat*.

See also TWindow::FirstThat

TControlBar class

controlb.h

Derived from *TGadgetWindow*, *TControlBar* implements a control bar that provides mnemonic access for its button gadgets. The sample MDIFILE.CPP ObjectWindows program on your distribution disk displays the following example of a control bar.



To construct, build, and insert a control bar into a frame window, you can first define the following response table:

```
DEFINE_RESPONSE_TABLE1(TMDIFileApp, TApplication
   EV_COMMAND(CM_FILENEW, CmFileNew),
   EV_COMMAND(CM_FILEOPEN, CmFileOpen),
   EV_COMMAND(CM_SAVESTATE, CmSaveState),
   EV_COMMAND(CM_RESTORESTATE, CmRestoreState),
END_RESPONSE_TABLE;
```

Next, add statements that will construct a main window and load its menu, accelerator table, and icon. Then, to construct, build and insert a control bar into the frame window, insert these statements:

```
TControlBar* cb = new TControlBar(frame);
cb->Insert(*new TButtonGadget(CM_FILENEW, CM_FILENEW));
cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN));
cb->Insert(*new TButtonGadget(CM_FILESAVE, CM_FILESAVE));
cb->Insert(*new TSeparatorGadget(6));
cb->Insert(*new TButtonGadget(CM_EDITCUT, CM_EDITCUT));
cb->Insert(*new TButtonGadget(CM_EDITCOPY, CM_EDITCOPY));
cb->Insert(*new TButtonGadget(CM_EDITPASTE, CM_EDITPASTE));
cb->Insert(*new TSeparatorGadget(6));
cb->Insert(*new TButtonGadget(CM_EDITUNDO, CM_EDITUNDO));
frame->Insert(*cb, TDecoratedFrame::Top);
```

Public constructor

Constructor

TControlBar(TWindow* parent = 0, TTileDirection direction = Horizontal, TFont* font = new TGadgetWindowFont, TModule* module = 0);

Constructs a *TControlBar* interface object with the specified direction (either horizontal or vertical) and window font.

Public member function

PreProcessMsg

bool PreProcessMsg(MSG& msg);

Preprocesses messages. Because *PreProcessMsg* does not translate any accelerator keys for *TControlBar*, it returns false.

Protected member function

PositionGadget
void PositionGadget(TGadget* previous, TGadget* next, TPoint& p);
Gets the border style, determines the direction of the gadget, and positions the button gadget on either a horizontal or vertical border if any overlapping is required.

TControlGadget class

controlg.h

TControlGadget serves as a surrogate for *TControl* so that you can place *TControl* objects such as edit controls, buttons, sliders, gauges, or third-party controls, into a gadget window. If necessary, *TControlGadget* sets a parent window and creates the control gadget. See *TGadget* for more information about gadget objects.

Public constructor and destructor

Constructor
TControlGadget(TWindow& control, TBorderStyle = None);
Creates a *TControlGadget* object associated with the specified *TControl* window.

Destructor
~TControlGadget();
Destroys a *TControlGadget* object and removes it from the associated window.

Protected data member

Control
TWindow* Control;
Points to the control window that is managed by this *TControlGadget*.

Protected member functions

GetDesiredSize
void GetDesiredSize(TSize& size);
Calls *TGadget::GetDesiredSize* and passes the size of the control gadget. Use *GetDesiredSize* to find the size the control gadget needs to be in order to accommodate the borders and margins as well as the highest and widest control gadget.

See also TGadget::GetDesiredSize

GetInnerRect
void GetInnerRect(TRect&);
Computes the area of the control gadget's rectangle excluding the borders and margins.

Inserted
void Inserted();
Called when the control gadget is inserted in the parent window. Displays the window in its current size and position.

Invalidate
void Invalidate(bool erase = true);
Used to invalidate the active (usually nonborder) portion of the control gadget, *Invalidate* calls *InvalidateRect* and passes the boundary width and height of the area to erase.

InvalidateRect
void InvalidateRect(const TRect&, bool erase = true);
Invalidates the control gadget rectangle in the parent window.

Removed
void Removed();
Called when the control gadget is removed from the parent window.

SetBounds
void SetBounds(TRect& rect);
Calls *TGadget::SetBounds* and passes the dimensions of the control gadget's rectangle. *SetBounds* informs the control gadget of a change in its bounding rectangle.

See also TGadget::SetBounds

Update
void Update();
Updates the client area of the specified window by immediately sending a WM_PAINT message.

Response table entries

The *TControlGadget* class has no response table entries.

TControl class

control.h

TControl unifies its derived control classes, such as *TScrollBar*, *TControlGadget*, and *TButton*. Control objects of derived classes are used to represent control interface elements. A control object must be used to create a control in a parent *TWindow* or a derived window. A control object can be used to facilitate communication between your application and the controls of a *TDialog*. *TControl* is a streamable class.

Public constructors

Constructors
Form 1 TControl(TWindow* parent, int id, const char far* title, int x, int y, int w, int h, TModule* module = 0);

Invokes *TWindow's* constructor, passing it parent (parent window), *title* (caption text), and *module*. Sets the control attributes using the supplied library ID (*Id*), position (*x, y*) relative to the origin of the parent window's client area, width (*w*), and height (*h*) parameters. It sets the control style to WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP.

Form 2 `TControl(TWindow* parent, int resourceId, TModule* module = 0);`
Constructs an object to be associated with an interface control of a *TDialog*. Invokes the *TWindow* constructor then enables the data transfer mechanism by calling *EnableTransfer*.

The *resourceId* parameter must correspond to a control interface resource that you define.

See also TWindow::TWindow, TWindow::EnableTransfer

Protected member functions

CompareItem

`virtual int CompareItem (COMPAREITEMSTRUCT far& compareInfo);`
Also used with owner-draw buttons and list boxes, *CompareItem* compares two items. The derived class supplies the compare logic.

See also
COMPAREITEMSTRUCT struct

DeleteItem

`virtual void DeleteItem (DELETEITEMSTRUCT far& deleteInfo);`
DeleteItem is used with owner-draw buttons and list boxes. In such cases, the derived class supplies the delete logic.

See also DELETEITEMSTRUCT struct

DrawItem

`virtual void DrawItem(DRAWITEMSTRUCT far& drawInfo);`
DrawItem responds to a message forwarded to a drawable control by *TWindow* when the control needs to be drawn. *TControl::DrawItem* calls *ODADrawEntire* if the entire control needs to be drawn, calls *ODASelect* if the selection state of the control has changed, or calls *ODAFocus* if the focus has been shifted to or from the control.

See also TControl::ODADrawEntire, TControl::ODASelect, TControl::ODAFocus, DRAWITEMSTRUCT struct

EvCompareItem

`LRESULT EvCompareItem(uint ctrlId, COMPAREITEMSTRUCT far& comp);`
Handles a WM_COMPAREITEM message for owner-draw controls.

See also COMPAREITEMSTRUCT struct

EvDeleteItem

`void EvDeleteItem(uint ctrlId, DELETEITEMSTRUCT far& del);`
Handles a WM_DELETEITEM message for owner-draw controls.

See also DELETEITEMSTRUCT struct

EvDrawItem

void EvDrawItem(uint ctrlId, DRAWITEMSTRUCT far& draw);

Handles a WM_DRAWITEM message.

See also DRAWITEMSTRUCT struct

EvMeasureItem

void EvMeasureItem(uint ctrlId, MEASUREITEMSTRUCT far& meas);

Handles a WM_MEASUREITEM message.

See also MEASUREITEMSTRUCT struct

EvPaint

void EvPaint();

If the control has a predefined class, *EvPaint* calls *TWindow::DefaultProcesing* for painting. Otherwise, it calls *TWindow::EvPaint*.

See also TWindow::DefaultProcessing

MeasureItem

virtual void MeasureItem (MEASUREITEMSTRUCT far& measureInfo);

Used by owner-drawn controls to set the dimensions of the specified item. For list boxes and control boxes, this function applies to specific items; for other owner-drawn controls, this function is used to set the total size of the control.

See also MEASUREITEMSTRUCT struct

ODADrawEntire

virtual void ODADrawEntire(DRAWITEMSTRUCT far& drawInfo);

Responds to a notification message sent to a drawable control when the control needs to be drawn. *ODADrawEntire* can be redefined by a drawable control to specify the manner in which it is to be drawn.

See also TControl::DrawItem, DRAWITEMSTRUCT struct

ODAFocus

virtual void ODAFocus(DRAWITEMSTRUCT far& drawInfo);

Responds to a notification sent to a drawable control when the focus has shifted to or from the control. *ODAFocus* can be redefined by a drawable control to specify the manner in which it is to be drawn when losing or gaining the focus.

See also TControl::DrawItem, DRAWITEMSTRUCT struct

ODASelect

virtual void ODASelect(DRAWITEMSTRUCT far& drawInfo);

Responds to a notification sent to a drawable control when the selection state of the control changes. *ODASelect* can be redefined by a drawable control to specify the manner in which it is drawn when its selection state changes.

See also TControl::DrawItem, DRAWITEMSTRUCT struct

Response table entries

Response table entry	Member function
EV_WM_PAINT	EvPaint
EV_WM_COMPAREITEM	EvCompareItem
EV_WM_DELETEITEM	EvDeleteItem
EV_WM_DRAWITEM	EvDrawItem
EV_WM_MEASUREITEM	EvMeasureItem

TCreatedDC class

dc.h

An abstract *TDC* class, *TCreatedDC* serves as the base for DCs that are created and deleted.

See TDC for more information about DC objects.

Public constructors and destructor

Constructors

Form 1 `TCreatedDC(const char far* driver, const char far* device, const char far* output, const DEVMODE far* initData=0);`

Creates a DC object for the device specified by *driver* (driver name), *device* (device name), and *output* (the name of the file or device [port] for the physical output medium). The optional *initData* argument provides a DEVMODE structure containing device-specific initialization data for this DC. *initData* must be 0 (the default) if the device is to use any default initializations specified by the user.

Form 2 `TCreatedDC(HDC handle TAutoDelete autoDelete);`

Creates a DC object using an existing DC.

Destructor

`~TCreatedDC();`

Calls *RestoreObjects* clears any nonzero *OrgXXX* data members. If *ShouldDelete* is **true** the destructor deletes this DC.

See also enum TDC::TAutoDelete, TDC::RestoreObjects, TDC::ShouldDelete, DEVMODE struct

Protected constructor

Constructor

`TCreatedDC();`

Creates a device context for the given device. DC objects can be constructed either by borrowing an existing HDC handle or by supplying device and driver information.

TCursor class

gdiobjec.h

TCursor, derived from *TGdiBase*, represents the GDI cursor object class. *TCursor* constructors can create cursors from a resource or from explicit information. Because cursors are not real GDI objects, the *TCursor* destructor overrides the base destructor, *~TGdiBase*.

Public constructors and destructor

Constructors

Form 1 TCursor(HCURSOR handle, TAutoDelete autoDelete = NoAutoDelete);
Creates a *TCursor* object and sets the *Handle* data member to the given borrowed handle. The *ShouldDelete* data member defaults to **false**, ensuring that the borrowed handle will not be deleted when the C++ object is destroyed.

Form 2 TCursor(HINSTANCE instance, const TCursor& cursor);
Creates a copy of the given *cursor* object. The 32-bit version (for compiling a Win32 application) uses *CopyIcon*() and does a cast to get to HICON.

Form 3 TCursor(HINSTANCE instance, ResId resId);
Constructs a *cursor* object from the specified resource ID.

Form 4 TCursor(HINSTANCE instance, const TPoint& hotSpot, const TSize& size, void far* andBits, void far* xorBits);
Constructs a *TCursor* object of the specified size at the specified point.

Form 5 TCursor(const void* resBits, uint32 resSize);
32 bit only. Constructs a *TCursor* object from the specified resource.

Form 6 TCursor(const ICONINFO* iconInfo);
32 bit only. Creates a *TCursor* object from the specified *ICONINFO* structure information.

Destructor

~TCursor();
Destroys a *TCursor* object.

See also ~TGdiObject, TGdiObject::Handle, TGdiObject::ShouldDelete, TPoint, TSize, ICONINFO struct

Public member function

GetIconInfo

bool GetIconInfo(ICONINFO* iconInfo) const;
32-bit only. Retrieves information about this icon and copies it in the given *ICONINFO* structure. Returns **true** if the call is successful; otherwise returns **false**.

See also ICONINFO struct

Operators

operator HCURSOR()
operator HCURSOR() const;
An inline typecasting operator. Converts this cursor's *Handle* to type HCURSOR (the data type representing the handle to a cursor resource).

operator ==
bool operator ==(const TCursor& other) const;
Returns **true** if this cursor equals *other*; otherwise returns **false**.

TDC class

dc.h

TDC is the root class for GDI DC wrappers. Each *TDC* object inherits a *Handle* from *TGdiBase* and casts that handle to an HDC using the HDC operator. Win API functions that take an HDC argument can therefore be called by a corresponding *TDC* member function without this explicit handle argument.

DC objects can be created directly with *TDC* constructors, or via the constructors of specialized subclasses (such as *TWindowDC*, *TMemoryDC*, *TMetaFileDC*, *TDibDC*, and *TPrintDC*) to get specific behavior. DC objects can be constructed with an already existing and borrowed HDC handle or from scratch by supplying device driver information, as with *::CreateDC*. The class *TCreateDC* takes over much of the creation and deletion work from *TDC*.

TDC has four handles as protected data members: *OrgBrush*, *OrgPen*, *OrgFont*, and *OrgPalette*. These handles keep track of the stock GDI objects selected into each DC. As new GDI objects are selected with *SelectObject* or *SelectPalette*, these data members store the previous objects. The latter can be restored individually with *RestoreBrush*, *RestorePen*, and so on, or they can all be restored with *RestoreObjects*. When a *TDC* object is destroyed (via *~TDC::TDC*), all the originally selected objects are restored. The data member *TDC::ShouldDelete* controls the deletion of the TDC object.

Public constructor and destructor

Constructor
TDC(HDC handle);
Creates a DC object "borrowing" the handle of an existing DC. The *Handle* data member is set to the given *handle* argument.

Destructor
virtual ~TDC();
Calls *RestoreObjects*.

See also TCreatedDC, TDC::RestoreObjects, TDC::ShouldDelete

Public member functions

AngleArc

bool AngleArc(int x, int y, uint32 radius, float startAngle, float sweepAngle);
bool AngleArc(const TPoint& center, uint32 radius, float startAngle, float sweepAngle);

32-bit only. Draws a line segment and an arc on this DC using the currently selected pen object. The line is drawn from the current position to the beginning of the arc. The arc is that part of the circle (with the center at logical coordinates (*x*, *y*) and positive radius, *radius*) starting at *startAngle* and ending at (*startAngle* + *sweepAngle*). Both angles are measured in degrees, counterclockwise from the x-axis (the default arc direction). The arc might appear to be elliptical, depending on the current transformation and mapping mode. *AngleArc* returns **true** if the figure is drawn successfully; otherwise, it returns **false**. If successful, the current position is moved to the end point of the arc.

See also TDC::Arc, TPoint class

Arc

bool Arc(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4);
bool Arc(const TRect& r, const TPoint& start, const TPoint& end);

Draws an elliptical arc on this DC using the currently selected pen object. The center of the arc is the center of the bounding rectangle, specified either by (*x1*, *y1*)/(*x2*, *y2*) or by the rectangle r. The starting/ending points of the arc are specified either by (*x3*, *y3*)/(*x4*, *y4*) or by the points *start* and *end*. All points are specified in logical coordinates. *Arc* returns **true** if the arc is drawn successfully; otherwise, it returns **false**. The current position is neither used nor altered by this call. The drawing direction default is counterclockwise.

See also TDC::AngleArc, TPoint, TRect

BeginPath

bool BeginPath();

32-bit only. Opens a new path bracket for this DC and discards any previous paths from this DC. Once a path bracket is open, an application can start calling draw functions on this DC to define the points that lie within that path. The draw functions that define points in a path are the following *TDC* members: *AngleArc*, *Arc*, *Chord*, *CloseFigure*, *Ellipse*, *ExtTextOut*, *LineTo*, *MoveToEx*, *Pie*, *PolyBezier*, *PolyBezierTo*, *PolyDraw*, *Polygon*, *Polyline*, *PolylineTo*, *PolyPolygon*, *PolyPolyline*, *Rectangle*, *RoundRect*, and *TextOut*.

A path bracket can be closed by calling *TDC::EndPath*.

BeginPath returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::FillPath, TDC::EndPath, TDC::PathToRegion, TDC::StrokePath, TDC::StrokeandFillPath, TDC::WidenPath

BitBlt

bool BitBlt(int dstX, int dstY, int w, int h, const TDC& srcDC, int srcX, int srcY, uint32 rop=SRCCOPY);
bool BitBlt(const TRect& dst, const TDC& srcDC, const TPoint& src, uint32 rop=SRCCOPY);

Performs a bit-block transfer from *srcDc* (the given source DC) to this DC (the destination DC). Color bits are copied from a source rectangle to a destination rectangle. The location of the source rectangle is specified either by its upper left-corner logical coordinates (*srcX*, *srcY*), or by the *TPoint* object, *src*. The destination rectangle can be

specified either by its upper left-corner logical coordinates (*dstX*, *dstY*), width *w*, and height *h*, or by the *TRect* object, *dst*. The destination rectangle has the same width and height as the source. The *rop* argument specifies the raster operation used to combine the color data for each pixel. See *TDC::MaskBlt* for a detailed list of *rop* codes.

When recording an enhanced metafile, an error occurs if the source DC identifies the enhanced metafile DC.

See also TPoint class, TRect class

Chord

```
bool Chord(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4);
bool Chord(const TRect& r, const TPoint& start, const TPoint& end);
```

Draws and fills a chord (a region bounded by the intersection of an ellipse and a line segment) on this DC using the currently selected pen and brush objects. The ellipse is specified by a bounding rectangle given either by (*x1*, *y1*)/(*x2*, *y2*) or by the rectangle *R*. The starting/ending points of the chord are specified either by (*x3*, *y3*)/(*x4*, *y4*) or by the points *Start* and *End*. *Chord* returns **true** if the call is successful; otherwise, it returns **false**. The current position is neither used nor altered by this call.

See also TDC::Arc, TPoint class, TRect class

CloseFigure

```
bool CloseFigure();
```

32-bit only. Closes an open figure in this DC's open path bracket by drawing a line from the current position to the first point of the figure (usually the point specified by the most recent *TDC::MoveTo* call), and connecting the lines using the current join style for this DC. If you close a figure with *TDC::LineTo* instead of with *CloseFigure*, end caps (instead of a join) are used to create the corner. The call fails if there is no open path bracket on this DC. Any line or curve added to the path after a *CloseFigure* call starts a new figure. A figure in a path remains open until it is explicitly closed with *CloseFigure* even if its current position and start point happen to coincide.

CloseFigure returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::BeginPath, TDC::EndPath

DPtoLP

```
bool DPtoLP(TPoint* points, int count = 1) const;
```

Converts each of the *count* points in the *points* array from device points to logical points. The conversion depends on this DC's current mapping mode and the settings of its window and viewport origins and extents. *DPtoLP* returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::LPtoDP, TPoint class

DrawFocusRect

```
bool DrawFocusRect(int x1, int x2, int y1, int y2);
bool DrawFocusRect(const TRect& rect);
```

Draws the given rectangle on this DC in the style used to indicate focus. Calling the function a second time with the same *rect* argument will remove the rectangle from the display. A rectangle drawn with *DrawFocusRect* cannot be scolled. *DrawFocusRect* returns true if the call is successful; otherwise, it returns false.

See also TRect class

DrawIcon

```
bool DrawIcon(int x, int y, const TIcon& icon);
bool DrawIcon(const TPoint& point, const TIcon& icon);
```

Draws the given *icon* on this DC. The upper left corner of the drawn icon can be specified by x- and y-coordinates or by the *point* argument. *DrawIcon* returns **true** if the call is successful; otherwise, it returns **false**.

See also TIcon class

DrawText

```
virtual bool DrawText(const char far* string, int count, const TRect& r, uint16 format = 0);
```

Formats and draws in the given rectangle, *r*, up to *count* characters of the null-terminated *string* using the current font for this DC. If *count* is –1, the whole string is written. The rectangle must be specified in logical units. Formatting is controlled with the *format* argument, which can be various combinations of the following values:

Value	Meaning
DT_BOTTOM	Specifies bottom-justified text. This value must be combined (bitwise OR'd) with DT_SINGLELINE.
DT_CALCRECT	Determines the width and height of the rectangle. If there are multiple lines of text, *DrawText* uses the width of *r* (the rectangle argument) and extends the base of the rectangle to bound the last line of text. If there is only one line of text, *DrawText* uses a modified value for the right side of *r* so that it bounds the last character in the line. In both cases, *DrawText* returns the height of the formatted text but does not draw the text.
DT_CENTER	Centers text horizontally.
DT_EXPANDTABS	Expands tab characters. The default number of characters per tab is eight.
DT_EXTERNALLEADING	Includes the font external leading in line height. Normally, external leading is not included in the height of a line of text.
DT_LEFT	Aligns text flush-left.
DT_NOCLIP	Draws without clipping. *DrawText* is somewhat faster when DT_NOCLIP is used.
DT_NOPREFIX	Turns off processing of prefix characters. Normally, *DrawText* interprets the prefix character **&** as a directive to underscore the character that follows, and the prefix characters **&&** as a directive to print a single **&**. By specifying DT_NOPREFIX, this processing is turned off.
DT_RIGHT	Aligns text flush-right.
DT_SINGLELINE	Specifies single line only. Carriage returns and linefeeds do not break the line.
DT_TABSTOP	Sets tab stops. Bits 15–8 (the high-order byte of the low-order word) of the *format* argument are the number of characters for each tab. The default number of characters per tab is eight.
DT_TOP	Specifies top-justified text (single line only).
DT_VCENTER	Specifies vertically centered text (single line only).
DT_WORDBREAK	Specifies word breaking. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by *r*. A carriage return/line sequence will also break the line.

Note that the DT_CALCRECT, DT_EXTERNALLEADING, DT_INTERNAL, DT_NOCLIP, and DT_NOPREFIX values cannot be used with the DT_TABSTOP value.

DrawText uses this DC's currently selected font, text color, and background color to draw the text. Unless the DT_NOCLIP format is used, *DrawText* clips the text so that it does not appear outside the given rectangle. All formatting is assumed to have multiple lines unless the DT_SINGLELINE format is given.

If the selected font is too large for the specified rectangle, *DrawText* does not attempt to substitute a smaller font.

If successful, *DrawText* returns **true**; otherwise, returns **false**.

See also TDC::GrayString, TDC::TabbedTextOut, TDC::TextOut, TRect class

Ellipse

bool Ellipse(int x1, int y1, int x2, int y2);
bool Ellipse(const TPoint& p1, const TPoint& p2);
bool Ellipse(const TPoint& point, const TSize& size);
bool Ellipse(const TRect& rect);

Draws and fills an ellipse on this DC using the currently selected pen and brush objects. The center of the ellipse is the center of the bounding rectangle specified either by (*x1, y1*)/(*x2, y2*) or by the *rect* argument. *Ellipse* returns **true** if the call is successful; otherwise, it returns **false**. The current position is neither used nor altered by this call.

See also TDC::Arc, TPoint class, TRect class, TSize class

EndPath

bool EndPath();

32-bit only. Closes the path bracket and selects the path it defines into this DC. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::BeginPath, TDC::CloseFigure

EnumFontFamilies

int EnumFontFamilies(const char far* family, FONTENUMPROC proc, void far* data) const;

Enumerates the fonts available to this DC in the font family specified by *family*. The given application-defined callback *proc* is called for each font in the family or until *proc* returns 0, and is defined as

```
typedef int (CALLBACK* FONTENUMPROC)(CONST LOGFONT *, CONST TEXTMETRIC *, DWORD, LPARAM);
```

data lets you pass both application-specific data and font data to *proc*. If successful, the call returns the last value returned by *proc*.

See also TDC::EnumFonts, LOGFONT struct, TEXTMETRIC struct

EnumFonts

int EnumFonts(const char far* faceName, OLDFONTENUMPROC callback, void far* data) const;

Enumerates the fonts available on this DC for the given *faceName*. The font type, LOGFONT, and TEXTMETRIC data retrieved for each available font is passed to the user-defined *callback* function together with any optional, user-supplied data placed in the *data* buffer. The *callback* function can process this data in any way desired. Enumeration continues until there are no more fonts or until the *callback* function returns 0. If *faceName* is 0, *EnumFonts* randomly selects and enumerates one font of each available typeface. *EnumFonts* returns the last value returned by the *callback* function. Note that OLDFONTENUMPROC is defined as FONTENUMPROC for Win32 only.

FONTENUMPROC is a pointer to a user-defined function and has the following prototype:

```
int CALLBACK EnumFontsProc(LOGFONT *lplf, TEXTMETRIC *lptm, uint32 dwType, LPARAM lpData);
```

where *dwType* specifies one of the following font types: DEVICE_FONTTYPE, RASTER_FONTTYPE, or TRUETYPE_FONTTYPE.

See also TDC::EnumFontFamilies, LOGFONT struct, TEXTMETRIC struct

EnumMetaFile

int EnumMetaFile(const TMetaFilePict& metaFile, MFENUMPROC callback, void* data) const;

Enumerates the GDI calls within the given *metaFile*. Each such call is retrieved and passed to the given *callback* function, together with any client data from *data*, until all calls have been processed or a callback function returns 0. *MFENUMPROC* is defined as

```
typedef int (CALLBACK* MFENUMPROC)(HDC, HANDLETABLE FAR*, METARECORD FAR*, int, LPARAM);
```

See also TDC::PlayMetaFile, METARECORD struct

EnumObjects

int EnumObjects(uint objectType, GOBJENUMPROC proc, void far* data) const;

Enumerates the pen or brush objects available for this DC. The parameter *objectType* can be either OBJ_BRUSH or OBJ_PEN. For each pen or brush found, *proc*, a user-defined *callback* function, is called until there are no more objects found or the callback function returns 0. *proc* is defined as

```
typedef int (CALLBACK* GOBJENUMPROC)(LPVOID, LPARAM);
```

Parameter *data* specifies an application-defined value that is passed to *proc*.

ExcludeClipRect

int ExcludeClipRect(const TRect& rect);

Creates a new clipping region for this DC. This new region consists of the current clipping region minus the given rectangle, *rect*. The return value indicates the new clipping region's type as follows:

Region	Meaning
COMPLEXREGION	Clipping Region has overlapping borders.
ERROR	Invalid DC.
NULLREGION	Clipping region is empty.
SIMPLEREGION	Clipping region has no overlapping borders.

See also TDC::GetClipBox, TRect class

ExcludeUpdateRgn

int ExcludeUpdateRgn(HWND wnd);

Prevents drawing within invalid areas of a window by excluding an updated region of this DC's window from its clipping region. The return value indicates the resulting clipping region's type as follows:

Region	Meaning
COMPLEXREGION	Clipping Region has overlapping borders.
ERROR	Invalid DC.
NULLREGION	Clipping region is empty.
SIMPLEREGION	Clipping region has no overlapping borders.

ExtFloodFill

```
bool ExtFloodFill(const TPoint& point, TColor color, uint16 fillType);
```

Fills an area on this DC starting at *point* and using the currently selected brush object. The *color* argument specifies the color of the boundary or of the region to be filled. The *fillType* argument specifies the type of fill, as follows:

FLOODFILLBORDER	The fill region is bounded by the given *color*. This style coincides with the filling method used by *FloodFill*.
FLOODFILLSURFACE	The fill region is defined by the given *color*. Filling continues outward in all directions as long as this color is encountered. Use this style when filling regions with multicolored borders.

Not every device supports *ExtFloodFill*, so applications should test first with *TDC::GetDeviceCaps*.

ExtFloodFill returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::FloodFill, TDC::GetDeviceCaps, TColor class, TPoint class

ExtTextOut

```
virtual bool ExtTextOut(int x, int y, uint16 options, const TRect* r, const char far* string, int count, const
  int far* dx = 0);
bool ExtTextOut(const TPoint& p, uint16 options, const TRect* r, const char far* string, int count, const
  int far* dx = 0);
```

Draws up to *count* characters of the given null-terminated *string* in the current font on this DC. If *count* is –1, the whole string is written.

An optional rectangle *r* can be specified for clipping, opaquing, or both, as determined by the *options* value. If *options* is set to ETO_CLIPPED, the rectangle is used for clipping the drawn text. If *options* is set to ETO_OPAQUE, the current background color is used to fill the rectangle. Both options can be used if ETO_CLIPPED is OR'd with ETO_OPAQUE.

The (*x*, *y*) o r*p* arguments specify the logical coordinates of the reference point that is used to align the first character. The current text-alignment mode can be inspected with *TDC::GetTextAlign* and changed with *TDC::SetTextAlign*. By default, the current position is neither used nor altered by *ExtTextOut*. However, if the align mode is set to TA_UPDATECP, *ExtTextOut* ignores the reference point argument(s) and uses or updates the current position as the reference point.

The *dx* argument is an optional array of values used to set the distances between the origins (upper left corners) of adjacent character cells. For example, *dx[i]* represents the number of logical units separating the origins of character cells *i* and *i+1*. If *dx* is 0, *ExtTextOut* uses the default inter-character spacings.

ExtTextOut returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::TextOut, TDC::GetTextAlign, TDC::TabbedTextOut, TPoint, TRect

FillPath

```
bool FillPath();
```

32-bit only. Closes any open figures in the current path of this DC and fills the path's interior using the current brush and polygon fill mode. After filling the interior, *FillPath* discards the path from this DC.

FillPath returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::BeginPath, TDC::CloseFigure, TDC::StrokePath, TDC::StrokeAndFillPath, TDC::SetPolyFillMode

FillRect

```
bool FillRect(int x1, int y1, int x2, int y2, const TBrush& brush);
bool FillRect(const TRect& rect, const TBrush& brush);
```

Fills the given rectangle on this DC using the specified brush. The fill covers the left and top borders but excludes the right and bottom borders. *FillRect* returns **true** if the call is successful; otherwise, it returns **false**.

See also TBrush, TRect

FillRgn

```
bool FillRgn(const TRegion& region, const TBrush& brush);
```

Fills the given *region* on this DC using the specified *brush*. *FillRgn* returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::InvertRgn, TDC::PaintRgn, TBrush class, TRegion class

FlattenPath

```
bool FlattenPath();
```

32-bit only. Transforms any curves in the currently selected path of this DC. All such curves are changed to sequences of linear segments. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::WidenPath, TDC::BeginPath

FloodFill

```
bool FloodFill(const TPoint& point, TColor color);
```

Fills an area on this DC starting at *point* and using the currently selected brush object. The *color* argument specifies the color of the boundary or of the area to be filled. Returns true if the call is successful; otherwise, returns false. *FloodFill* is maintained in the WIN32 API for compatibility with earlier APIs. New WIN32 applications should use *TDC::ExtFloodFill*.

See also TDC::ExtFloodFill, TColor, TPoint

FrameRect

```
bool FrameRect(int x1, int x2, int y1, int y2, const TBrush& brush);
bool FrameRect(const TRect& rect, const TBrush& brush);
```

Draws a border on this DC around the given rectangle, *rect*, using the given brush, *brush*. The height and width of the border is one logical unit. Returns **true** if the call is successful; otherwise, it returns **false**.

See also TBrush class, TRect class

FrameRgn

```
bool FrameRgn(const TRegion& region, const TBrush& brush, const TPoint& p);
```

Draws a border on this DC around the given region, *region*, using the given brush, *brush*. The width and height of the border is specified by the *p* argument. Returns **true** if the call is successful; otherwise, returns **false**.

See also TBrush class, TRegion class

GetAspectRatioFilter

```
bool GetAspectRatioFilter(TSize& size) const;
```

Retrieves the setting of the current aspect-ratio filter for this DC.

See also TSize

GetBkColor

```
TColor GetBkColor() const;
```

Returns the current background color of this DC.

See also TDC::SetBkColor, TColor class

GetBkMode

```
int GetBkMode() const;
```

Returns the background mode of this DC, either OPAQUE or TRANSPARENT.

See also TDC::SetBkMode

GetBoundsRect

```
bool GetBoundsRect(TRect& bounds, uint16 flags) const;
```

Reports in *bounds* the current accumulated bounding rectangle of this DC or of the Windows manager, depending on the value of *flags*. Returns **true** if the call is successful; otherwise, returns **false.**

The *flags* argument can be DCB_RESET or DCB_WINDOWMGR or both. The *flags* value work as follows:

DCB_RESET	Forces the bounding rectangle to be cleared after being set in bounds.
DCB_WINDOWMGR	Reports the Windows current bounding rectangle rather than that of this DC.

There are two bounding-rectangle accumulations, one for Windows and one for the application. *GetBoundsRect* returns screen coordinates for the Windows bounds, and logical units for the application bounds. The Windows accumulated bounds can be queried by an application but not altered. The application can both query and alter the DC's accumulated bounds.

See also TDC::SetBoundsRect, TRect class

GetBrushOrg

bool GetBrushOrg(TPoint& point) const;

Places in *point* the current brush origin of this DC. Returns **true** if successful; otherwise, returns false.

See also TPoint class

GetCharABCWidths

bool GetCharABCWidths(uint firstChar, uint lastChar, ABC* abc);

Retrieves the widths of consecutive characters in the range *firstChar* to *lastChar* from the current TrueType font of this DC. The widths are reported in the array *abc* of *ABC* structures. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::GetCharWidth, ABC struct

GetCharWidth

bool GetCharWidth(uint firstChar, uint lastChar, int* buffer);

Retrieves the widths in logical units for a consecutive sequence of characters in the current font for this DC. The sequence is specified by the inclusive range, *firstChar* to *lastChar*, and the widths are copied to the given *buffer*. If a character in the range is not represented in the current font, the width of the default character is assigned. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::GetCharABCWidths

GetClipBox

int GetClipBox(TRect& rect) const;
TRect GetClipBox() const;

Places the current clip box size of this DC in *rect*. The clip box is defined as the smallest rectangle bounding the current clipping boundary. The return value indicates the clipping region's type as follows:

Region	Meaning
COMPLEXREGION	Clipping Region has overlapping borders.
ERROR	Invalid DC.
NULLREGION	Clipping region is empty.
SIMPLEREGION	Clipping region has no overlapping borders.

See also TDC::ExcludeClipRect, TRect class

GetClipRgn

bool GetClipRgn(TRegion& region) const;

Retrieves this DC's current clip-region and, if successful, places a copy of it in the *region* argument. You can alter this copy without affecting the current clip-region. Returns **true** if the call is successful; otherwise, returns **false**.

See also TRegion

GetCurrentObject

HANDLE GetCurrentObject(uint objectType) const;

Returns a handle to the currently selected object of the given *objectType* associated with this DC. Returns 0 if the call fails. *objectType* can be OBJ_PEN, OBJ_BRUSH, OBJ_PAL, OBJ_FONT, or OBJ_BITMAP.

GetCurrentPosition

```
bool GetCurrentPosition(TPoint& point) const;
```

Reports in *point* the logical coordinates of this DC's current position. Returns **true** if the call is successful; otherwise, returns **false**.

See also TPoint class

GetDCOrg

```
bool GetDCOrg(TPoint& point) const;
```

Obtains the final translation origin for this device context and places the value in *point*. This value specifies the offset used to translate device coordinates to client coordinates for points in an application window. Returns **true** if the call is successful; otherwise, returns **false**.

See also TPoint class

GetDeviceCaps

```
virtual int GetDeviceCaps(int index) const;
```

Used under WIN3.1 or later, *GetDeviceCaps* returns capability information about this DC. The *index* argument specifies the type of information required.

GetDIBits

```
bool GetDIBits(const TBitmap& bitmap, uint16 startScan, uint16 numScans, void HUGE* bits,
  const BITMAPINFO far& info, uint16 usage);
bool GetDIBits(const TBitmap& bitmap, TDib& dib);
```

The first version retrieves some or all of the bits from the given *bitmap* on this DC and copies them to the *bits* buffer using the DIB (device-independent bitmap) format specified by the BITMAPINFO argument, *info*. *numScan* scanlines of the bitmap are retrieved, starting at scan line *startScan*. The *usage* argument determines the format of the *bmiColors* member of the BITMAPINFO structure, according to the following table:

Value	Meaning
DIB_PAL_COLORS	The color table is an array of 16-bit indexes into the current logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.
DIB_PAL_INDICES	There is no color table for this bitmap. The DIB bits consist of indexes into the system palette. No color translation occurs. Only the BITMAPINFOHEADER portion of BITMAPINFO is filled in.

In the second version of *GetDIBits*, the bits are retrieved from *bitmap* and placed in the *dib.Bits* data member of the given *TDib* argument. The BITMAPINFO argument is supplied from *dib.info*.

GetDIBits returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::SetDIBits, TBitmap, TDib, BITMAPINFO struct

GetFontData

```
uint32 GetFontData(uint32 table, uint32 offset, void* buffer, long data);
```

Retrieves font-metric information from a scalable TrueType font file (specified by *table* and starting at *offset* into this table) and places it in the given *buffer*. *data* specifies the size in bytes of the data to be retrieved. If the call is successful, it returns the number of bytes set in *buffer*; otherwise, –1 is returned.

GetGlyphOutline

uint32 GetGlyphOutline (uint chr, uint format, GLYPHMETRICS far& gm, uint32 buffSize, voidfar* buffer, const MAT2 far& mat2);

Retrieves TrueType metric and other data for the given character, *chr*, on this DC and places it in *gm* and *buffer*. The *format* argument specifies the format of the retrieved data as indicated in the following table. (A value of 0 simply fills in the *GLYPHMETRICS* structure but does not return glyph-outline data.)

Value	Meaning
1	Retrieves the glyph bitmap.
2	Retrieves the curve data points in the rasterizer's native format and uses the font's design units. With this value of format, the *mat2* transformation argument is ignored.

The *gm* argument specifies the GLYPHMETRICS structure that describes the placement of the glyph in the character cell. *buffSize* specifies the size of buffer that receives data about the outline character. If either *buffSize* or *buffer* is 0, *GetGlyphOutline* returns the required buffer size. Applications can rotate characters retrieved in bitmap format (format = 1) by specifying a 2 x 2 transformation matrix via the *mat2* argument.

GetGlyphOutline returns a positive number if the call is successful; otherwise, it returns GDI_ERROR.

See also TDC::GetOutlineTextMetrics, GLYPHMETRICS struct

GetKerningPairs

int GetKerningPairs(int pairs, KERNINGPAIR far* krnPair);

Retrieves kerning pairs for the current font of this DC up to the number specified in *pairs* and copies them into the *krnPair* array of KERNINGPAIR structures. If successful, the function returns the actual number of pairs retrieved. If the font has more than *pairs* kerning pairs, the call fails and returns 0. The *krnPair* array must allow for at least *pairs* KERNINGPAIR structures. If *krnPair* is set to 0, *GetKerningPairs* returns the total number of kerning pairs for the current font.

See also KERNINGPAIR struct

GetMapMode

int GetMapMode() const;

If successful, returns the current window mapping mode of this DC; otherwise, returns 0. The mapping mode defines how logical coordinates are mapped to device

coordinates. It also controls the orientation of the device's x- and y-axes. The mode values are shown in the following table:

Value	Meaning
MM_ANISOTROPIC	Logical units are mapped to arbitrary units with arbitrarily scaled axes. *SetWindowExtEx* and *SetViewportExtEx* must be used to specify the desired units, orientation, and scaling.
MM_HIENGLISH	Each logical unit is mapped to 0.001 inch. Positive x is to the right; positive y is at the top.
MM_HIMETRIC	Each logical unit is mapped to 0.01 millimeter. Positive x is to the right; positive y is at the top.
MM_ISOTROPIC	Logical units are mapped to arbitrary units with equally scaled axes; that is, one unit along the x-axis is equal to one unit along the y-axis. *SetWindowExtEx* and *SetViewportExtEx* must be used to specify the desired units and the orientation of the axes. GDI makes adjustments as necessary to ensure that the x and y units remain the same size (e.g., if you set the window extent, the viewport is adjusted to keep the units isotropic).
MM_LOENGLISH	Each logical unit is mapped to 0.01 inch. Positive x is to the right; positive y is at the top.
MM_LOMETRIC	Each logical unit is mapped to 0.1 millimeter. Positive x is to the right; positive y is at the top.
MM_TEXT	Each logical unit is mapped to one device pixel. Positive x is to the right; positive y is at the bottom.
MM_TWIPS	Each logical unit is mapped to one twentieth of a printer's point (1/1440 inch). Positive x is to the right; positive y is at the top.

See also TDC::SetMapMode

GetNearestColor

```
TColor GetNearestColor(TColor Color) const;
```

Returns the color nearest to the given *Color* argument for the current palette of this DC.

See also TColor

GetOutlineTextMetrics

```
uint32 GetOutlineTextMetrics(uint data, OUTLINETEXTMETRIC far& otm);
uint16 GetOutlineTextMetrics(uint data, OUTLINETEXTMETRIC far& otm);
```

Retrieves metric information for TrueType fonts on this DC and copies it to the given array of OUTLINETEXTMETRIC structures, *otm*. This structure contains a TEXTMETRIC and several other metric members, as well as four string-pointer members for holding family, face, style, and full font names. Since memory must be allocated for these variable-length strings in addition to the font metric data, you must pass (with the *data* argument) the total number of bytes required for the retrieved data. If *GetOutlineTextMetrics* is called with *otm* = 0, the function returns the total buffer size required. You can then assign this value to *data* in subsequent calls.

Returns nonzero if the call is successful; otherwise, returns 0.

See also TDC::GetTextMetrics, OUTLINETEXTMETRIC struct, TEXTMETRIC struct

GetPixel

```
TColor GetPixel(int x, int y) const;
TColor GetPixel(const TPoint& point) const;
```

Returns the color of the pixel at the given location.

See also TDC::SetPixel, TPoint class

GetPolyFillMode

int GetPolyFillMode() const;

Returns the current polygon-filling mode for this DC, either ALTERNATE or WINDING.

See also TDC::SetPolyFillMode

GetROP2()

int GetROP2() const;

Returns the current drawing (raster operation) mode of this DC.

See also TDC::SetROP2

GetStretchBltMode

int GetStretchBltMode() const;

Returns the current stretching mode for this DC: BLACKONWHITE, COLORONCOLOR, or WHITEONBLACK. The stretching mode determines how bitmaps are stretched or compressed by the *StretchBlt* function.

See also TDC::SetStretchBltMode, TDC::StretchBlt

GetSystemPaletteEntries

uint GetSystemPaletteEntries(int start, int num, PALETTEENTRY far* entries) const;

Retrieves a range of up to *num* palette entries, starting at *start*, from the system palette to the *entries* array of PALETTEENTRY structures. Returns the actual number of entries transferred.

See also PALETTEENTRY struct

GetSystemPaletteUse

uint GetSystemPaletteUse() const;

Determines whether this DC has access to the full system palette. Returns SYSPAL_NOSTATIC or SYSPAL_STATIC.

See also TDC::SetSystemPaletteUse

GetTabbedTextExtent

bool GetTabbedTextExtent(const char far* string, int stringLen, int numPositions, const int far* positions, TSize& size) const;

TSize GetTabbedTextExtent(const char far* string, int stringLen, int numPositions, const int far* positions) const;

Computes the extent (width and height) in logical units of the text line consisting of *stringLen* characters from the null-terminated *string*. The extent is calculated from the metrics of the current font or this DC, but ignores the current clipping region. In the first version of *GetTabbedTextExtent*, the extent is returned in *size*; in the second version, the extent is the returned *TSize* object. Width is *size.x* and width is *size.y*.

The width calculation includes the spaces implied by any tab codes in the string. Such tab codes are interpreted using the *numPositions* and *positions* arguments. The *positions* array specifies *numPositions* tab stops given in device units. The tab stops must have strictly increasing values in the array. If *numPositions* and *positions* are both 0, tabs are

expanded to eight times the average character width. If *numPositions* is 1, all tab stops are taken to be *positions*[0] apart.

If kerning is being applied, the sum of the extents of the characters in a string might not equal the extent of the string.

See also TDC::TabbedTextOut, TDC::GetTextExtent, TSize class

GetTextAlign

uint GetTextAlign() const;

If successful, returns the current text-alignment flags for this DC; otherwise, returns the value GDI_ERROR. The text-alignment flags determine how *TDC::TextOut* and *TDC::ExtTextOut* align text strings in relation to the first character's screen position. *GetTextAlign* returns certain combinations of the flags listed in the following table:

Value	Meaning	
TA_BASELINE	The reference point will be on the baseline of the text.	
TA_BOTTOM	The reference point will be on the bottom edge of the bounding rectangle.	
TA_TOP	The reference point will be on the top edge of the bounding rectangle.	
TA_CENTER	The reference point will be aligned horizontally with the center of the bounding rectangle.	
TA_LEFT	The reference point will be on the left edge of the bounding rectangle.	
TA_RIGHT	The reference point will be on the right edge of the bounding rectangle.	
TA_NOUPDATECP	The current position is not updated after each text output call.	
TA_UPDATECP	The current position is updated after each text output call.	
	When the current font has a vertical default baseline (as with Kanji) the following values replace TA_BASELINE and TA_CENTER:	
	VTA_BASELINE	The reference point will be on the baseline of the text.
	VTA_CENTER	The reference point will be aligned vertically with the center of the bounding rectangle.

The text-alignment flags are not necessarily single bit-flags and might be equal to 0. The flags must be examined in groups of the following related flags:

- TA_LEFT, TA_RIGHT, and TA_CENTER
- TA_BOTTOM, TA_TOP, and TA_BASELINE
- TA_NOUPDATECP and TA_UPDATECP

If the current font has a vertical default baseline (as with Kanji), these are groups of related flags:

- TA_LEFT, TA_RIGHT, and VTA_BASELINE
- TA_BOTTOM, TA_TOP, and VTA_CENTER
- TA_NOUPDATECP and TA_UPDATECP

To verify that a particular flag is set in the return value of this function, the application must perform the following steps:

1. Apply the bitwise OR operator to the flag and its related flags.
2. Apply the bitwise AND operator to the result and the return value.

3 Test for the equality of this result and the flag.

The following example shows a method for determining which horizontal alignment flag is set:

```
switch ((TA_LEFT | TA_RIGHT | TA_CENTER) & dc.GetTextAlign()) {
   case TA_LEFT:
   ...
   case TA_RIGHT:
   ...
   case TA_CENTER:
   ...
}
```

See also TDC::SetTextAlign, TDC::TextOut, TDC::ExtTextOut

GetTextCharacterExtra

int GetTextCharacterExtra() const;

If successful, returns the current intercharacter spacing, in logical units, for this DC; otherwise, returns INVALID_WIDTH.

See also TDC::SetTextCharacterExtra

GetTextColor

TColor GetTextColor() const;

Returns the current text color of this DC. The text color determines the color displayed by *TDC::TextOut* and *TDC::ExtTextOut*.

See also TDC::SetTextColor, TDC::TextOut, TDC::ExtTextOut, TColor

GetTextExtent

bool GetTextExtent(const char far* string, int stringLen, TSize& size);
TSize GetTextExtent(const char far* string, int stringLen);

Computes the extent (width and height) in logical units of the text line consisting of *stringLen* characters from the null-terminated *string*. The extent is calculated from the metrics of the current font or this DC, but ignores the current clipping region. In the first version of *GetTextExtent* the extent is returned in *size*; in the second version, the extent is the returned *TSize* object. Width is *size.x* and height is *size.y*.

If kerning is being applied, the sum of the extents of the characters in a string might not equal the extent of the string.

See also TSize class

GetTextFace

int GetTextFace(int count, char far* facename) const;

Retrieves the typeface name for the current font on this DC. Up to *count* characters of this name are copied to *facename*. If successful, *GetTextFace* returns the number of characters actually copied; otherwise, it returns 0.

See also TDC::GetTextAlign, TDC::GetTextMetrics

GetTextMetrics

bool GetTextMetrics(TEXTMETRIC far& metrics) const;

Fills the *metrics* structure with metrics data for the current font on this DC. Returns **true** if the call is successful; otherwise, returns **false**.

See also TEXTMETRIC struct

GetViewportExt

```
bool GetViewportExt(TSize& extent) const;
TSize GetViewportExt() const;
```

The first version retrieves this DC's current viewport's x- and y-extents (in device units) and places the values in *extent*. This version returns **true** if the call is successful; otherwise, it returns **false**. The second version returns only these x- and y-extents.

The *extent* value determines the amount of stretching or compression needed in the logical coordinate system to fit the device coordinate system. *extent* also determines the relative orientation of the two coordinate systems.

See also TDC::SetViewportExt, TSize class

GetViewportOrg

```
bool GetViewportOrg(TPoint& point) const;
TPoint GetViewportOrg() const;
```

The first version sets in the *point* argument the x- and y-extents (in device-units) of this DC's viewport. It returns **true** if the call is successful; otherwise, it returns **false**. The second version returns the x- and y-extents (in device-units) of this DC's viewport.

See also TDC::SetViewportOrg, TDC::OffsetViewportOrg, TPoint class

GetWindowExt

```
bool GetWindowExt(TSize& extent) const;
TSize GetWindowExt() const;
```

Retrieves this DC's window current x- and y-extents (in device units). The first version places the values in *extent* and returns true if the call is successful; otherwise, it returns false. The second version returns the current extent values. The *extent* value determines the amount of stretching or compression needed in the logical coordinate system to fit the device coordinate system. *extent* also determines the relative orientation of the two coordinate systems.

See also TDC::SetWindowExt, TSize class

GetWindowOrg

```
bool GetWindowOrg(TPoint& point) const;
TPoint GetWindowOrg() const;
```

Places in *point* the x- and y-coordinates of the origin of the window associated with this DC. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::SetWindowOrg, TDC::OffsetWindowOrg, TPoint class

GrayString

```
virtual bool GrayString(const TBrush& brush, GRAYSTRINGPROC outputFunc, const char far* string, int count,
  const TRect& r);
```

Draws in the given rectangle (*r*) up to *count* characters of gray text from *string* using the given brush, *brush*, and the current font for this DC. If *count* is –1 and *string* is null-terminated, the whole string is written. The rectangle must be specified in logical units.

If *brush* is 0, the text is grayed with the same brush used to draw window text on this DC. Gray text is primarily used to indicate disabled commands and menu items.

GrayString writes the selected text to a memory bitmap, grays the bitmap, then displays the result. The graying is performed regardless of the current brush and background color.

The *outputFunc* pointer to a function can specify the procedure instance of an application-supplied drawing function and is defined as

```
typedef BOOL (CALLBACK* GRAYSTRINGPROC)(HDC, LPARAM, int);
```

If *outputFunc* is 0, *GrayString* uses *TextOut* and *string* is assumed to be a normal, null-terminated character string. If *string* cannot be handled by *TextOut* (for example, the string is stored as a bitmap), you must provide a suitable drawing function via *outputFunc*.

If the device supports a solid gray color, it is possible to draw gray strings directly without using *GraySring*. Call *GetSysColor* to find the color value; for example, *G* of COLOR_GRAYTEXT. If *G* is nonzero (non-black), you can set the text color with *SetTextColor(G)* and then use any convenient text-drawing function.

GrayString returns **true** if the call is successful; otherwise, it returns **false**. Failure can result if *TextOut* or *outputFunc* return false, or if there is insufficient memory to create the bitmap.

See also TDC::TextOut, TBrush class, TRect class

operator HDC()

operator HDC() const{return Handle;}

Typecasting operator. Converts a pointer to type HDC (the data type representing the handle to a DC).

IntersectClipRect

int IntersectClipRect(const TRect& rect);

Creates a new clipping region for this DC's window by forming the intersection of the current region with the rectangle specified by *rect*. The return value indicates the resulting clipping region's type as follows:

Region	Meaning
COMPLEXREGION	Clipping Region has overlapping borders.
ERROR	Invalid DC.
NULLREGION	Clipping region is empty.
SIMPLEREGION	Clipping region has no overlapping borders.

See also TDC::GetClipBox, TRect class

InvertRect

bool InvertRect(int x1, int x2, int y1, int y2);
bool InvertRect(const TRect& rect);

Inverts the given rectangle, *rect*, on this DC. On monochrome displays, black and white pixels are interchanged. On color displays, inversion depends on how the colors are

generated for particular displays. Calling *InvertRect* an even number of times restores the original colors. *InvertRect* returns **true** if the call is successful; otherwise, it returns **false**.

See also TRect class

InvertRgn

```
bool InvertRgn(const TRegion& region);
```

Inverts the given *region*, on this DC. On monochrome displays, black and white pixels are interchanged. On color displays, inversion depends on how the colors are generated for particular displays. Calling *InvertRegion* an even number (n>=2) of times restores the original colors. Returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::PaintRgn, TDC::FillRgn, TRegion class

LineTo

```
bool LineTo(int x, int y);
bool LineTo(const TPoint& point);
```

Draws a line on this DC using the current pen object. The line is drawn from the current position up to, but not including, the given end point, which is specified by (*x*, *y*) or by *point*. If the call is successful, *LineTo* returns **true** and the current point is reset to *point*; otherwise, it returns **false**.

See also TPoint

LPtoDP

```
bool LPtoDP(TPoint* points, int count = 1) const;
```

Converts each of the *count* points in the *points* array from logical points to device points. The conversion depends on this DC's current mapping mode and the settings of its window and viewport origins and extents. Returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::DPtoLP, TPoint

MaskBlt

```
bool MaskBlt(const TRect& dst, const TDC& srcDC, const TPoint& src, const TBitmap& maskBm,
  const TPoint& maskPos, uint32 rop=SRCCOPY);
```

Copies a bitmap from the given source DC to this DC. *MaskBlt* combines the color data from source and destination bitmaps using the given mask and raster operation. The *srcDC* argument specifies the DC from which the source bitmap will be copied. The destination bitmap is given by the rectangle, *dst*. The source bitmap has the same width and height as *dst*. The *src* argument specifies the logical coordinates of the upper left corner of the source bitmap. The *maskBm* argument specifies a monochrome mask bitmap. An error will occur if *maskBm* is not monochrome. The *maskPos* argument gives the upper left corner coordinates of the mask. The raster-operation code, *rop*, specifies how the source, mask, and destination bitmaps combine to produce the new destination bitmap. The raster-operation codes are as follows:

Value of rop	Meaning
BLACKNESS	Fill *dst* with index-0 color of physical palette (default is black).
DSTINVERT	Invert *dst*.

Value of rop	Meaning
MERGECOPY	Merge the colors of source with mask with Boolean AND.
MERGEPAINT	Merge the colors of inverted-source with the colors of *dst* using Boolean OR.
NOTSRCCOPY	Copy inverted-source to *dst*.
NOTSRCERASE	Combine the colors of source and *dst* using Boolean OR, then invert result.
PATCOPY	Copy mask to *dst*.
PATINVERT	Combine the colors of mask with the colors of *dst* using Boolean XOR.
PATPAINT	Combine the colors of mask with the colors of inverted-source using Boolean OR, then combine the result with the colors of *dst* using Boolean OR.
SRCAND	Combine the colors of source and *dst* using the Boolean AND.
SRCCOPY	Copy source directly to *dst*.
SRCERASE	Combine the inverted colors of *dst* with the colors of source using Boolean AND.
SRCPAINT	Combine the colors of source and *dst* using Boolean OR.
WHITENESS	Fill *dst* with index-1 color of physical palette (default is white).

If *rop* indicates an operation that excludes the source bitmap, the *srcDC* argument must be 0. A value of 1 in the mask indicates that the destination and source pixel colors should be combined using the high-order word of *rop*. A value of 0 in the mask indicates that the destination and source pixel colors should be combined using the low-order word of *rop*. If the mask rectangle is smaller than *dst*, the mask pattern will be suitably duplicated.

When recording an enhanced metafile, an error occurs if the source DC identifies the enhanced metafile DC.

If a rotation or shear transformation is in effect for the source DC when *MaskBlt* is called, an error occurs. Other transformations are allowed. If necessary, *MaskBlt* will adjust the destination and mask color formats to match that of the source bitmaps. Before using *MaskBlt*, an application should call *GetDeviceCaps* to determine if the source and destination DCs support *MaskBlt*.

MaskBlt returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::BitBlt, TDC::PlgBlt, TDC::GetDeviceCaps, TBitmap class, TPoint class, TRect class

ModifyWorldTransform

bool ModifyWorldTransform(XFORM far& xform, uint32 mode);

Changes the current world transformation for this DC using the given *xform* and *mode* arguments. *mode* determines how the given XFORM structure is applied, as listed below.

Value	Meaning
MWT_IDENTITY	Resets the current world transformation using the identity matrix. If this mode is specified, the XFORM structure pointed to by *lpXform* is ignored.

Value	Meaning
MWT_LEFTMULTIPLY	Multiplies the current transformation by the data in the XFORM structure. (The data in the XFORM structure becomes the left multiplicand, and the data for the current transformation becomes the right multiplicand.)
MWT_RIGHTMULTIPLY	Multiplies the current transformation by the data in the XFORM structure. (The data in the XFORM structure becomes the right multiplicand, and the data for the current transformation becomes the left multiplicand.) *ModifyWorldTransform* returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::SetWorldTransform, XFORM struct

MoveTo

```
bool MoveTo(int x, int y);
bool MoveTo(const TPoint& point);
bool MoveTo(const TPoint& point, TPoint& oldPoint);
```

Moves the current position of this DC to the given x- and y-coordinates or to the given *point*. The third version sets the previous current position in *oldPoint*. Returns **true** if the call is successful; otherwise, returns **false**.

See also TPoint class

OffsetClipRgn

```
int OffsetClipRgn(const TPoint& delta);
```

Moves the clipping region of this DC by the x- and y-offsets specified in *delta*. The return value indicates the resulting clipping region's type as follows:

Region	Meaning
COMPLEXREGION	Clipping region has overlapping borders.
ERROR	Invalid DC.
NULLREGION	Clipping region is empty.
SIMPLEREGION	Clipping region has no overlapping borders.

See also TDC::GetClipBox, TPoint class

OffsetViewportOrg

```
virtual bool OffsetViewportOrg(const TPoint& delta, TPoint* oldOrg = 0);
```

Modifies this DC's viewport origin relative to the current values. The *delta* x- and y-components are added to the previous origin and the resulting point becomes the new viewport origin. The previous origin is saved in *oldOrg*. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::SetViewportOrg, TDC::GetViewportOrg, TPoint class

OffsetWindowOrg

```
bool OffsetWindowOrg(const TPoint& delta, TPoint* oldOrg = 0);
```

Modifies this DC's window origin relative to the current values. The *delta* x- and y-components are added to the previous origin and the resulting point becomes the new window origin. The previous origin is saved in *oldOrg*. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::GetWindowOrg, TDC::SetWindowOrg, TPoint class

OWLFastWindowFrame

void OWLFastWindowFrame(TBrush& brush, TRect& rect, int xWidth, int yWidth)

Draws a frame of the specified size and thickness with the given brush. The old brush is restored after completion.

See also TBrush, TRect

PaintRgn

bool PaintRgn(const TRegion& region);

Paints (fills) the given *region* on this DC using the currently selected brush. Returns **true** if the call is successful; otherwise, returns, **false**.

See also TDC::FillRgn, TDC::SelectObject, TRegion class

PatBlt

bool PatBlt(int x, int y, int w, int h, uint32 rop=PATCOPY);
bool PatBlt(const TRect& dst, uint32 rop=PATCOPY);

Paints the given rectangle using the currently selected brush for this DC. The rectangle can be specified by its upper left coordinates (*x*, *y*), width *w*, and height *h*, or by a single *TRect* argument. The raster-operation code, *rop*, determines how the brush and surface color(s) are combined, as explained in the following table:

Value	Meaning
PATCOPY	Copies pattern to destination bitmap.
PATINVERT	Combines destination bitmap with pattern using the Boolean OR operator.
DSTINVERT	Inverts the destination bitmap.
BLACKNESS	Turns all output to binary 0s.
WHITENESS	Turns all output to binary 1s.

The allowed values of *rop* for this function are a limited subset of the full 256 ternary raster-operation codes; in particular, an operation code that refers to a source cannot be used with *PatBlt*.

Not all devices support the *PatBlt* function, so applications should call *TDC::GetDeviceCaps* to check the features supported by this DC.

PatBlt returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::GetDeviceCaps, TRect class

PathToRegion

HRGN PathToRegion();

If successful, returns a region created from the closed path in this DC; otherwise, returns 0.

Pie

bool Pie(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4);
bool Pie(const TRect& rect, const TPoint& start, const TPoint& end);

Using the currently selected pen and brush objects, draws and fills a pie-shaped wedge by drawing an elliptical arc whose center and end points are joined by lines. The center

of the ellipse is the center of the rectangle specified either by (*x1*, *y1*)/(*x2*, *y2*) or by the *rect* argument. The starting/ending points of pie are specified either by (*x3*, *y3*)/(*x4*, *y4*) or by the points *Start* and *End*. Returns true if the call is successful; otherwise, returns false. The current position is neither used nor altered by this call.

See also TDC::Chord, TDC::Arc, TPoint class, TRegion class, TRect class

PlayMetaFile

bool PlayMetaFile(const TMetaFilePict& metaFile);

Plays the contents of the given *metaFile* on this DC. The metafile can be played any number of times. Returns **true** if the call is successful; otherwise, returns **false**.

See also class TDC::EnumMetaFile, TDC::PlayMetaFileRecord, TMetaFilePict

PlayMetaFileRecord

void PlayMetaFileRecord(HANDLETABLE far& Handletable, METARECORD far& metaRecord, int count);

Plays the metafile record given in *metaRecord* to this DC by executing the GDI function call contained in that record. *Handletable* specifies the object handle table to be used. *count* specifies the number of handles in the table.

See also TDC::PlayMetaFile, TDC::EnumMetaFile, HANDLETABLE struct, METARECORD struct

PlgBlt

bool PlgBlt(const TPoint& dst, const TDC& srcDC, const TRect& src, const TBitmap& maskBm, const TPoint& maskPos, uint32 rop=SRCCOPY);

32-bit only. Performs a bit-block transfer from the given source DC to this DC. Color bits are copied from the *src* rectangle on *srcDC*, the source DC, to the parallelogram *dst* on this DC. The *dst* array specifies three points A,B, and C as the corners of the destination parallelogram. The fourth point D is generated internally from the vector equation D = B + C – A. The upper left corner of *src* is mapped to A, the upper right corner to B, the lower left corner to C, and the lower right corner to D. An optional monochrome bitmap can be specified by the *maskBm* argument. (If *maskBm* specifies a valid monochrome bitmap, *PlgBlt* uses it to mask the colorbits in the source rectangle. An error occurs if *maskBm* is not a monochrome bitmap.) *maskPos* specifies the upper left corner coordinates of the mask bitmap. With a valid *maskBm*, a value of 1 in the mask causes the source color pixel to be copied to *dst*; a value of 0 in the mask indicates that the corresponding color pixel in *dst* will not be changed. If the mask rectangle is smaller than *dst*, the mask pattern will be suitably duplicated.

The destination coordinates are transformed according to this DC (the destination DC). The source coordinates are transformed according to the source DC. If a rotation or shear transformation is in effect for the source DC when *PlgBlt* is called, an error occurs. Other transformations, such as scaling, translation, and reflection are allowed. The stretching mode of this DC (the destination DC) determines how *PlgBlt* will stretch or compress the pixels if necessary. When recording an enhanced metafile, an error occurs if the source DC identifies the enhanced metafile DC.

If necessary, *PlgBlt* adjusts the source color formats to match that of the destination. An error occurs if the source and destination DCs are incompatible. Before using *PlgBlt*, an application should call *GetDeviceCaps* to determine if the source and destination DCs are compatible.

PlgBlt returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::GetDeviceCaps, TDC::MaskBlt, TDC::SetStretchBltMode, TDC::StretchBlt, TBitmap class, TPoint class, TRect class

PolyBezier

bool PolyBezier(const TPoint* points, int count);

Draws one or more connected cubic Bezier splines through the points specified in the *points* array using the currently selected pen object. The first spline is drawn from the first to the fourth point of the array using the second and third points as controls. Subsequent splines, if any, each require three additional points in the array, since the previous end point is taken as the next spline's start point. The *count* argument (>= 4) specifies the total number of points needed to specify the complete drawing. To draw *n* splines, *count* must be set to (3n + 1). Returns **true** if the call is successful; otherwise, returns **false**. The current position is neither used nor altered by this call. The resulting figure is not filled.

See also TDC::PolyBezierTo, TPoint

PolyBezierTo

bool PolyBezierTo(const TPoint* points, int count);

Draws one or more connected cubic Beziers plines through the points specified in the *points* array using the currently selected pen object. The first spline is drawn from the current position to the third point of the array using the first and second points as controls. Subsequent splines, if any, each require three additional points in the array, since the previous end point is taken as the next spline's start point. The *count* argument (>= 4) specifies the total number of points needed to specify the complete drawing. To draw *n* splines, *count* must be set to *3n*. Returns **true** if the call is successful; otherwise, returns **false**. The current position is moved to the end point of the final Bezier curve. The resulting figure is not filled.

See also TDC::PolyBezier, TPoint class

PolyDraw

bool PolyDraw(const TPoint* points, uint8* types, int count);

Using the currently selected pen object, draws one or more possibly disjoint sets of line segments or Bezier splines or both on this DC. The *count* points in the *points* array provide the end points for each line segment or the end points and control points for each Bezier spline or both. The *count* BYTEs in the *types* array determine as follows how the corresponding point in *points* is to be interpreted:

Byte	Meaning
PT_BEZIERTO	This point is a control or end point for a Bezier spline. PT_BEZIERTO types must appear in sets of three: the current position is the Bezier start point; the first two PT_BEZIERTO points are the Bezier control points; and the third PT_BEZIERTO point is the Bezier end point, which becomes the new current point. An error occurs if the PT_BEZIERTO types do not appear in sets of three. An end-point PT_BEZIERTO can be bit-wise OR'd with PT_CLOSEFIGURE to indicate that the current figure is to be closed by drawing a spline from this end point to the start point of the most recent disjoint figure.
PT_CLOSEFIGURE	Optional flag that can be bit-wise OR'd with PT_LINETO or PT_BEZIERTO, as explained above. Closure updates the current point to the new end point.

Byte	Meaning
PT_LINETO	A line is drawn from the current position to this point, which then becomes the new current point. PT_LINETO can be bit-wise OR'd with PT_CLOSEFIGURE to indicate that the current figure is to be closed by drawing a line segment from this point to the start point of the most recent disjoint figure.
PT_MOVETO	This point starts a new (disjoint) figure and becomes the new current point.

PolyDraw is an alternative to consecutive calls to *MoveTo*, *LineTo*, *Polyline*, *PolyBezier*, and *PolyBezierTo*. If there is an active path invoked via *BeginPath*, *PolyDraw* will add to this path.

Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::MoveTo, TDC::LineTo, TDC::PolyBezier, TDC::PolyBezierTo, TDC::Polyline, TDC::BeginPath, TPoint class

Polygon

bool Polygon(const TPoint* points, int count);

Using the current pen and polygon-filling mode, draws and fills on this DC a closed polygon with a number of line segments equal to *count* (which must be >= 2). The *points* array specifies the vertices of the polygon to be drawn. The polygon is automatically closed, if necessary, by drawing a line from the last to the first vertex. The current position is neither used nor altered by *Polygon*. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::Polyline, TDC::SetPolyFillMode, TDC::GetPolyFillMode, TPoint class

Polyline

bool Polyline(const TPoint* points, int count);

Using the current pen object, draws on this DC a *count* of line segments (there must be at least 2). The *points* array specifies the sequence of points to be connected. The current position is neither used nor altered by *Polyline*. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::Polygon, TDC::PolyPolyline, TPoint

PolylineTo

bool PolylineTo(const TPoint* points, int count);

Draws one or more connected line segments on this DC using the currently selected pen object. The first line is drawn from the current position to the first of the *count* points in the *points* array. Subsequent lines, if any, connect the remaining points in the array, with each end point providing the start point of the next segment. The final end point becomes the new current point. No filling occurs even if a closed figure is drawn. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::PolyDraw, TDC::LineTo, TPoint class

PolyPolygon

bool PolyPolygon(const TPoint* points, const int* PolyCounts, int count);

Using the current pen and polygon-filling mode, draws and fills on this DC the number of closed polygons indicated in *count* (which must be >= 2). The polygons can overlap. The *points* array specifies the vertices of the polygons to be drawn. *PolyCounts* is an array

of *count* integers specifying the number of vertices in each polygon. Each polygon must be a closed polygon. The current position is neither used nor altered by *Polygon*. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::PolyPolyline, TDC::SetPolyFillMode, TDC::GetPolyFillMode, TPoint class

PolyPolyline

```
bool PolyPolyline(const TPoint* points, const int* PolyCounts, int count);
```

Using the currently selected pen object, draws on this DC the number of polylines (connected line segments) indicated in *count*. The resulting figures are not filled. The *PolyCounts* array provides *count* integers specifying the number of points (>= 2) in each polyline. The *points* array provides, consecutively, each of the points to be connected. Returns true if the call is successful; otherwise, returns false. The current position is neither used nor altered by this call.

See also TDC::Polyline, TDC::PolyPolygon, TPoint class

PtVisible

```
bool PtVisible(const TPoint& point) const;
```

Returns **true** if the given *point* lies within the clipping region of this DC; otherwise, returns **false**.

See also TDC::RectVisible, TPoint class

RealizePalette

```
int RealizePalette();
```

Maps to the system palette the logical palette entries selected into this DC. Returns the number of entries in the logical palette that were mapped to the system palette.

See also TPalette class

Rectangle

```
bool Rectangle(int x1, int y1, int x2, int y2);
bool Rectangle(const TPoint& p1, const TPoint& p2);
bool Rectangle(const TPoint& point, const TSize& s);
bool Rectangle(const TRect& rect);
```

Draws and fills a rectangle of the given size on this DC with the current pen and brush objects. The current position is neither used nor altered by this call. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::RoundRect, TPoint class, TRect class, TSize class

RectVisible

```
bool RectVisible(const TRect& rect) const;
```

Returns **true** if any part of the given rectangle, *rect*, lies within the clipping region of this DC; otherwise, returns **false**.

See also TDC::PtVisible, TRect class

ResetDC

```
virtual bool ResetDC(DEVMODE far& devMode);
```

Updates this DC using data in the given *devMode* structure. Returns **true** if the call is successful; otherwise, returns **false**.

See also DEVMODE struct

RestoreBrush

```
void RestoreBrush();
```

Restores the original GDI brush object to this DC.

See also TDC::OrgBrush, TBrush class

RestoreDC

```
virtual bool RestoreDC(int savedDC = -1);
```

Restores the given *savedDC*. Returns **true** if the context is successfully restored; otherwise, returns **false**.

See also TDC::SaveDC

RestoreFont

```
virtual void RestoreFont();
```

Restores the original GDI font object to this DC.

See also TDC::OrgFont, TFont class

RestoreObjects

```
void RestoreObjects();
```

Restores all the original GDI objects to this DC.

See also TGdiObject class

RestorePalette

```
void RestorePalette();
```

Restores the original GDI palette object to this DC.

See also TDC::OrgPalette, TPalette class

RestorePen

```
void RestorePen();
```

Restores the original GDI pen object to this DC.

See also TDC::OrgPen, TPen class

RestoreTextBrush

```
void RestoreTextBrush();
```

Restores the original GDI text brush object to this DC.

See also TBrush class

RoundRect

```
bool RoundRect(int x1, int y1, int x2, int y2, int x3, int y3);
bool RoundRect(const TPoint& p1, const TPoint& p2, const TPoint& rad);
bool RoundRect(const TPoint& p, const TSize& s, const TPoint& rad);
bool RoundRect(const TRect& rect, const TPoint& rad);
```

Draws and fills a rounded rectangle of the given size on this DC with the current pen and brush objects. The current position is neither used nor altered by this call. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::Rectangle, TPoint class, TRect class, TSize class

SaveDC

virtual int SaveDC() const;

Saves the current state of this DC on a context stack. The saved state can be restored later with *RestoreDC*. Returns a value specifying the saved DC or 0 if the call fails.

See also TDC::RestoreDC

ScaleViewportExt

virtual bool ScaleViewportExt(int xNum, int xDenom, int yNum, int yDenom, TSize* oldExtent = 0);

Modifies this DC's viewport extents relative to the current values. The new extents are derived as follows:

```
xNewVE = (xOldVE * xNum) / xDenom
yNewVE = (I * yNum) / yDenom
```

The previous extents are saved in *oldExtent*. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::SetViewportExt, TSize cass

ScaleWindowExt

virtual bool ScaleWindowExt(int xNum, int xDenom, int yNum, int yDenom, TSize* oldExtent = 0);

Modifies this DC's window extents relative to the current values. The new extents are derived as follows:

```
xNewWE = (xOldWE * xNum) / xDenom
yNewWE = (yOldWE * yNum) / yDenom
```

The previous extents are saved in *oldExtent*. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::SetWindowExt, TSize

ScrollDC

Form 1 bool ScrollDC(int x, int y, const TRect& scroll, const TRect& clip, TRegion& updateRgn, TRect& updateRect);

Form 2 bool ScrollDC(const TPoint& delta, const TRect& scroll, const TRect& clip, TRegion& updateRgn, TRect& updateRect);

Scrolls a rectangle of bits horizontally by *x* (or *delta.x* in the second version) device-units, and vertically by *y* (or *delta.y*) device-units on this DC. The scrolling and clipping rectangles are specified by *scroll* and *clip*. *ScrollDC* provides data in the *updateRgn* argument telling you the region (not necessarily rectangular) that was uncovered by the scroll. Similarly, *ScrollDC* reports in *updateRect* the rectangle (in client coordinates) that bounds the scrolling update region. This is the largest area that requires repainting.

Returns **true** if the call is successful; otherwise, returns **false**.

See also TPoint class, TRect class, TRegion class

SelectClipPath

bool SelectClipPath(int mode);

Selects the current path on this DC as a clipping region, combining any existing clipping region using the specified *mode* as shown in the following table:

Mode	Meaning
RGN_AND	The new clipping region includes the overlapping areas of the current clipping region and the current path (intersection).
RGN_COPY	The new clipping region is the current path.
RGN_DIFF	The new clipping region includes the areas of the current clipping region with those of the current path excluded.
RGN_OR	The new clipping region includes the combined areas of the current clipping region and the current path (union).
RGN_XOR	The new clipping region includes the combined areas of the current clipping region and the current path but without the overlapping areas.

Returns true if the call is successful; otherwise, returns false.

SelectClipRgn

```
int SelectClipRgn(const TRegion& region);
```

Selects the given *region* as the current clipping region for this DC. A copy of the given *region* is used, letting you select the same region for other DC objects. The return value indicates the new clipping region's type as follows:

Region	Meaning
COMPLEXREGION	Clipping Region has overlapping borders.
ERROR	Invalid DC.
NULLREGION	Clipping region is empty.
SIMPLEREGION	Clipping region has no overlapping borders.

See also TDC::OffsetClipRgn, TDC::GetClipBox, TRegion class

SelectObject

```
void SelectObject(const TBrush& brush);
void SelectObject(const TPen& pen);
virtual void SelectObject(const TFont& font);
void SelectObject(const TPalette& palette, bool forceBackground = false);
```

Selects the given GDI object into this DC. The previously selected object is saved in the appropriate *OrgXXX* protected data member. For a palette argument, if *forceBackgound* is set **false** (the default), the selected logical palette is a foreground palette when the window has input focus. If *forceBackground* is true, the selected palette is always a background palette whether the window has focus or not.

See also TDC::OrgBrush, TDC::OrgFont, TDC::OrgPalette, TDC::OrgPen, TDC::OrgTextBrush, TBrush class, TFont class, TPalette class, TPen class, TMemoryDC::SelectObject

SelectStockObject

```
virtual void SelectStockObject(int index);
```

Selects into the DC a predefined stock pen, brush, font, or palette.

See also TPrintPreviewDC::SelectStockObject

SetBkColor

virtual TColor SetBkColor(TColor color);

Sets the current background color of this DC to the given *color* value or the nearest available. Returns 0x80000000 if the call fails.

See also TDC::GetBkColor, TColor class

SetBkMode

int SetBkMode(int mode);

Sets the background mode to the given *mode* argument, which can be either OPAQUE or TRANSPARENT. Returns the previous background mode.

See also TDC::GetBkMode

SetBoundsRect

uint SetBoundsRect(TRect& bounds, uint flags);

Controls the accumulation of bounding rectangle information for this DC. Depending on the value of *flags*, the given *bounds* rectangle (possibly NULL) can combine with or replace the existing accumulated rectangle. *flags* can be any appropriate combination of the following values:

Constant	Meaning
DCB_ACCUMULATE	Add *bounds* (rectangular union) to the current accumulated rectangle.
DCB_DISABLE	Turn off bounds accumulation.
DCB_ENABLE	Turn on bounds accumulation (the default setting for bounds accumulation is disabled).
DCB_RESET	Set the bounding rectangle to empty.
DCB_SET	Set the bounding rectangle to *bounds*.

There are two bounding-rectangle accumulations, one for Windows and one for the application. The Windows-accumulated bounds can be queried by an application but not altered. The application can both query and alter the DC's accumulated bounds.

See also TDC::GetBoundsRect, TRect class

SetBrushOrg

bool SetBrushOrg(const TPoint& origin, TPoint* oldOrg = 0);

Sets the origin of the currently selected brush of this DC with the given *origin* value. The previous origin is passed to *oldOrg*. Returns **true** if successful; otherwise, returns **false**.

See also TDC::GetBrushOrg, TPoint class

SetDIBits

bool SetDIBits(TBitmap& bitmap, uint16 startScan, uint16 numScans, const void HUGE* bits,
  const BITMAPINFO far& Info, uint16 usage);
bool SetDIBits(TBitmap& Bitmap, const TDib& dib);

The first version sets the pixels in *bitmap* (the given destination bitmap on this DC) from the source DIB (device-independent bitmap) color data found in the byte array *bits* and the BITMAPINFO structure, *Info.numScan* scanlines are taken from the DIB, starting at scanline *startScan*. The *usage* argument specifies how the *bmiColors* member of BITMAPINFO is interpreted, as explained in *TDC::GetDIBits*.

In the second version of *SetDIBits*, the pixels are set in *bitmap* from the given source *TDib* argument.

SetDIBits returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::GetDIBits, TDC::SetDIBitsToDevice, TBitmap class, BITMAPINFO struct, TDib class

SetDIBitsToDevice

```
bool SetDIBitsToDevice(const TRect& dst, const TPoint& src, uint16 startScan, uint16 numScans,
  const void HUGE* bits, const BITMAPINFO far& bitsInfo, uint16 usage);
bool SetDIBitsToDevice(const TRect& dst, const TPoint& src, const TDib& dib);
```

The first version sets the pixels in *dst* (the given destination rectangle on this DC) from the source DIB (device-independent bitmap) color data found in the byte array *bits* and the BITMAPINFO structure, *bitsInfo*. The DIB origin is specified by the point *src*. *numScan* scanlines are taken from the DIB, starting at scanline *startScan*. The *usage* argument determines how the *bmiColors* member of BITMAPINFO is interpreted, as explained in *TDC::GetDIBits*.

In the second version of *SetDIBitsToDevice*, the pixels are set in *dst* from *dib*, the given source *TDib* argument.

SetDIBitsToDevice returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::GetDIBits, TDib class, TPoint class, TRect class, BITMAPINFO struct

SetMapMode

```
virtual int SetMapMode(int mode);
```

Sets the current window mapping mode of this DC to *mode*. Returns the previous mapping mode value. The mapping mode defines how logical coordinates are mapped to device coordinates. It also controls the orientation of the device's x- and y-axes. See *TDC::GetMapMode* for a complete list of mapping modes.

See also TDC::GetMapMode

SetMapperFlags

```
uint32 SetMapperFlags(uint32 flag);
```

Alters the algorithm used by the font mapper when mapping logical fonts to physical fonts on this DC. If successful, the function sets the current font-mapping flag to *flag* and returns the previous mapping flag; otherwise GDI_ERROR is returned. The mapping flag determines whether the font mapper will attempt to match a font's aspect ratio to this DC's aspect ratio. If bit 0 of *flag* is set to 1, the mapper selects only matching fonts. If no matching fonts exist, a new aspect ratio is chosen and a font is retrieved to match this ratio.

SetMiterLimit

```
bool SetMiterLimit(float newLimit, float* oldLimit = 0);
```

Sets the limit of miter joins to *newLimit* and puts the previous value in *oldLimit*. Returns **true** if successful; otherwise, returns **false**.

SetPixel

```
TColor SetPixel(int x, int y, TColor color);
TColor SetPixel(const TPoint& p, TColor color);
```

Sets the color of the pixel at the given location to the given *color* and returns the pixel's previous color.

See also TDC::GetPixel, TColor, TPoint

SetPolyFillMode

int SetPolyFillMode(int mode);

Sets the polygon-filling mode for this DC to the given *mode* value, either ALTERNATE or WINDING. Returns the previous fill mode.

See also TDC::GetPolyFillMode, TDC::Polygon

SetROP2

int SetROP2(int mode);

Sets the current foreground mix mode mode of this DC to the given *mode* value and returns the previous mode. The *mode* argument determines how the brush, pen, and existing screen image combine when filling and drawing. *mode* can be one of the following values:

Value	Meaning
R2_BLACK	Pixel is always binary 0.
R2_COPYPEN	Pixel is the pen color.
R2_MASKNOTPEN	Pixel is a combination of the colors common to both the display and the inverse of the pen.
R2_MASKPEN	Pixel is a combination of the colors common to both the pen and the display.
R2_MASKPENNOT	Pixel is a combination of the colors common to both the pen and the inverse of the display.
R2_MERGEPEN	Pixel is a combination of the pen color and the display color.
R2_MERGENOTPEN	Pixel is a combination of the display color and the inverse of the pen color.
R2_MERGEPENNOT	Pixel is a combination of the pen color and the inverse of the display color.
R2_NOP	Pixel remains unchanged.
R2_NOT	Pixel is the inverse of the display color.
R2_NOTCOPYPEN	Pixel is the inverse of the pen color.
R2_NOTMASKPEN	Pixel is the inverse of the R2_MASKPEN color.
R2_NOTMERGEPEN	Pixel is the inverse of the R2_MERGEPEN color.
R2_NOTXORPEN	Pixel is the inverse of the R2_XORPEN color.
R2_WHITE	Pixel is always binary 1.
R2_XORPEN	Pixel is a combination of the colors in the pen and in the display, but not in both.

See also TDC::GetROP2, TDC::GetDeviceCaps

SetStretchBltMode

int SetStretchBltMode(int mode);

Sets the stretching mode of this DC to the given *mode* value and returns the previous mode. The *mode* argument (BLACKONWHITE, COLORONCOLOR, or WHITEONBLACK) defines which scan lines or columns or both are eliminated by *TDC::StretchBlt*.

See also TDC::GetStretchBltMode, TDC::StretchBlt

SetSystemPaletteUse

int SetSystemPaletteUse(int usage);

Changes the usage of this DC's system palette. The *usage* argument can be SYSPAL_NOSTATIC or SYSPAL_STATIC. Returns the previous usage value.

See also TDC::GetSystemPaletteUse

SetTextAlign

uint SetTextAlign(uint flags);

Sets the text-alignment flags for this DC. If successful, *SetTextAlign* returns the previous text-alignment flags; otherwise, it returns GDI_ERROR. The flag values are as listed for the *TDC::GetTextAlign* function. The text-alignment flags determine how *TDC::TextOut* and *TDC::ExtTextOut* align text strings in relation to the first character's screen position.

See also TDC::GetTextAlign, TDC::TextOut, TDC::ExtTextOut

SetTextCharacterExtra

int SetTextCharacterExtra(int extra);

If successful, sets the current intercharacter spacing to *extra*, in logical units, for this DC, and returns the previous intercharacter spacing. Otherwise, returns 0. If the current mapping mode is not MM_TEXT, the *extra* value is transformed and rounded to the nearest pixel.

See also TDC::GetTextCharacterExtra

SetTextColor

virtual TColor SetTextColor(TColor color);

Sets the current text color of this DC to the given *color* value. The text color determines the color displayed by *TDC::TextOut* and *TDC::ExtTextOut*.

See also TDC::GetTextColor, TColor

SetTextJustification

bool SetTextJustification(int breakExtra, int breakCount);

When text strings are displayed using *TDC::TextOut* and *TDC::ExtTextOut*, sets the number of logical units specified in *breakExtra* as the total extra space to be added to the number of break characters specified in *breakCount*. The extra space is distributed evenly between the break characters. The break character is usually ASCII 32 (space), but some fonts define other characters. *TDC::GetTextMetrics* can be used to retrieve the value of the break character.

If the current mapping mode is not MM_TEXT, the extra value is transformed and rounded to the nearest pixel.

SetTextJustification returns **true** if the call is successful; otherwise, it returns **false**.

SetViewportExt

virtual bool SetViewportExt(const TSize& extent, TSize* oldExtent = 0);

Sets this DC's viewport x- and y-extents to the given *extent* values. The previous extents are saved in *oldExtent*. Returns **true** if the call is successful; otherwise, returns **false**. The *extent* value determines the amount of stretching or compression needed in the logical coordinate system to fit the device coordinate system. *extent* also determines the relative orientation of the two coordinate systems.

See also TDC::GetViewportExt, TSize class

SetViewportOrg

virtual bool SetViewportOrg(const TPoint& origin, TPoint* oldOrg = 0);

Sets this DC's viewport origin to the given *origin* value, and saves the previous origin in *oldOrg*. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::GetViewportOrg, TDC::OffsetViewportOrg, TPoint class

SetWindowExt

virtual bool SetWindowExt(const TSize& extent, TSize* oldExtent = 0);

Sets this DC's window x- and y-extents to the given *extent* values. The previous extents are saved in *oldExtent*. Returns **true** if the call is successful; otherwise, returns **false**. The *extent* value determines the amount of stretching or compression needed in the logical coordinate system to fit the device coordinate system. *extent* also determines the relative orientation of the two coordinate systems.

See also TDC::GetWindowExt, TDC::ScaleWindowExt, TSize class

SetWindowOrg

bool SetWindowOrg(const TPoint& origin, TPoint* oldOrg = 0);

Sets the origin of the window associated with this DC to the given *origin* value, and saves the previous origin in *oldOrg*. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::GetWindowOrg, TDC::OffsetWindowOrg, TPoint class

SetWorldTransform

bool SetWorldTransform(XFORM far& xform);

32-bit only. Sets a two-dimensional linear transformation, given by the *xform* structure, between world space and page space for this DC. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::ModifyWorldTransform, XFORM struct

StretchBlt

bool StretchDIBits(const TRect& dst, const TRect& src, const void HUGE* bits, const BITMAPINFO far& bitsInfo, uint16 usage, uint32 rop=SRCCOPY);

bool StretchDIBits(const TRect& dst, const TRect& src, const TDib& dib, uint32 rop=SRCCOPY);

Copies the color data from *src*, the source rectangle of pixels in the given DIB (device-independent bitmap) on this DC, to *dst*, the destination rectangle. The DIB bits and color data are specified in either the byte array *bits* and the BITMAPINFO structure *bitsInfo* or in the *TDib* object, *dib*. The rows and columns of color data are stretched or compressed to match the size of the destination rectangle. The *usage* argument specifies how the *bmiColors* member of BITMAPINFO is interpreted, as explained in *TDC::GetDIBits* The raster operation code, *rop*, specifies how the source pixels, the current brush for this DC, and the destination pixels are combined to produce the new image. See *TDC::MaskBlt* for a detailed list of *rop* codes.

See also TDC::GetDIBits, TDC::MaskBlt, TDib class, TRect class, BITMAPINFO struct

StretchDIBits

```
bool StretchDIBits(const TRect& dst, const TRect& src, const void HUGE* bits, const BITMAPINFO far& bitsInfo,
  uint16 usage, uint32 rop=SRCCOPY);
bool StretchDIBits(const TRect& dst, const TRect& src, const TDib& dib, uint32 rop=SRCCOPY);
```

Copies the color data from *src*, the source rectangle of pixels in the given DIB (device-independent bitmap) on this DC, to *dst*, the destination rectangle. The DIB bits and color data are specified in either the byte array *bits* and the BITMAPINFO structure *bitsInfo* or in the *TDib* object, *dib*. The rows and columns of color data are stretched or compressed to match the size of the destination rectangle. The *usage* argument specifies how the *bmiColors* member of BITMAPINFO is interpreted, as explained in *TDC::GetDIBits* The raster operation code, *rop*, specifies how the source pixels, the current brush for this DC, and the destination pixels are combined to produce the new image. See *TDC::MaskBlt* for a detailed list of *rop* codes.

See also TDC::GetDIBits, TDC::MaskBlt, TDib class, TRect class, BITMAPINFO struct

StrokeAndFillPath

```
bool StrokeAndFillPath();
```

32-bit only. Closes any open figures in the current path of this DC, strokes the outline of the path using the current pen, and fills its interior using the current brush and polygon fill mode. Returns **true** if the call is successful; otherwise, returns **false**.

See also TDC::StrokePath, TDC::BeginPath, TDC::FillPath, TDC::EndPath, TDC::SetPolyFillMode, TBrush class, TPen class

StrokePath

```
bool StrokePath();
```

32-bit only. Renders the current, closed path on this DC and uses the DC's current pen.

See also TDC::StrokeAndFillPath, TDC::BeginPath

TabbedTextOut

```
bool TabbedTextOut(const TPoint& p, const char far* string, int count, int numPositions, const int far* positions,
  int tabOrigin);
virtual bool TabbedTextOut(const TPoint& p, const char far* string, int count, int numPositions,
  const int far* positions, int tabOrigin, TSize& size);
```

Draws up to *count* characters of the given null-terminated *string* in the current font on this DC. If *count* is –1, the whole string is written.

Tabs are expanded according to the given arguments. The *positions* array specifies *numPositions* tab stops given in device units. The tab stops must have strictly increasing values in the array. If *numPositions* and *positions* are both 0, tabs are expanded to eight times the average character width. If *numPositions* is 1, all tab stops are taken to be *positions[0]* apart. *tabOrigin* specifies the x-coordinate in logical units from which tab expansion will start.

The *p* argument specifies the logical coordinates of the reference point that is used to align the first character depending on the current text-alignment mode. This mode can be inspected with *TDC::GetTextAlign* and changed with *TDC::SetTextAlign*. By default, the current position is neither used nor altered by *TabbedTextOut*. However, if the align mode is set to TA_UPDATECP, *TabbedTextOut* ignores the reference point argument(s) and uses/updates the current position as the reference point.

The *size* argument in the second version of *TabbedTextOut* reports the dimensions (*size.y* = height and *size.x* = width) of the string in logical units.

TabbedTextOut returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::TextOut, TDC::GetTextAlign, TDC::SetTextAlign, TPoint class, TSize class

TextOut

```
virtual bool TextOut(int x, int y, const char far* string, int count = -1);
bool TextOut(const TPoint& p, const char far* string, int count = -1);
```

Draws up to *count* characters of the given null-terminated *string* in the current font on this DC. If *count* is –1 (the default), the entire string is written.

The (*x*, *y*) or *p* arguments specify the logical coordinates of the reference point that is used to align the first character, depending on the current text-alignment mode. This mode can be inspected with *TDC::GetTextAlign* and changed with *TDC::SetTextAlign*. By default, the current position is neither used nor altered by *TextOut*. However, the align mode can be set to TA_UPDATECP, which makes Windows use and update the current position. In this mode, *TextOut* ignores the reference point argument(s).

TextOut returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::ExtTextOut, TDC::GetTextAlign, TDC::SetTextAlign, TPoint class

TextRect

```
bool TextRect(int x1, int y1, int x2, int y2);
bool TextRect(const TRect& rect);
bool TextRect(int x1, int y1, int x2, int y2, TColor color);
bool TextRect(const TRect rect, TColor color);
```

Fills the given rectangle, clipping any text to the rectangle. If no *color* argument is supplied, the current backgound color is used. If a *color* argument is supplied, that color is set to the current background color which is then used for filling. *TextRect* returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::SetBkColor, TColor class, TRect class

UpdateColors

```
void UpdateColors();
```

Updates the client area of this DC by matching the current colors in the client area to the system palette on a pixel-by-pixel basis.

WidenPath

```
bool WidenPath();
```

32-bit only. Redefines the current, closed path on this DC as the area that would be painted if the path were stroked with this DC's current pen. The current pen must have been created under the following conditions:

If the *TPen(***int** *Style*, **int** *Width*, *TColor Color)* constructor, or the *TPen(const LOGPEN* LogPen)* constructor is used, the width of the pen in device units must be greater than 1.

If the *TPen(***uint32** *PenStyle*, **uint32** *Width*, **const** *TBrush& Brush*, **uint32** *StyleCount*, *LPDWORD pSTyle)* constructor, or the *TPen(***uint32** *PenStyle*, **uint32** *Width*, **const**

LOGBRUSH& logBrush, **uint32** *StyleCount*, *LPDWORD pSTyle)* constructor is used, the pen must be a geometric pen.

Any Bezier curves in the path are converted to sequences of linear segments approximating the widened curves, so no Bezier curves remain in the path after a *WidenPath* call.

WidenPath returns **true** if the call is successful; otherwise, it returns **false**.

See also TDC::FlattenPath, TDC::BeginPath, TPen class

Protected constructors

Constructors

Form 1 TDC();
For use by derived classes only. Calls *Init* to clear the *OrgXXX* data members and sets *ShouldDelete* to **true**.

Form 2 TDC(HDC handle, TAutoDelete AutoDelete);
For use by derived classes only. Constructs a TDC object using an existing DC handle. Calls *Init* to clear the *OrgXXX* data members.

See also TDC::Init

Protected data members

Handle

TGdiBase::Handle;
The handle of this DC. Uses the base class's handle (*TGdiBase::Handle*.)

See also TDC

OrgBrush

HBRUSH OrgBrush;
Handle to the original GDI brush object for this DC. Holds the previous brush object whenever a new brush is selected with *SelectObject(brush)*.

See also TDC::SelectObject, TBrush class

OrgFont

HFONT OrgFont;
Handle to the original GDI font object for this DC. Holds the previous font object whenever a new font is selected with *SelectObject(font)*.

See also TDC::SelectObject, TFont class

OrgPalette

HPALETTE OrgPalette;
Handle to the original GDI palette object for this DC. Holds the previous palette object whenever a new palette is selected with *SelectObject(palette)*.

See also TDC::SelectObject, TPalette class

OrgPen

HPEN OrgPen;

Handle to the original GDI pen object for this DC. Holds the previous pen object whenever a new pen is selected with *SelectObject(pen)*.

See also TDC::SelectObject, TPen class

OrgTextBrush

HBRUSH OrgTextBrush

32-bit only. The handle to the original GDI text brush object for this DC. Stores the previous text brush handle whenever a new brush is selected with *SelectObject(text_brush)*.

See also TDC::SelectObject, TBrush class

ShouldDelete

TGdiBase::ShouldDelete;

Set to **true** if the handle for this object should be deleted by the destructor; otherwise, set to **false**.

Protected member functions

CheckValid

Form 1 TGdiBase::CheckValid(uint resId=IDS_GDIFAILURE)

Form 2 static void CheckValid(HANDLE handle, uint resId=IDS_GDIFAILURE)

Both versions of *CheckValid* check for a valid GDI object handle. If one is not found a GDI exception is thrown for the given resource id. Both versions use *TGdiBase::CheckValid*.

GetAttributeHDC

virtual HDC GetAttributeHDC() const;

Returns the attributes of the DC object.

See also TPrintPreviewDC::GetAttributeHDC

GetHDC

HDC GetHDC() const;

Returns a handle to the DC.

Init

void Init();

Sets *OrgBrush*, *OrgPen*, *OrgFont*, *OrgBitmap*, and *OrgPalette* to 0, and sets *ShouldDelete* to **true**. This function is for internal use by the *TDC* constructors.

See also TDC constructors, TDC::SelectObject

TDecoratedFrame class

decframe.h

TDecoratedFrame automatically positions its client window (you must supply a client window) so that it is the same size as the client rectangle. You can add additional

decorations like toolbars and status lines to a window.You can create a *TDecoratedFrame* without a caption bar by clearing all of the bits in the style data member of the *TWindowAttr* structure. *TDecoratedFrame* is a streamable class.

For OLE-enabled applications, use *TOleFrame*, which creates a decorated frame and manages decorations such as toolbars for the main window of an SDI (Single Document Interface) OLE application.

See also TOleFrame

Type definitions

TLocation

```
enum TLocation (Top, Bottom, Left, Right);
```

TLocation enum describes *Top, Left, Bottom,* and *Right* positions where the decoration can be placed. *Insert* uses this enum to position the decoration.

Public constructor

Constructor

```
TDecoratedFrame(TWindow* parent, const char far *title, TWindow* clientWnd, bool trackMenuSelection = false,
  TModule* module = 0);
```

Constructs a *TDecoratedFrame* object with the specified parent window (*parent*), window caption (*title*), and module ID. Sets *TWindow::Attr.Title* to the new title. Passes a pointer to the client window if one is specified. By default set to **false**, *trackMenuSelection* controls whether hint text appears at the bottom of the window when a menu item is highlighted.

Public member functions

Insert

```
void Insert (TWindow& decoration, TLocation = Top);
```

After you specify where the decoration should be placed, *Insert* adds it just above, below, left, or right of the client window. This process is especially important when there are multiple decorations. *Insert* looks at the decoration's *Attr.Style* member and checks the WS_VISIBLE flag to tell whether the decoration should initially be visible or hidden.

To position the decoration, *Insert* uses *TLocation* enum, which describes *Top, Left, Bottom,* and *Right* positions where the decoration can be placed.

PreProcessMsg

```
bool PreProcessMsg (MSG& msg);
```

Overrides the virtual function defined in *TFrameWindow* to give decorations an opportunity to perform mnemonic access preprocessing.

See also TFrameWindow::PreProcessMsg, TWindow::PreProcessMsg

SetClientWindow

```
TWindow* SetClientWindow(TWindow* clientWnd);
```

Overrides *TFrameWindow's* virtual function. Sets the client window to the specified window. Users are responsible for destroying the old client window if they want to remove it.

Protected data members

MenuItemID
uint MenuItemID;
Specifies the menu item ID.

TrackMenuSelection
bool TrackMenuSelection;
Specifies whether you want menu selection and help status information visible.

Protected member functions

EvCommand
LRESULT EvCommand(uint Id, HWND hWndCtl, uint notifyCode);
Automates hiding and showing of decorations.

EvCommandEnable
void EvCommandEnable(TCommandEnabler& ce);
Handles checking and unchecking of menu items that are associated with decorations.

EvEnterIdle
void EvEnterIdle(uint source, HWND hWndDlg);
Responds to a window message that tells an application's main window that a dialog box or a menu is entering an idle state. *EvEnterIdle* also handles updating the status bar with the appropriate help message.

EvMenuSelect
void EvMenuSelect(uint MenuItemId, uint flags, HMENU hMenu);
Responds to user menu selection. If *MenuItemId* is blank, displays an empty help message; otherwise, it displays a help message with the specified string ID. See *EvEnterIdle* for a description of how the help message is loaded.

EvSize
void EvSize(uint sizeType, TSize& size);
Passes a WM_SIZE message to *TLayoutWindow*.

SetupWindow
void SetupWindow();
Calls *TLayoutWindow::Layout* to size and position the decoration.

See also TFrameWindow::SetUpWindow, TWindow::SetUpWindow, TLayoutWindow::Layout

Response table entries

Response table entry	Member function
EV_WM_ENTERIDLE	EvEnterIdle
EV_WM_MENUSELECT	EvMenuSelect
EV_WM_SIZE	EvSize

TDecoratedMDIFrame class

decmdifr.h

Derived from both *TMDIFrame* and *TDecoratedFrame*, *TDecoratedMDIFrame* is an MDI frame that supports decorated child windows.

TDecorated MDIFrame supports custom toolbars. You can insert one set of decorations (for example, toolbars and rulers) into a decorated frame. When a different set of tools is needed, you can remove the previous set and reinsert another set of decorations. However, be sure to remove all of the unwanted decorations from the adjusted sides (that is, the top, left, bottom, and right) before reinserting a new set.

TDecoratedMDIFrame is a streamable class.

Public constructor

Constructor

```
TDecoratedMDIFrame(const char far *title, TResId menuResId, TMDIClient &clientWnd = *new TMDIClient,
  bool trackMenuSelection = false, TModule* module = 0);
```

Constructs a decorated MDI frame of the specified client window with the indicated menu resource ID. By default, menu hint text is not displayed.

Protected member function

DefWindowProc

```
LRESULT DefWindowProc(uint message, WPARAM wParam, LPARAM lParam);
```

Overrides *TWindow::DefWindowProc*. If the message parameter is WM_SIZE, *DefWindowProc* returns 0; otherwise, *DefWindowProc* returns the result of calling *TMDIFRame::DefWindowProc*.

See also TMDIFrame::DefWindowProc

Response table entries

The *TDecoratedMDIFrame* response table has no entries.

TDesktopDC class

dc.h

A DC class derived from *TWindowDC*, *TDesktopDC* provides access to the desktop window's client area, which is the screen behind all other windows.

Public constructor

Constructor
TDesktopDC();
Default constructor for *TDesktopDC* objects.

TDialog class

dialog.h

Typically used to obtain information from a user, a dialog box is a window inside of which other controls such as buttons and scroll bars can appear. Unlike actual child windows which can only be displayed in the parent window's client area, dialog boxes can be moved anywhere on the screen. *TDialog* objects represent both modal and modeless dialog box interface elements. (A modal dialog box disables operations in its parent window while it is open, and, thus, lets you function in only one window "mode.")

A *TDialog* object has a corresponding resource definition that describes the placement and appearance of its controls. The identifier of this resource definition is supplied to the constructor of the *TDialog* object. A *TDialog* object is associated with a modal or modeless interface element by calling its *Execute* or *Create* member function, respectively.

You can use *TDialog* to build an application that uses a dialog as its main window by constructing your dialog as a *TDialog* and passing it as the client of a *TFrameWindow*. Your code might look something like this:

```
SetMainWindow(new TFrameWindow(0, "title" new TDialog(0, IDD_MYDIALOG)));
```

ObjectWindows provides three-dimensional (3-D) support for dialog boxes. If your application expects to use Microsoft's CTL3D DLL, you need to register your application by calling *TApplication::EnableCtl3d*. ObjectWindows will then automatically forward the WM_CTLCOLOR message to the CTL3D DLL.

ObjectWindows also provides BWCC support for dialog boxes. Unless a custom template is specified, *TDialog* uses the BWCC templates. (By default, *TApplication's* member function *EnableBWCC* enables BWCC support.)

TDialog is a streamable class.

ObjectWindows also encapsulates common dialog boxes that let the user select font, file name, color, print options, and so on. *TCommonDialog* is the parent class for this group of common dialog box classes.

Public data members

Attr

TDialogAttr Attr;

Attr holds the creation attributes of the dialog box (for example, size and style).

See also TDialogAttr

IsModal

bool IsModal;

IsModal is **true** if the dialog box is modal and **false** if it is modeless.

Public constructor and destructor

Constructor

TDialog(TWindow* parent, TResId resId, TModule* module = 0);

Invokes a *TWindow* constructor, passing *parent* and *module*, and calls *DisableAutoCreate* to prevent *TDialog* from being automatically created and displayed along with its parent. *TDialog* then initializes *Title* to –1 and sets *TDialogAttr.Name* using the dialog box's integer or string resource identifier, which must correspond to a dialog resource definition in the resource file. Finally, it initializes *TDialogAttr.Param* to 0 and sets *IsModal* to **false**.

Destructor

~TDialog();

If *Attr.Name* is a string and not an integer resource identifier, this destructor frees memory allocated to *Attr.Name*, which holds the name of the dialog box.

See also TApplication::EnableBWCC, TWindow::~TWindow, TWindow::DisableAutoCreate, TWindow::TWindow, TDialog::Attr

Public member functions

CloseWindow

void CloseWindow(int retValue = IDCANCEL);

Overrides the virtual function defined by *TWindow* and conditionally shuts down the dialog box. If the dialog box is modeless, it calls *TWindow::CloseWindow*. If the dialog box is modal, it calls *CanClose*. If *CanClose* returns **true**, *CloseWindow* calls *TransferData* to transfer dialog box data, passing it *retValue*. The default value of *retValue* is IDCANCEL.

See also TWindow::CloseWindow

CmCancel

void CmCancel();

Automatic response to a click on the Cancel button of the dialog box. Calls *Destroy*, passing IDCANCEL.

See also TDialog::CloseWindow

CmOk

void CmOk();

Responds to a click on the dialog box's OK button with the identifier IDOK. Calls *CloseWindow*, passing IDOK.

See also TDialog::CloseWindow

Create

virtual bool Create();

Creates a modeless dialog box interface element associated with the *TDialog* object. Registers all the dialog's child windows for custom control support. Calls *DoCreate* to perform the actual creation of the dialog box.

Create returns **true** if successful. If unsuccessful, *Create* throws a *TXInvalidWindow* exception.

See also TDialog::Execute, TModule::MakeWindow, TWindow::DisableAutoCreate

Destroy

virtual void Destroy(int retValue = IDCANCEL);

Destroys the interface element associated with the *TDialog* object. If the element is a modeless dialog box, *Destroy* calls *TWindow::Destroy*. If the element is a modal dialog box, *Destroy* calls *EnableAutoCreate* on all child windows. Then *Destroy* calls the Windows function *::EndDialog*, passing *retValue* as the value returned to indicate the result of the dialog's execution. The default *retValue* is IDCANCEL.

See also TWindow::Destroy, TWindow::EnableAutoCreate

DialogFunction

virtual bool DialogFunction(uint message, WPARAM wParam, LPARAM lParam);

To process messages within the dialog function, your application must override this function. *DialogFunction* returns **true** if the message is handled and **false** if the message is not handled.

DoCreate

virtual HWND DoCreate();

DoCreate is called by *Create* to performs the actual creation of a modeless dialog box.

DoExecute

virtual int DoExecute();

DoExecute is called by *Execute* to perform the actual execution of a modal dialog box.

See also TDialog::Execute

EvClose

void EvClose();

Responds to an incoming *EvClose* message by shutting down the window.

EvCtlColor

HBRUSH EvCtlColor(HDC hDC, HWND hWndChild, uint ctlType);

Passes the handle to the display context for the child window, the handle to the child window, and the default system colors to the parent window. The parent window then

uses the display-context handle given in *hDC* to set the text and background colors of the child window.

If three-dimensional (3-D) support is enabled, *EvCtlColor* handles the EV_WM_CTLCOLOR message by allowing the CTL3D DLL to process the WM_CTLCOLOR message in order to set the background color and provide a background brush for the window.

See also TApplication::EnableCtl3d

EvInitDialog

virtual bool EvInitDialog(HWND hWndFocus);

EvInitDialog is automatically called just before the dialog box is displayed. It calls *SetupWindow* to perform any setup required for the dialog box or its controls.

See also TWindow::SetupWindow

EvPaint

void EvPaint();

EvPaint calls *TWindow's* general-purpose default processing function, *DefaultProcessing*, for Windows-supplied painting.

See also TWindow::DefaultProcessing

EvSetFont

virtual void EvSetFont(HFONT hfont, bool redraw);

Responds to a request to change a dialog's font.

Execute

virtual int Execute();

Creates and executes a modal dialog box interface element associated with the *TDialog* object. If the element is successfully associated, *Execute* does not return until the *TDialog* is closed.

Execute performs the following tasks:

1. Registers this dialog's window class and all of the dialog's child windows.
2. Calls *DoExecute* to execute the dialog box.
3. Checks for exceptions and throws a *TXWindow* exception if an error occurs.

Execute returns an integer value that indicates how the user closed the modal dialog box. If the dialog box cannot be created, *Execute* returns –1.

See also TModule::ExecDialog, TWindow::DisableAutoCreate, TXWindow

GetDefaultId

uint GetDefaultId() const;

Gets the default resource ID.

GetItemHandle

HWND GetItemHandle(int childId);

Returns the dialog box control's window handle identified by the supplied ID. Because *GetItemHandle* is now obsolete, new applications should use *TWindow::GetDlgItem*.

PerformDlgInit

bool PerformDlgInit();

Initializes the dialog box controls with the contents of RT_DLGINIT, the dialog box resource identifier, which describes the appearance and location of the controls (buttons, group boxes, and so on). Returns **true** if successful; otherwise, returns **false**.

PreProcessMsg

bool PreProcessMsg(MSG& msg);

Overrides the virtual function defined by *TWindow* in order to perform preprocessing of window messages. If the child window has requested keyboard navigation, *PreProcessMsg* handles any accelerator key messages and then processes any other keyboard messages.

See also TWindow::PreProcessMsg, MSG struct

SendDlgItemMsg

uint32 SendDlgItemMsg(int ChildId, uint16 Msg, uint16 WParam, uint32 LParam);

Sends a window control message, identified by *Msg*, to the dialog box's control identified by its supplied ID, *ChildID*. *WParam* and *LParam* become parameters in the control message. *SendDlgItemMsg* returns the value returned by the control, or 0 if the control ID is invalid. This function is obsolete.

SetCaption

void SetCaption(const char far* title);

Sets the caption of the dialog box. to the value of the *title* parameter.

See also TWindow::SetCaption

SetDefaultId

void SetDefaultId(uint Id);

Sets the default resource ID.

Protected member functions

GetClassName

char far* GetClassName();

Overrides the virtual function defined in *TWindow* and returns the name of the dialog box's default Windows class, which must be used for a modal dialog box. For a modeless dialog box, *GetClassName* returns the name of the default *TWindow*. If BWCC is enabled, *GetClassName* returns BORDLGCLASS.

See also TWindow::GetClassName

GetWindowClass

void GetWindowClass(WNDCLASS& wndClass);

Overrides the virtual function defined in *TWindow*. Fills *WndClass* with a *TDialog* registration attributes obtained from an existing *TDialog* window or from BWCC if it is enabled.

If the class style is registered with CS_GLOBALCLASS, you must unregister the class style. You can do this by turning off the style bit. For example,

```
{
baseclass::GetWindowClass(wndClass);
....
WndClass.style &= ~CS_GLOBALCLASS:
...
}
```

See also TWindow::GetWindowClass, TWindow, WNDCLASS struct

SetupWindow

void SetupWindow();

Overrides the virtual function defined in *TWindow*. Sets up the dialog box by calling *SetCaption* (sets *Title*) and *TWindow::SetupWindow*.

If three-dimensional (3-D) support is enabled, *SetupWindow* calls the CTL3D DLL to register the dialog box.

See also TCommonDialog::SetupWindow, TDialog::SetCaption, TWindow::SetupWindow

Response table entries

Response table entry	Member function
EV_COMMAND (IDCANCEL, CmCancel)	*CmCancel*
EV_COMMAND (IDOK, CmOk)	*CmOk*
EV_WM_CTLCOLOR	*EvCtlColor*
EV_WM_CLOSE	*EvClose*
EV_WM_PAINT	*EvPaint*

See also TCommonDialog

TDialog class::TDialogAttr struct

dialog.h

A *TDialogAttr* is used to hold a *TDialog*'s creation attributes, which include the style, appearance, size, and types of controls associated with the dialog box. *TDialogAttr* contains two data members: *Name* (the resource id) and *Param*. These members contain user-defined data used for dialog box creation.

Public data members

Name

char far* Name;

Name holds the identifier, which can be either a string or an integer resource identifier, of the dialog box resource.

Param

uint32 Param;

Param is used to pass initialization data to the dialog box when it is constructed. You can assign a value to this field in a derived class's constructor. Although any *Param*-type information passed to the dialog box can be saved as a data member, *Param* is especially useful if you want to create a dialog box that's implemented by non-ObjectWindows code.

After *Param* is accepted it is then available in the message response functions (for example, *EvInitDialog*), associated with WM_INITDIALOG.

See also TDialog::Attr

TDib class

gdiobjec.h

The class *TDib*, derived from *TGdiObject*, represents GDI Device Independent Bitmap (DIB) objects. *TDibDCs* encapsulate the creation of DCs using DIB.DRV (a GDI driver provided with Windows MME and 3.1). DIBs have no Windows handle; they are just structures containing format and palette information and a collection of bits or pixels. *TDib* provides a convenient way to work with DIBs like any other GDI object. The memory for the DIB is in one *GlobalAlloc'd* unit so it can be passed to the Clipboard, OLE 2, and so on.

The *TDib* Destructor overloads the base destructor because DIBs are not real GDI objects.

Type definitions

Map

enum Map{MapFace, MapText, MapShadow, MapHighlight, MapFrame};

Enumerates the values for the part of the window whose color is to be set. You can OR these together to control the colors used for face shading on push buttons, the color of a selected control button, the edge shading on push buttons, text on push buttons, the color of the window frame, and the background color of the various parts of a window. The function *MapUIColors* uses one of these values to map the colors of various parts of a window to a specified color.

See also TDib::MapUIColors

Public constructors and destructor

Constructors

Form 1 TDib(HGLOBAL handle, TAutoDelete autoDelete = NoAutoDelete);

Creates a *TDib* object and sets the *Handle* data member to the given borrowed handle. The *ShouldDelete* data member defaults to false, ensuring that the borrowed handle will not be deleted when the C++ object is destroyed.

Form 2 TDib(const TClipboard& clipboard);

Constructs a *TDib* object with a handle borrowed from the given Clipboard.

Form 3 TDib(const TDib& dib);

This public copy constructor creates a complete copy of the given *dib* object as in

```
TDib myDib = yourDib;
```

Form 4 **TDib(int width, int height, int nColors, uint16 mode = DIB_RGB_COLORS);**
Creates a DIB object with the given width, height, number of colors, mode values.

Form 5 **TDib(HINSTANCE instance, TResID resID);**
Creates a DIB object from the resource with the given ID.

Form 6 **TDib(const char* name);**
Creates a DIB object from the given resource file.

Form 7 **TDib(const TBitmap& bitmap, const TPalette* pal = 0);**
Creates a DIB object from the given resource bitmap and palette. If *pal* is 0 (the default), the default palette is used.

Destructor

~TDib();
Overrides the base destructor.

See also ::GetClipboardData, ~TGdiObject, TDib::InfoFromHandle, TDib::LoadFile, TDib::LoadResource, TGdiObject::Handle, TGdiObject::ShouldDelete

Public member functions

operator BITMAPINFO()

operator const BITMAPINFO far*() const;
operator BITMAPINFO far*();

Typecasts this DIB by returning a pointer to its bitmap information structure (BITMAPINFO) which contains information about this DIB's color format and dimensions (size, width, height, resolution, and so on).

See also TDib::GetInfo, BITMAPINFO struct

operator BITMAPINFOHEADER()

operator const BITMAPINFOHEADER far*() const;
operator BITMAPINFOHEADER far*();
Typecasts this DIB by returning a pointer to its bitmap info header.

See also TDib::GetInfoHeader, BITMAPINFOHEADER struct

ChangeModeToPal

bool ChangeModeToPal(const TPalette& pal);
Converts the existing color table in place to use palette relative values. The palette that is passed is used as a reference. Returns **true** if the call is successful; otherwise returns **false**.

See also TDib::ChangeModeToRGB, TPalette::GetPaletteEntry

ChangeModeToRGB

bool ChangeModeToRGB(const TPalette& pal);

Converts the existing color table in place to use absolute RGB values. The palette that is passed is used as a reference. Returns **true** if the call is successful; otherwise returns **false**.

See also TDib::ChangeModetoPal, TPalette::GetPaletteEntry

FindColor

```
int FindColor(TColor color);
```

Returns the palette entry for the given color.

See also TDib::GetColor, TColor, TDib::SetColor, TDib::MapColor

FindIndex

```
int FindIndex(uint16 index);
```

Returns the palette entry corresponding to the given index.

See also TDib::GetIndex, TDib::SetIndex, TDib::MapIndex

GetBits

```
const void HUGE* GetBits() const;
void HUGE* GetBits();
```

Returns the *Bits* data member for this DIB.

See also TDib::Bits

GetColor

```
TColor GetColor(int entry) const;
```

Returns the color for the given palette entry.

See also TDib::SetColor, TColor, TDib::FindColor, TDib::MapColor

GetColors

```
const TRgbQuad far* GetColors() const;
TRgbQuad far* GetColors();
```

Returns the *bmiColors* value of this DIB.

See also TRgbQuad

GetIndex

```
uint16 GetIndex(int entry) const;
```

Returns the color index for the given palette entry.

See also TDib::SetIndex, TDib::FindIndex,, TDib::MapIndex

GetIndices

```
const uint16 far* GetIndices() const;
uint16 far* GetIndices();
```

Returns the *bmiColors* indexes of this DIB.

See also TDib::GetColors

GetInfo

```
const BITMAPINFO far* GetInfo() const;
BITMAPINFO far* GetInfo();
```

Returns this DIB's *Info* field. A DIB's *BITMAPINFO* structure contains information about the dimensions and color of the DIB and specifies an array of data types that define the colors in the bitmap.

See also TDib::Info, TDib::GetInfoHeader

GetInfoHeader

```
const BITMAPINFOHEADER far* GetInfoHeader() const;
BITMAPINFOHEADER far* GetInfoHeader();
```

Returns this DIB's *bmiHeader* field of the *BITMAPINFO* structure contains information about the color and dimensions of this DIB.

See also TDib::Info, TDib::GetInfo, BITMAPINFOHEADER struct

operator HANDLE()

```
operator HANDLE() const;
```

Typecasts this DIB by returning its *Handle*.

See also TGdiObject::Handle

Height

```
int Height() const;
```

Returns *H*, this DIB's height.

See also TDib::Info

IsOK

```
bool IsOK() const;
```

Returns **false** if *Info* is 0, otherwise returns **true**. If there is a problem with the construction of the DIB, memory is freed and *Info* is set to 0. Therefore, using *Info* is a reliable way to determine if the DIB is constructed correctly.

See also TDib constructors, TDib::Info

IsPM

```
bool IsPM() const;
```

Returns **true** if *IsCore* is **true**; that is, if the DIB is an old-style PM DIB using core headers. Otherwise returns **false**.

See also TDib::IsCore

MapColor

```
int MapColor(TColor fromColor, TColor toColor, bool doAll = false);
```

Maps the *fromColor* to the *toColor* in the current palette of this DIB.

Returns the palette entry for the given color. Returns the palette entry for the *toColor* argument.

See also TDib::GetColor, TColor, TDib::SetColor, TDib::FindColor

MapIndex

```
int MapIndex(uint16 fromIndex, Word toIndex, bool doAll = false);
```

Maps the *fromIndex* to the *toIndex* in the current palette of this DIB.

Returns the palette entry for the *toIndex* argument.

See also TDib::FindIndex, TDib::SetIndex, TDib::GetIndex

MapUIColors

```
void MapUIColors(uint mapColors, TColor* bkColor = 0 )
```

Maps the UI colors to the value specified in the parameter, *mapColors*, which is one of the values of the *Map* enum. Use this function to get the colors for the face shading on pushbuttons, the highlighting for a selected control button, the text on pushbuttons, the shade of the window frame and the background color.

See also TDib::Map enum

NumColors

```
long NumColors() const;
```

Returns *NumClrs*, the number of colors in this DIB's palette.

See also TDib::Info

NumScans

```
uint16 NumScans() const;
```

Returns the number of scans in this DIB.

See also TDib::StartScan

operator <<

```
TClipboard& operator <<(TClipboard& clipboard, TDib& dib);
```

Writes the given *dib* to the given *clipboard*. Returns a reference to the resulting Clipboard, allowing the normal chaining of <<.

See also TClipboard

operator ==

```
bool operator ==(const TDib& other) const;
```

Compares two handles and returns **true** if this DIB's handle equals the other (*other*) DIB's handle.

See also Tdib::Handle

SetColor

```
void SetColor(int entry, TColor color);
```

Sets the given color for the given palette entry.

See also TDib::GetColor, TColor, TDib::MapColor, TDib::FindColor

SetIndex

```
void SetIndex(int entry, uint16 index);
```

Sets the given index for the given entry.

See also TDib::GetIndex, TDib::FindIndex, TDib::MapIndex

Size

```
TSize Size() const;
```

Returns *TSize(W,H)*, the size of this DIB.

See also TDib::Info, TSize

StartScan
uint16 StartScan() const;
Returns the DIB's starting scan line.

See also TDib::numScans

ToClipboard
void ToClipboard(TClipboard& clipboard);
Puts this DIB onto the specified Clipboard.

See also TClipboard

operator TRgbQuad()
operator const TRgbQuad far*() const;
operator TRgbQuad far*() const;
Typecasts this DIB by returning a pointer to its colors structure.

See also TDib::GetColors, TRgbQuad

Usage
uint16 Usage() const;
Returns the *Mode* for this DIB. This value tells *GDI* how to treat the color table.

See also TDib::Mode

Width
int Width() const;
Returns *W*, the DIB width.

See also TDib::Info

WriteFile
bool WriteFile(const char* filename);
Returns **true** if the call is successful; otherwise returns **false**.

Protected data members

Bits
void HUGE* Bits;
Bits points into the block of memory pointed to by *Info*.

See also TDib::GetBits

H
int H;
The height of this DIB in pixels.

See also TDib::Height, TDib::Size, TDib::NumScans

Info
BITMAPINFO far* Info;
Locked global allocated block.

See also TDib::GetInfo

IsCore
bool IsCore;
Set **true** if this DIB is an old-style PM DIB using core headers; otherwise, set **false**.

See also TDib::isPM

IsResHandle
bool IsResHandle;
Set **true** if this DIB is using a resource handle; otherwise, set **false**.

Mode
uint16 Mode;
If *Mode* is DIB_RGB_*Colors*, the color table contains 4-byte RGB entries. If *Mode* is *DIB_PAL_COLORS*, the color table contains 2-byte indexes into some other palette (such as the system palette). Because either of these two cases might exist, two versions of certain functions (such as *GetColors* and *GetIndices*) are required.

See also TDib::GetColors, TDib::GetIndices, TDib::Usage

NumClrs
long NumClrs;
The number of colors associated with this DIB.

See also TDib::NumColors

W
int W;
The width of this DIB in pixels.

See also TDib::Width, TDib::Size

Protected member functions

InfoFromHandle
void InfoFromHandle();
Locks this DIB's handle and extracts the remaining data member values from the DIB header.

See also TDib::GetInfoHeader

LoadFile
bool LoadFile(const char* name);
Loads this DIB from the given file name. Returns **true** if the call is successful; otherwise returns **false**.

See also TDib::LoadResource, TDib constructors

LoadResource
bool LoadResource(HINSTANCE instance, TResID resID);
Loads this DIB from the given resource and returns true if successful.

See also TDib::LoadFile, TDib constructors

Read

bool Read(TFile& file, long offBits = 0);

Reads data to this DIB, starting at offset *offBits*, from any file, BMP, or resource. Returns **true** if the call is successful; otherwise returns **false**.

See also TDib::LoadFile

TDibDC Class dc.h

A DC class derived from *TDC*, *TDibDC* provides access to device-independent bitmaps (DIBs).

Public constructors

Constructor

TDibDC(const TDib& dib);

Creates a *TDibDC* object with the data provided by the given *TDib* object.

See also classTDib, TDC::TDC

TDocManager class docmanag.h

TDocManager creates a document manager object that manages the list of current documents and registered templates, handles standard file menu commands, and displays the user-interface for file and view selection boxes. To provide support for documents and views, an instance of *TDocManager* must be created by the application and attached to the application.

The document manager normally handles events on behalf of the documents by using a response table to process the standard CM_FILENEW, CM_FILEOPEN, CM_FILECLOSE, CM_FILESAVE, CM_FILESAVEAS, CM_FILEREVERT, CM_FILEPRINT, CM_FILEPRINTERSETUP, and CM_VIEWCREATE. and CM_VIEWCREATE File menu commands. In response to a CM_FILENEW or a CM_FILEOPEN command, the document manager creates the appropriate document based on the user's selections. In response to the other commands, the document manager determines which of the open documents contains the view associated with the window that has focus. The menu commands are first sent to the window that is in focus and then through the parent window chain to the main window and finally to the application, which forwards the commands to the document manager.

When you create a *TDocManager* or a derived class, you must specify that it has either a multi-document (*dmMDI*) or single-document (*dmSDI*) interface. In addition, if you want the document manager to handle the standard file commands, you must OR *dmMDI* or *dmSDI* with *dmMenu*.

You can also enable or disable the document manager menu options by passing *dmSaveEnable* or *dmNoRevert* in the constructor. If you want to enable the File | Save menu option if the document is unmodified, pass the *dmSaveEnable* flag in the

constructor. To disable the "Revert to Saved" menu option, pass *dmNoRevert* in the constructor.

When the application directly creates a new document and view, it can attach the view to its frame window, create MDI children, float the window, or create a splitter. However, when the document manager creates a new document and view from the File | Open or File | New menu selection, the application doesn't control the process. To give the application control, the document manager sends messages after the new document and view are successfully created. Then, the application can use the information contained in the template to determine how to install the new document or view object.

Public constructor and destructor

Constructors

Form 1 **TDocManager(int mode, TDocTemplate*& templateHead = DocTemplateStaticHead);**

Constructs a *TDocManager* object that supports either single (SDI) or multiple (MDI) open documents depending on the application. *mode* is set to either *dmMenu*, *dmMDI*, *dmSDI*, *dmSaveEnable*, or *dmNoRevert*. To install the standard *TDocManager* File menu commands, you must OR *dmMDI* or *dmSDI* with *dmMenu*. For example,

```
DocManager = new TDocManager(DocMode | dmMenu);
```

The document manager can then use its menu and response table to handle these events. If you do not specify the *dmMenu* parameter, you must provide the menu and functions to handle these commands. However, you can still use your application object's *DocManager* data member to access the document manager's functions.

Form 2 **TDocManager(int mode, TApplication* app, TDocTemplate*& templateHead = DocTemplateStaticHead);**

The constructor performs the same operations as the first constructor. The additional *app* parameter, however, points to the application associated with this document.

Destructor

virtual ~TDocManager();

Destroys a *TDocManager* object removes attached documents templates. The constructor resets *TDocTemplate::DocTemplateStaticHead* so that it points to the head of the static template list.

See also dmxxxx document manager mode constants

Public data members

DocList

TDocument::List DocList;

Holds the list of attached documents or 0 if no documents exist.

Public member functions

AttachTemplate

void AttachTemplate(TDocTemplate&);

Inserts a template into the chain of templates.

CmFileClose

virtual void CmFileClose();

Responds to a file close message. Tests to see if the document has been changed since it was last saved, and if not, prompts the user to confirm the save operation.

CmFileNew

virtual void CmFileNew();

Calls *CreateAnyDoc* with no path specified.

See also TDocManager::CreateAnyDoc, dtxxxx document template constants

CmFileOpen

virtual void CmFileOpen();

Lets the user select a registered template from the list displayed in the dialog box. Calls *CreateAnyDoc*.

See also TDocManager::CreateAnyDoc

CmFileRevert

virtual void CmFileRevert();

Reverts to the previously saved document. Does not revert if the document has not been changed since last save; that is, if the document's *IsDirty* function returns **false**.

CmFileSave

virtual void CmFileSave();

Responds to a file save message. Sets *doc* to the current document. *CmFileSave* checks *IsDirty* only if the *dmSaveEnable flag* was not specified. Calls *PostDocError* with IDS_NOTCHANGED if *dmSaveEnable* was NOT specified and *IsDirty* returns **false**.

See also IDS_xxxx Document String ID constants

CmFileSaveAs

virtual void CmFileSaveAs();

Prompts the user to enter a new name for the document and saves the document to that file.

CmViewCreate

virtual void CmViewCreate();

Creates a document view based on the view name of the current document. If more than one template exists for the document, *CmViewCreate*allows the user to select the type of view from the template list.

CreateAnyDoc

virtaul TDocument* CreateAnyDoc(const char far* path, long flags= 0);

Creates a document based on the directory path and the specified template. *flags*, one of the document template constants, determines how the document template is created. If *path* is 0 and this is not a new document (the flag *dtNewDoc* is not set), it displays a dialog box. If *path* is 0, *dtNewDoc* is not set, and more than one template exists, it displays a dialog box and a list of templates.

See also TDocTemplate::CreateDoc, dtxxxx document template constants

CreateAnyView

virtual TView* CreateAnyView(TDocument& doc, long flags= 0);

Creates a document view based on the directory path and specified template. *flags*, one of the document template constants, determines how the document template is created.

See also TDocument, TDocTemplate::CreateView, dtxxxx document templateconstants

CreateDoc

TDocument* CreateDoc(TDocTemplate* tpl, const char far *, TDocument* parent, long flags = 0);

CreateDoc creates a document based on the directory path and the specified template. *flags* contains one of the document template constants that determines how the documentis created.

See also TDocument, TDocTemplate::CreateView, dtxxxx document template constants

CreateView

TView* CreateView(TDocument* doc);

CreateView creates a view of the specified document.

See also TDocTemplate::CreateView, dtxxxx document template constants

DeleteTemplate

void DeleteTemplate(TDocTemplate&);

Removes a template from the list of templates attached to the document.

See also TDocManager::RefTemplate

EvCanClose

bool EvCanClose();

Checks to see if all child documents can be closed before closing the current document. If any child returns **false**, returns **false** and aborts the process. If all children return **true**, *EvCanClose* calls *TDocManager::FlushDoc* for each document. If *FlushDoc* finds that the document is dirty, it displays a message asking the user to save the document, discard any changes, or cancel the operation. If the document is not dirty and *EvCanClose* returns true, *EvCanClose* returns **true**.

See also TApplication::CanClose, TDocManager::FlushDoc

EvPreProcessMenu

void EvPreProcessMenu(HMENU hmenu);

Called from *MainWindow*, *EvPreProcessMenu* loads and deletes a menu at the position specified by MF_POSITION or MF_POPUP. Your application can call *EvPreProcessMenu* to process the main window's menu before it is displayed.

See also TApplication::PreProcessMenu

EvWakeUp

void EvWakeUp();

Used only after streaming in the doc manager, *EvWakeUp* allows for the windows to be created after the streaming has occurred.

FindDocument

TDocTemplate* FindDocument(const char far* path);

MatchDocument returns the first document whose pattern matches the given file name. If no document is compatible with the supplied file name, or if the document is open already, it returns 0.

See also TDocTemplate

FlushDoc

virtual bool FlushDoc(TDocument& doc);

Updates the document with any changes and prompts the user for confirmation of updates.

See also TDocument

GetApplication

TApplication* GetApplication();

Returns the current application.

See also TApplication

GetCurrentDoc

virtual TDocument* GetCurrentDoc();

Calls *TWindow::GetFocus* to determine the window with the focus. Searches the list of documents and returns the document that contains the view with the focus. Returns 0 if no document has a view with focus.

See also TDocument

GetNextTemplate

TDocTemplate* GetNextTemplate(TDocTemplate* tpl);

Returns the next document template.

See also TDocTemplate

InitDoc

TDocument* InitDoc(TDocument* doc, const char far* path, long flags)

Initializes the documents, directory path for the document, and *dtXXXX* document flag values (such as *dtNewDoc*) used to create document templates.

See also TDocTemplate, dt Documentview Constants

IsFlagSet

bool IsFlagSet(int Flag);

Returns true if the *dtXXXX* document template constant specified in *Flag* is set.

See also dt Documentview Constants

MatchTemplate

TDocTemplate* MatchTemplate(const char far* path);

MatchTemplate returns the first registered template whose pattern matches the given file name. If no template is compatible with the supplied file name, or if the template is open already, it returns 0.

See also TDocTemplate

PostDocError

virtual uint PostDocError(TDocument& doc, uint sid, uint choice = MB_OK);

Displays a message box with the error message passed as a string resource ID in *sid*. By default, the message box contains either an OK pushbutton or a question mark icon. If an error message can't be found, *PostDocError* displays a "Message not found" message. *choice* can be one or more of the *MB_Xxxx message* style constants. This function can be overridden.

See also TDocument::PostError, MB_Xxxx Message Constants

PostEvent

Form 1 virtual void PostEvent(int id, TDocument& doc);

If the current document changes, posts a WM_OWLDOCUMENT message to indicate a change in the status of the document.

Form 2 virtual void PostEvent(int id, TView& view);

If the current view changes, posts a WM_OWLVIEW message to indicate a change in the status of the view.

See also TDocument, TView

RefTemplate

void RefTemplate(TDocTemplate&);

Adds a template to the list of templates attached to the document.

See also TDocManager::UnRefTemplate, TDocTemplate

SelectAnySave

virtual TDocTemplate* SelectAnySave(TDocument& doc, bool samedoc = true);

Selects a registered template to save with this document.

See also TDocTemplate, TDocument

SelectSave

bool SelectSave(TDocument& doc);

Prompts the user to select a file name for the document. Filters out read-only files.

See also TDocTemplate::SelectSave

UnRefTemplate

void UnRefTemplate(TDocTemplate&);

Removes a template to the list of templates attached to the document.

See also TDocManager::RefTemplate, TDocTemplate

Protected member functions

SelectDocPath

virtual int SelectDocPath(TDocTemplate** tpllist, int tplcount, char far* path, int buflen, long flags,
  bool save=false);

Prompts the user to select one of the templates to use for the file to be opened. Returns the template index used for the selection or 0 if unsuccessful. For a file open operation,

save is **false**. For a file save operation, *save* is **true**. This function can be overridden to provide a customized user-interface.

See also TDocTemplate

SelectDocType

virtual int SelectDocType(TDocTemplate** tpllist, int tplcount);

SelectDocType, which can be overridden, lets the user select a document type from a list of document templates. Returns the template index used for the selection or 0 if unsuccessful.

See also TDocTemplate

SelectViewType

virtual int SelectViewType(TDocTemplate** tpllist, int tplcount);

SelectViewType, which can be overridden, lets the user select a view name for a new view from a list of view names. Returns the template index used for the selection or 0 if unsuccessful.

See also TDocTemplate

Response table entries

Response table entry	Member function
EV_COMMAND(CM_FILECLOSE, CmFileClose)	*CmFileClose*
EV_COMMAND(CM_FILENEW, CmFileNew)	*CmFileNew*
EV_COMMAND(CM_FILEOPEN, CmFileOpen)	*CmFileOpen*
EV_COMMAND(CM_FILEREVERT, CmFileRevert)	*CmFileRevert*
EV_COMMAND(CM_FILESAVE, CmFileSave)	*CmFileSave*
EV_COMMAND(CM_FILESAVEAS, CmFileSaveAs)	*CmFileSaveAs*
EV_COMMAND(CM_VIEWCREATE, CmViewCreate)	*CmViewCreate*
EV_COMMAND_ENABLE(CM_FILECLOSE, CmEnableClose)	*CmEnableClose*
EV_COMMAND_ENABLE(CM_FILENEW, CmEnableNew)	*CmEnableNew*
EV_COMMAND_ENABLE(CM_FILEOPEN, CmEnableOpen)	*CmEnableOpen*
EV_COMMAND_ENABLE(CM_FILEREVERT, CmEnableRevert)	*EmEnableRevert*
EV_COMMAND_ENABLE(CM_FILESAVE, CmEnableSave)	*CmEnableSave*
EV_COMMAND_ENABLE(CM_FILESAVEAS, CmEnableSaveAs)	*CmEnableSaveAs*
EV_COMMAND_ENABLE(CM_VIEWCREATE, CmEnableCreate)	*CmEnableCreate*
EV_WM_CANCLOSE	*EvCanClose*
EV_WM_PREPROCMENU	*EvPreProcessMenu*
EV_WM_WAKEUP	*EvWakeUp*

TDocument class

docview.h

An abstract base class, *TDocument* is the base class for all document objects and serves as an interface between the document, its views, and the document manager

(*TDocManager* class). *TDocument* creates, destroys, and sends messages about the view. For example, if the user changes a document, *TDocument* tells the view that the document has been updated. The DEFINE_DOC_TEMPLATE_CLASS macro associates a document with its views.

In order to send messages to its associated views, the document maintains a list of all the views existing for that document and communicates with the views using ObjectWindows event-handling mechanism. Rather than using the function *SendMessage*, the document accesses the view's event table. The views can update the document's data by calling the member functions of the particular document. Views can also request streams, which are constructed by the document.

Both documents and views have lists of properties for their applications to use. When documents and views are created or destroyed, messages are sent to the application, which can then query the properties to determine how to process the document or view. It is the document manager's responsibility to determine if a particular view is appropriate for the given document.

Because the property attribute functions are virtual, a derived class (which is called first) might override the properties defined in a base class. Each derived class must implement its own property attribute types of either string or binary data. If the derived class duplicates the property names of the parent class, it should provide the same behavior and data type as the parent.

In order to add persistence to documents, *TDocument* contains several virtual functions (for example, *InStream* and *OutStream*) that support streaming. Your derived classes need to override these functions in order to read and write data.

Although documents are usually associated with files, they do not necessarily have to be files; they can also consist of database tables, mail systems, fax or modem transmissions, disk directories, and so on.

Public data members

ChildDoc

```
List ChildDoc;
```

The list of child documents associated with this document.

Tag

```
void far* Tag;
```

Tag holds a pointer to the application-defined data. Typically, you can use *Tag* to install a pointer to your own application's associated data structure. *Tag*, which is initialized to 0 at the time a *TDocument* is constructed, is not used otherwise by the document view classes.

Type definition

TDocProp

```
enum TDocProp{ PrevProperty = 0, DocumentClass, TemplateName, ViewCount, StoragePath, DocTitle,
  NextProperty, };
```

These property values, defined for *TDocument*, are available in classes derived from *TDocument*. PrevProperty and NextProperty are delimiters for every document's property list.

Public constructor and destructor

Constructor

```
TDocument(TDocument* parent = 0);
```

Although you don't create a *TDocument* object directly, you must call the constructor when you create a derived class. parent points to the parent of the new document. If no parent exists, parent is 0.

Destructor

```
virtual ~TDocument();
```

Deletes a *TDocument* object. Normally, *Close* is called first. *TDocument*'s destructor destroys all children and closes all open streams. If this is the last document that used the template, it closes the object's template and any associated views, deletes the object's stream, and removes itself from the parent's list of children if a parent exists. If there is no parent, it removes itself from the document manager's document list.

See also TDocument::Close

Public member functions

CanClose

```
virtual bool CanClose();
```

Checks to see if all child documents can be closed before closing the current document. If any child returns false, *CanClose* returns false and aborts the process. If all children return **true**, calls *TDocManager::FlushDoc*. If *FlushDoc* finds that the document has been changed but not saved, it displays a message asking the user to either save the document, discard any changes, or cancel the operation. If the document has not been changed and all children's *CanClose* functions return **true**, this *CanClose* function returns **true**.

See also TDocManager::FlushDoc

Close

```
virtual bool Close();
```

Closes the document but does not delete or detach the document. Before closing the document, *Close* checks any child documents and tries to close them before closing the parent document. Even if you write your own *Close* function, call *TDocument*'s version so that all child documents are checked before the parent document is closed.

Commit

```
virtual bool Commit(bool force = false);
```

Saves the current data to storage. When a file is closed, the document manager calls either *Commit* or *Revert*. If *force* is **true**, all data is written to storage. *TDocument*'s *Commit* checks any child documents and commits their changes to storage also. Before the current data is saved, all child documents must return true. If all child documents return

true, *Commit* flushes the views for operations that occurred since the last time the view was checked. Once all data for the document is updated and saved, *Commit* returns true.

See also TDocument::Revert

FindProperty

virtual int FindProperty(const char far* name);

Gets the property index, given the property name (*name*). Returns the integer index number that corresponds to the name or 0 if the name isn't found in the list of properties.

See also pfxxxx property attribute constants, TDocument::PropertyName

GetDocManager

TDocManager& GetDocManager();

Returns a pointer to the current document manager.

GetDocPath

const char far* GetDocPath();

Returns the directory path for the document. This might change the SaveAs operation.

GetOpenMode

int GetOpenMode;

Gets the mode and protection flag values for the current document.

See also TDocument::SetOpenMode

GetParentDoc

TDocument* GetParentDoc():

Returns the parent document of the current document or 0 if there is no parent document.

GetProperty

virtual int GetProperty(int index, void far* dest, int textlen=0);

Returns the total number of properties for this document where *index* is the property index, *dest* contains the property data, and *textlen* is the size of the array. If *textlen* is 0, property data is returned as binary data; otherwise, property data is returned as text data.

See also pfxxxx property attribute constants, TDocument::SetProperty

GetTemplate

TDocTemplate* GetTemplate();

Gets the template used for document creation. The template can be changed during a SaveAs operation.

GetTitle

CONST CHAR FAR* GetTitle();

Returns the title of the document.

HasFocus

bool HasFocus(HWND hwnd);

Used by the document manager, *HasFocus* returns **true** if this document's view has focus. *hwnd* is a handle to the document to determine if the document contains a view with a focus.

The view associated with this document is the active view.

InitDoc

```
virtual bool InitDoc();
```

TDocument's *InitDoc* is a virtual method that is overridden by *TOleDocument*'s *InitDoc*. You can use this function to prepare the document before the *dnCreate* event, which indicates that the document has been created, is posted and before the view is constructed.

See also TOleDocument::InitDoc, dnxxxx document message enum

InStream

```
virtual TInStream* InStream(int mode, const char far* strmId=0);
```

Generic input for the particular storage medium, *InStream* returns a pointer to a *TInStream*. *mode* is a combination of the *ios* bits defined in iostream.h. See the document open mode constants for a list of the open modes. Used for documents that support named streams, *strmId* is a pointer to the name of a stream. Override this function to provide streaming for your document class.

See also TDocument::OutStream

IsDirty

```
virtual bool IsDirty();
```

Returns **true** if the document or one of its views has changed but has not been saved.

IsEmbedded

```
virtual bool IsEmbedded();
```

Returns **true** if the document is embedded in an OLE2 container.

See also TDocument::SetEmbedded

IsOpen

```
virtual bool IsOpen();
```

Checks to see if the document has any streams in its stream list. Returns **false** if no streams are open; otherwise, returns **true**.

NextStream

```
TStream* NextStream(const TStream* strm);
```

Gets the next entry in the stream. Holds 0 if none exists.

NextView

```
TView* NextView(const TView* view);
```

Gets the next view in the list of views. Holds 0 if none exists.

NotifyViews

```
bool NotifyViews(int event, long item=0, TView* exclude=0);
```

Notifies the views of the current document and the views of any child documents of a change, In contrast to *QueryViews*, *NotifyViews* sends notification of an event to all views and returns **true** if any views returned a true result. The event, EV_OWLNOTIFY, is

sent with an event code, which is private to the particular document and view class, and a **long** argument, which can be cast appropriately to the actual type passed in the argument of the response function.

See also TDocument::QueryViews

Open

virtual bool Open(int mode, const char far* path = 0);

Opens the document using the path specified by *DocPath*. Sets *OpenMode* to *mode*. *TDocument*'s version always returns true and actually performs no actions. Other classes override this function to open specified file documents and views.

See also TFileDocument::Open

OutStream

virtual TOutStream* OutStream(int mode, const char far* strmId = 0);

Generic output for the particular storage medium, *OutStream* returns a pointer to a *TOutStream*. *mode* is a combination of the *ios* bits defined in iostream.h. Used for documents that support named streams, *strmId* points to the name of the stream. *TDocument*'s version always returns 0. Override this function to provide streaming for your document class.

See also TDocument::InStream

PostError

virtual uint PostError(uint sid, uint choice = MB_OK);

Posts the error message passed as a string resource ID in *sid*. *choice* is one or more of the MB_Xxxx style constants.

See also TDocManager::PostDocError, MB_Xxxx message constants

PropertyCount

virtual int PropertyCount();

Gets the total number of properties for the *TDocument* object. Returns *NextProperty* –1.

See also pfxxxx property attribute constants

PropertyFlags

virtual int PropertyFlags(int index);

Returns the attributes of a specified property given the index (*index*) of the property whose attributes you want to retrieve.

See also pfxxxx property attribute constants, TDocument::FindProperty, TDocument::PropertyName

PropertyName

virtual const char* PropertyName(int index);

Returns the name of the property given the index value (*index*).

See also pfxxxx property attribute constants, TDocument::FindProperty

QueryViews

TView* QueryViews(int event, long item=0, TView* exclude=0);

Queries the views of the current document and the views of any child documents about a specified event, but stops at the first view that returns **true**, In contrast to *NotifyViews*, *QueryViews* returns a pointer to the first view that responded to an event with a **true** result. The event, EV_OWLNOTIFY, is sent with an event code (which is private to the particular document and view class) and a **long** argument (which can be cast appropriately to the actual type passed in the argument of the response function).

See also TDocument::NotifyViews

Revert

virtual bool Revert(bool clear = false);

Performs the reverse of *Commit* and cancels any changes made to the document since the last *commit*. If *clear* is **true**, data is not reloaded for views. *Revert* also checks all child documents and cancels any changes if all children return **true**. When a file is closed, the document manager calls either *Commit* or *Revert*. Returns **true** if the operation is successful.

See also TDocument::Commit

RootDocument

virtual TDocument& RootDocument();

Returns the **this** pointer as the root document.

SetDocManager

void SetDocManager(TDocManager& dm);

Sets the current document manager to the argument *dm*.

SetDocPath

virtual bool SetDocPath(const char far* path);

Sets the document path for Open and Save operations.

SetEmbedded

virtual bool SetEmbedded();

SetEmbedded marks the document as being embedded in an OLE2 container. Typically, this happens when the server is created and when the factory template class create the component.

See also TDocument::IsEmbedded

SetOpenMode

void SetOpenMode(int mode);

Sets the mode and protection flag values for the current document.

See also TDocument::GetOpenMode

SetProperty

virtual bool SetProperty(int index, const void far* src);

Sets the value of the property, given *index*, the index value of the property, and *src*, the data type (either binary or text) to which the property must be set.

See also pfxxxx property attribute constants, TDocument::GetProperty

SetTemplate
bool SetTemplate(TDocTemplate* tpl);
Sets the document template. However, if the template type is incompatible with the file, the document manager will refuse to save the file as this template type.

SetTitle
virtual void SetTitle(const char far* title);
Sets the title of the document.

Protected data members

DirtyFlag
bool DirtyFlag;
Indicates that unsaved changes have been made to the document. Views can also independently maintain their local disk status.

Embedded
bool Embedded;
Indicates whether the document is embedded.

Protected member functions

AttachStream
virtual void AttachStream(TStream& strm);
Called from *TStream*'s constructor, *AttachStream* attaches a stream to the current document.

DetachStream
virtual void DetachStream(TStream& strm);
Called from *TStream*'s destructor, *DetachStream* detaches the stream from the current document.

TDocument::List class

docview.h

The *TDocument::List* nested class encapsulates the chain of documents. It allows addition, removal, and destruction of documents from the document list.

Public constructor and destructor

Constructor
List();
Constructs a *TDocument::List* object.

Destructor
~List();
Destroys the *TDocument::List* object.

Public member functions

Destroy
void Destroy();
Deletes all documents.

Insert
bool Insert(TDocument* doc);
Inserts a new document into the document list. Fails if the document already exists.

Next
TDocument* Next(const TDocument* doc);
If the *doc* parameter is 0, *Next* returns the first document in the list of documents.

Remove
bool Remove(TDocument* doc);
Removes a document from the document list.

TEdgeConstraint struct

layoutco.h

TEdgeConstraint adds member functions that set edge (but not size) constraints. *TEdgeConstraint* always places your window one pixel above the other window and then adds margins.

For example, if the margin is 4, *TEdgeConstraint* places your window 5 pixels above the other window. The margin, which does not need to be measured in pixels, is defined using the units specified by the constraint. Therefore, if the margin is specified as 8 layout units (which are then converted to 12 pixels), your window would be placed 13 pixels above the other window.

See also
TLayoutConstraint struct, TEdgeOrSizeConstraint struct, TLayoutWindow (layout constraints example)

Public member functions

Above
void Above (TWindow *sibling, int margin = 0)
Positions your window above a sibling window. You must specify the sibling window and an optional margin between the two windows. If no margin is specified, *Above* sets the bottom of one window one pixel above the top of the other window.

See also TEdgeConstraint::Below

Absolute
void Absolute (TEdge edge, int value)
Sets an edge of your window to a fixed value.

See also TEdgeConstraint::PercentOf, TEdgeOrSizeConstraint::Absolute

Below

void Below(TWindow *sibling, int margin = 0);

Positions your window with respect to a sibling window. You must specify the sibling window and an optional margin between the two windows. If no margin is specified, *Below* sets the top of one window one pixel below the bottom of the other window.

See also TEdgeConstraint::Above

LeftOf

void LeftOf (TWindow *sibling, int margin = 0)

Positions one window with respect to a sibling window. You can specify the sibling window and an optional margin between the two windows.

See also TEdgeConstraint::RightOf

PercentOf

void PercentOf (TWindow *otherWin, TEdge edge, int percent)

Specifies that the edge of one window indicated in *edge* should be a percentage of the corresponding edge of another window (*otherWin).*

See also TEdgeConstraint::Absolute, TEdgeOrSizeConstraint::PercentOf

RightOf

void RightOf (TWindow *sibling, int margin = 0)

Positions one window with respect to a sibling window. You can specify the sibling window and an optional margin between the two windows.

See also TEdgeConstraint::LeftOf

SameAs

void SameAs (TWindow *otherWin, TEdge edge)

Sets the edge of your window indicated by *edge* equivalent to the corresponding edge of the window in *otherWin.*

See also TEdgeConstraint::Set, TEdgeOrSizeConstraint::SameAs

Set

void Set(TEdge edge, TRelationship rel, TWindow *otherWin, TEdge otherEdge, int value = 0);

Used for setting arbitrary edge constraints, *Set* specifies that your window's edge should be of a specified relationship to *otherWin's* specified edge.

See also TEdgeConstraint::SameAs

TEdgeOrSizeConstraint struct — layoutco.h

Derived from *TEdgeConstraint, TEdgeOrSizeConstraint* is a template class that supports size constraints in addition to all the operations that *TEdgeConstraint* provides. The width or height is specified in the template instantiation of this class. There are two versions of each member function: one sets both edge and size constraints; the other sets only edge constraints.

Public member functions

Absolute

Form 1 void Absolute (int value)

Sets the width or height of your window to a fixed value.

Form 2 void Absolute (TEdge edge, int value)

Used to determine edge constraints only, *Absolute* sets the edge of your window to a fixed value.

See also TEdgeConstraint::Absolute

PercentOf

Form 1 void PercentOf (TWindow *otherWin, int percent, TWidthHeight otherWidthHeight = widthOrHeight)

Although a window's width or height defaults to being a percentage of the sibling or parent window's corresponding dimension, it can also be a percentage of the sibling or parent's opposite dimension. For example, one window's width can be 50% of the parent window's height.

Form 2 void PercentOf (TWindow *otherWin, TEdge edge, int percent)

Used to determine edge constraints only, *PercentOf* specifies that the edge of one window indicated in *edge* should be a percentage of the corresponding edge of another window (*otherWin*.

See also TEdgeOrSizeConstraint::Absolute, TEdgeConstraint::PercentOf

SameAs

Form 1 void SameAs (TWindow *otherWin, TWidthHeight otherWidthHeight = widthOrHeight, int value = 0)

Although a window's width or height defaults to being the same as the sibling or parent window's corresponding dimension, it can be the same of the sibling's or parent's opposite dimension. For example, one window's width can be the same as the parent window's height.

Form 2 void SameAs (TWindow *otherWin, TEdge edge)

Used to determine edge constraints only, *SameAs* sets the edge of one window the same as the corresponding edge of the other window specified in *otherWin*.

See also TEdgeOrSizeConstraint::PercentOf, TEdgeConstraint::SameAs

TEdit class

edit.h

A *TEdit* is an interface object that represents an edit control interface element. A *TEdit* object must be used to create an edit control in a parent *TWindow*. A *TEdit* can be used to facilitate communication between your application and the edit controls of a *TDialog*. This class is streamable.

There are two styles of edit control objects: single-line and multiline. Multiline edit controls allow editing of multiple lines of text.

The position of the first character in an edit control is zero. For a multiline edit control, the position numbers continue sequentially from line to line; line breaks count as two characters.

Most of *TEdit's* member functions manage the edit control's text. *TEdit* also includes some automatic member response functions that respond to selections from the edit control's parent window menu, including cut, copy, and paste. Two important member functions inherited from *TEdit's* base class (*TStatic*) are *GetText* and *SetText*.

Public constructors

Constructors

Form 1 `TEdit(TWindow* parent, int Id, const char far *text, int x, int y, int w, int h, uint textLen = 0, bool multiline = false, TModule* module = 0);`

Constructs an edit control object with a parent window (*parent*). Sets the creation attributes of the edit control and fills its *Attr* data members with the specified control ID (*Id*), position (*x, y*) relative to the origin of the parent window's client area, width (*w*), and height (*h*).

If text buffer length (*textLen*) is 0 or 1, there is no explicit limit to the number of characters that can be entered. Otherwise *textLen* – 1 characters can be entered. By default, initial text (*text*) in the edit control is left-justified and the edit control has a border. Multiline edit controls have horizontal and vertical scroll bars.

Form 2 `TEdit(TWindow* parent, int resourceID, uint textLen = 0, TModule* module = 0);`

Constructs a *TEdit* object to be associated with an edit control of a *TDialog*. Invokes the *TStatic* constructor with identical parameters. The *resourceID* parameter must correspond to an edit resource that you define. Enables the data transfer mechanism by calling *EnableTransfer*.

See also TStatic::TStatic

Public member functions

CanUndo

`bool CanUndo();`

Returns **true** if it is possible to undo the last edit.

See also TEdit::Undo

Clear

`void Clear();`

Overrides *TStatic's* virtual member function and clears all text.

ClearModify

`void ClearModify();`

Resets the change flag of the edit control causing *IsModified* to return **false**. The flag is set when text is modified.

See also TEdit::IsModified

Copy

`void Copy();`

Copies the currently selected text into the Clipboard.

Cut
void Cut();
Deletes the currently selected text and copies it into the Clipboard.

DeleteLine
bool DeleteLine(int lineNumber);
Deletes the text in the line specified by *lineNumber* in a multiline edit control. If –1 passed, deletes the current line. *DeleteLine* does not delete the line break and affects no other lines. Returns true if successful. Returns false if *lineNumber* is not –1 and is out of range or if an error occurs.

DeleteSelection
bool DeleteSelection();
Deletes the currently selected text, and returns **false** if no text is selected.

DeleteSubText
bool DeleteSubText(uint startPos, uint endPos);
Deletes the text between the starting and ending positions specified by *startPos* and *endPos*, respectively. *DeleteSubText* returns **true** if successful.

EmptyUndoBuffer
void EmptyUndoBuffer();
If an operation inside the edit control can be undone, the edit control undo flag is set. *EmptyUndoBuffer* resets or clears this flag.

FormatLines
void FormatLines(bool addEOL);
Indicates if the end-of-line characters (carriage return, linefeed) are to be added or removed from text lines that are wordwrapped in a multiline edit control. Returns **true** if these characters are placed at the end of wordwrapped lines or **false** if they are removed.

GetFirstVisibleLine
int GetFirstVisibleLine() const;
Indicates the topmost visible line in an edit control. For single-line edit controls, the return value is 0. For multiline edit controls, the return value is the index of the topmost visible line.

GetHandle
HLOCAL GetHandle() const;
Returns the data handle of the buffer that holds the contents of the control window.

This function is obsolete, and is not available under Presentation Manager.

See also TEdit::SetHandle

GetLine
bool GetLine(char far* str, int strSize, int lineNumber);
Retrieves a line of text (whose line number is supplied) from the edit control and returns it in *str* (NULL-terminated). *strSize* indicates how many characters to retrieve. *GetLine* returns **false** if it is unable to retrieve the text or if the supplied buffer is too small.

See also TStatic::GetText, TEdit::GetNumLines, TEdit::GetLineLength

GetLineFromPos

```
int GetLineFromPos(uint charPos);
```

From a multiline edit control, returns the line number on which the character position specified by *charPos* occurs. If *charPos* is greater than the position of the last character, the number of the last line is returned. If *charPos* is –1, the number of the line that contains the first selected character is returned. If there is no selection, the line containing the caret is returned.

GetLineIndex

```
uint GetLineIndex(int lineNumber);
```

In a multiline edit control, *GetLineIndex* returns the number of characters that appear before the line number specified by *lineNumber*. If *lineNumber* is –1, *GetLineIndex* returns the number of the line that contains the caret is returned.

GetLineLength

```
int GetLineLength(int lineNumber);
```

From a multiline edit control, *GetLineLength* returns the number of characters in the line specified by *lineNumber*. If it is –1, the following applies: if no text is selected, *GetLineLength* returns the length of the line where the caret is positioned; if text is selected on the line, *GetLineLength* returns the line length minus the number of selected characters; if selected text spans more than one line, *GetLineLength* returns the length of the lines minus the number of selected characters.

GetNumLines

```
int GetNumLines();
```

Returns the number of lines that have been entered in a multiline edit control: 1 if the edit control has no text (if it has one line with no text in it), or 0 if there is no text or if an error occurs.

GetPasswordChar

```
uint GetPasswordChar() const;
```

Returns the character to be displayed in place of a user-typed character. When the edit control is created with the ES_PASSWORD style specified, the default display character is an asterisk (*).

See also TEdit::SetPasswordChar

GetRect

```
void GetRect(TRect& frmtRect) const;
```

Gets the formatting rectangle of a multiline edit control.

See also TEdit::SetRect, TEdit::SetRectNP

GetSelection

```
void GetSelection(uint& startPos, uint& endPos);
```

Returns the starting (*startPos*) and ending (*endPos*) positions of the currently selected text. By using *GetSelection* in conjunction with *GetSubText*, you can get the currently selected text.

See also TEdit::GetSubText

GetSubText

```
void GetSubText(char far* str, uint startPos, uint endPos);
```

Retrieves the text in an edit control from indexes *startPos* to *endPos* and returns it in *str*.

See also TEdit::GetSelection

GetWordBreakProc

```
EDITWORDBREAKPROC GetWordBreakProc() const;
```

Retrieves the current wordwrap function. Returns the address of the wordwrap function defined by the application or 0 if none exists.

See also TEdit::SetWordBreakProc

Insert

```
void Insert(const char far* str);
```

Inserts the text supplied in *str* into the edit control at the current text insertion point (cursor position), and replaces any currently selected text. *Insert* is similar to *Paste*, but does not affect the Clipboard.

See also TEdit::Paste

IsModified

```
bool IsModified();
```

Returns **true** if the user has changed the text in the edit control.

See also TEdit::ClearModify

IsValid

```
bool IsValid(bool reportErr = false);
```

Returns **true** if the contents of the edit control are valid. *reportErr* is **false** so that, by default, *IsValid* doesn't bring up a system or custom message box with an error string created from the default string table.

LockBuffer

```
char far* LockBuffer(uint newsize=0);
```

Locks the edit control's buffer and returns a pointer to the buffer. Passing *newsize* greater than 0 causes the buffer to be resized to *newsize*. You must call *Unlock* when you are finished.

See also TEdit::UnLockBuffer

Paste

```
void Paste();
```

Inserts text from the Clipboard into the edit control at the current text insertion point (cursor position).

See also TEdit::CMEditPaste

Scroll

```
void Scroll(int horizontalUnit, int verticalUnit);
```

Scrolls a multiline edit control horizontally and vertically using the numbers of characters specified in *horizontalUnit* and *verticalUnit*. Positive values result in scrolling to the right or down in the edit control, and negative values result in scrolling to the left or up.

Search

int Search(uint startPos, const char far* text, bool caseSensitive=false, bool wholeWord=false, bool up=false);

Performs either a case-sensitive or case-insensitive search for the supplied text. If the text is found, the matching text is selected, and *Search* returns the position of the beginning of the matched text. If the text is not found in the edit control's text, *Search* returns –1. If –1 is passed as *startPos*, then the search starts from either the end or the beginning of the currently selected text, depending on the search direction.

SetHandle

void SetHandle(HLOCAL localMem);

Sets a handle to the text buffer used to hold the contents of a multiline edit control.

This function is obsolete, and is not available under Presentation Manager.

See also TEdit::GetHandle

SetPasswordChar

void SetPasswordChar(uint ch);

SetPasswordChar sets the character to be displayed in place of a user-typed character. When the ES_PASSWORD style is specified, the default display character is an asterisk (*).

See also TEdit::GetPasswordChar

SetReadOnly

void SetReadOnly(bool readOnly);

Sets the edit control to be read-only or read-write.

SetRect

void SetRect(const TRect& frmtRect);

Sets the formatting rectangle for a multiline edit control.

See also TEdit::GetRect, TEdit::SetRectNP

SetRectNP

void SetRectNP(const TRect& frmtRect);

Sets the formatting rectangle for a multiline edit control. Unlike *SetRect*, *SetRectNP* does not repaint the edit control.

See also TEdit::GetRect, TEdit::SetRect

SetSelection

bool SetSelection(uint startPos, uint endPos);

Forces the selection of the text between the positions specified by *startPos* and *endPos*, but not including the character at *endPos*.

SetTabStops

void SetTabStops(int numTabs, const int far* tabs);

Sets the tab stop positions in a multiline edit control.

SetValidator

void SetValidator(TValidator* validator);

Establishes the validator object for the edit control.

SetWordBreakProc

void SetWordBreakProc(EDITWORDBREAKPROC proc);

In a multiline edit control, *SetWordBreakProc* indicates that an application-supplied word-break function has replaced the default word-break function. The application-supplied word-break function might break the words in the text buffer at a character other than the default blank character.

See also TEdit::GetWordBreakProc

Transfer

uint Transfer(void* buffer TTransferDirection direction);

Transfers information for *TEdit* controls and sends information to the *Validator* if one exists, and if it has the transfer option set. Transfer can perform type conversion when validators are in place, for example, when *TRangeValidator* transfers integers. The return value is the size (in bytes) of the transfer data.

Undo

void Undo();

Undoes the last edit.

See also TEdit::CanUndo, TEdit::CMEditUndo

UnlockBuffer

void UnlockBuffer(const char far* buffer, bool updateHandle=false);

Unlocks a locked edit control buffer. If the contents were changed, *updateHandle* should be **true**.

See also TEdit::LockBuffer

ValidatorError

void ValidatorError();

Handles validation errors that occur as a result of validating the edit control.

Protected data member

Validator

TValidator* Validator;

Points to the validator object constructed in your derived class to validate input text. If no validator exists, *Validator* is zero.

Protected member functions

CanClose

bool CanClose();

Checks to see if all child windows can be closed before closing the current window. If any child window returns **false**, *CanClose* returns **false** and terminates the process. If all child windows can be closed, *CanClose* returns **true**.

CmCharsEnable

void CmCharsEnable(TCommandEnabler& commandHandler);

Determines whether the *Clear* menu item is enabled for the currently selected text.

CmEditClear
void CmEditClear();
Automatically responds to a menu selection with a menu ID of CM_EDITCLEAR by calling *Clear*.

See also TStatic::Clear

CmEditCopy
void CmEditCopy();
Automatically responds to a menu selection with a menu ID of CM_EDITCOPY by calling *Copy*.

See also TEdit::Copy

CmEditCut
void CmEditCut();
Automatically responds to a menu selection with a menu ID of CM_EDITCUT by calling *Cut*.

See also TEdit::Cut

CmEditDelete
void CmEditDelete();
Automatically responds to a menu selection with a menu ID of CM_EDITDELETE by calling *DeleteSelection*.

See also TEdit::DeleteSelection

CmEditPaste
void CmEditPaste();
Automatically responds to a menu selection with a menu ID of CM_EDITPASTE by calling *Paste*.

See also TEdit::Paste

CmEditUndo
void CmEditUndo();
Automatically responds to a menu selection with a menu ID of CM_EDITUNDO by calling *Undo*.

See also TEdit::Undo

CmModEnable
void CmModEnable(TCommandEnabler& commandHandler);
Determines whether the Undo menu item is enabled for the selected text.

CmPasteEnable
void CmPasteEnable(TCommandEnabler& commandHandler);
Determines whether the Paste menu item is enabled for the selected text.

CmSelectEnable
void CmSelectEnable(TCommandEnabler& commandHandler);

Determines whether the Cut, Copy, or Delete menu items are enabled for the selected text.

ENErrSpace

```
void ENErrSpace();
```

Sounds a beep in response to an error notification message that is sent when the edit control unsuccessfully attempts to allocate more memory.

EvChar

```
void EvChar(uint key, uint repeatCount, uint flags);
```

Validates the text entered into the edit control. If the input is incorrect, the original text is restored. Otherwise, the validated and modified text is placed back into the edit control, so the results of the auto-fill (if any) can be viewed. When *IsValidInput* is called, the *SupressFill* parameter defaults to False, so that the string can be modified.

EvGetDlgCode

```
uint EvGetDlgCode(MSG far* msg);
```

Responds to WM_GETDLGCODE messages that are sent to a dialog box associated with a control. *EvGetDlgCode* allows the dialog manager to intercept a message that would normally go to a control and then ask the control if it wants to process this message. If not, the dialog manager processes the message. The *msg* parameter indicates the kind of message, for example a control, command, or edit message, sent to the dialog box manager.

If the edit control contains valid input, then Tabs are allowed for changing focus and a DLGC_WANTTABS code is returned.

See also DLGC_xxxxdialog control message constants

EvKeyDown

```
void EvKeyDown(uint key, uint repeatCount, uint flags);
```

EvKeyDown translates the virtual key code into a movement. *key* indicates the virtual key code of the pressed key, *repeatCount* holds the number of times the same key is pressed, *flags* contains one of the messages that translates to a virtual key (VK) code for the mode indicators. If the Tab key is sent to the Edit Control, *EvKeyDown* checks the validity before allowing the focus to change.

EvKillFocus

```
void EvKillFocus(HWND hWndGetFocus);
```

In response to a WM_KILLFOCUS message sent to a window that is losing the keyboard, *EvKillFocus* hides and then destroys the caret. *EvKillFocus* validates text whenever the focus is about to be lost and holds onto the focus if the text is not valid. Doesn't kill the focus if another application, a Cancel button, or an OK button (in which case *CanClose* is called to validate text) has the focus.

GetClassName

```
char far* GetClassName();
```

Returns the name of *TEdit's* registration class, "EDIT."

See also TWindow::GetClassName

SetupWindow

void SetupWindow();

If the *textLen* data member is nonzero, *SetupWindow* limits the number of characters that can be entered into the edit control to *textLen* –1.

See also TStatic::TextLen, TWindow::SetupWindow

Response table entries

Response table entry	Member function
EV_COMMAND (CM_EDITCLEAR, CmEditClear)	*CmEditClear*
EV_COMMAND (CM_EDITCOPY, CmEditCopy)	*CmEditCopy*
EV_COMMAND (CM_EDITCUT, CmEditCut)	*CmEditCut*
EV_COMMAND (CM_EDITDELETE, CmEditDelete)	*CmEditClear*
EV_COMMAND (CM_EDITPASTE, CmEditPaste)	*CmEditPaste*
EV_COMMAND (CM_EDITUNDO, CmEditUndo)	*CmEditUndo*
EV_COMMAND_ENABLE(CM_EDITCLEAR, CmCharsEnable)	*CmCharsEnable*
EV_COMMAND_ENABLE(CM_EDITCOPY, CmSelectEnable)	*CmSelectEnable*
EV_COMMAND_ENABLE(CM_EDITCUT, CmSelectEnable)	*CmSelectEnable*
EV_COMMAND_ENABLE(CM_EDITDELETE, CmSelectEnable)	*CmSelectEnable*
EV_COMMAND_ENABLE(CM_EDITPASTE, CmPasteEnable)	*CmPasteEnable*
EV_COMMAND_ENABLE(CM_EDITUNDO, CmModEnable)	*CmModEnable*
EV_NOTIFY_AT_CHILD (EN_ERRSPACE, ENErrSpace)	*ENErrSpace*
EV_WM_CHAR	*EvChar*
EV_WM_GETDLGCODE	*EvGetdlgcode*
EV_WM_KEYDOWN	*EvKeydown*
EV_WM_KILLFOCUS	*EvKillFocus*
EV_WM_CHILDINVALID	*EvChildInvalid*

TEditFile class

editfile.h

TEditFile is a file-editing window. *TEditFile's* data members and member functions manage the file dialog box and automatic responses for file commands such as Open, Read, Write, Save, and SaveAs. *TEditFile* is streamable.

Public constructors and destructor

Constructor

TEditFile(TWindow* ()t = 0, int Id = 0, const char far* text = 0, const char far* fileName = 0, TModule* module = 0);

Constructs a *TEditFile* window given the parent window, resource ID (*Id*), text, file name, and module ID. Sets *Filename* to *fileName*.

Destructor

~TEditFile();

Frees memory allocated to hold the name of the *TEditFile*.

Public data members

FileData
TOpenSaveDialog::TData FileData;
Contains information about the user's file open or save selection.

See also TOpenSaveDialog::TData struct

FileName
char far* FileName;
Contains the name of the file being edited.

Public member functions

CanClear
virtual bool CanClear();
Returns **true** if the text of the associated edit control can be cleared.

CanClose
virtual bool CanClose();
Returns **true** if the edit window can be closed.

CmFileNew
void CmFileNew();
Calls *NewFile* in response to an incoming File New command with a CM_FILENEW command identifier.

See also TEditFile::NewFile

CmFileOpen
void CmFileOpen();
Calls *Open* in response to an incoming File Open command with a CM_FILEOPEN command identifier.

See also TEditFile::Open

CmFileSave
void CmFileSave();
Calls *Save* in response to an incoming File Save command with a CM_FILESAVE command identifier.

See also TEditFile::Save

CmFileSaveAs
void CmFileSaveAs();
Calls *SaveAs* in response to an incoming File SaveAs command with a CM_FILESAVEAS command identifier.

See also TEditFile::SaveAs

NewFile
void NewFile();

Begins the edit of a new file after calling *CanClear* to determine that it is safe to clear the text of the editor.

See also TEditFile::CanClear

Open

Open();

Opens a new file after determining that it is OK to clear the text of the *Editor*. Calls *CanClear*, and if **true** is returned, brings up a file dialog box to retrieve the name of a new file from the user. Calls *ReplaceWith* to pass the name of the new file.

See also TEditFile::CanClear, TEditFile::ReplaceWith

Read

bool Read(const char far* fileName=0);

Reads the contents of a previously specified file into the *Editor*. Returns **true** if read operation is successful.

ReplaceWith

void ReplaceWith(const char far* fileName);

Calls *SetFileName* and *Read* to replace the file currently being edited with a file whose name is supplied.

See also TEditFile::SetFileName, TEditFile::Read

Save

bool Save();

Saves changes to the contents of the *Editor* to a file. If *Editor->IsModified* returns **false**, *Save* returns **true**, indicating there have been no changes since the last open or save.

See also TEditFile::SaveAs, TEditFile::Write

SaveAs

bool SaveAs();

Saves the contents of the *Editor* to a file whose name is retrieved from the user, through execution of a File Save dialog box. If the user selects OK, *SaveAs* calls *SetFileName* and *Write*. Returns **true** if the file was saved.

See also TEditFile::SetFileName, TEditFile::Write

SetFileName

void SetFileName (const char far* fileName);

Sets *FileName* and updates the caption of the window.

Write

bool Write(const char far* fileName=0);

Saves the contents of the *Editor* to a file whose name is specified by *FileName*. Returns **true** if the write operation is successful.

Protected member functions

SetupWindow

void SetupWindow();

Creates the edit window's *Editor* edit control by calling *TEditFile::SetupWindow*. Sets the window's caption to *FileName*, if available; otherwise sets the name to "Untitled."

See also TEditFile::SetFileName, TEditFile::Read

Response table entries

Response table entry	Member function
EV_COMMAND (CM_FILENEW, CmFileNew)	*CmFileNew*
EV_COMMAND (CM_FILEOPEN, CmFileOpen)	*CmFileOpen*
EV_COMMAND (CM_FILESAVE, CmFileSave)	*CmFileSave*
EV_COMMAND (CM_FILESAVEAS, CmFileSaveAs)	*CmFileSaveAs*

TEditSearch class

editsear.h

TEditSearch is an edit control that responds to Find, Replace, and FindNext menu commands. This class is streamable.

Public constructor

Constructor

```
TEditSearch(TWindow* parent = 0, int Id = 0, const char far* text = 0, int x = 0, int y = 0, int w = 0, int h = 0,
  TModule* module = 0);
```

Constructs a *TEditSearch* object given the parent window, resource ID, and character string (*text*).

Public data members

SearchCmd

```
uint SearchCmd;
```

Contains the search command identifier that opened the dialog box if one is open.

SearchData

```
TFindReplaceDialog::TData SearchData;
```

The *SearchData* structure defines the search text string, the replacement text string, and the size of the text buffer.

See also TFindReplaceDialog::TData

SearchDialog

```
TFindReplaceDialog* SearchDialog;
```

Contains find or replace dialog-box information (such as the text to find and replace) and check box settings.

Public member functions

CmEditFind

void CmEditFind();

Opens a *TFindDialog* in response to an incoming Find command with a CM_EDITFIND command.

CmEditFindNext

void CmEditFindNext();

Responds to an incoming FindNext command with a CM_EDITFINDNEXT command identifier by calling *DoSearch* to repeat the search operation.

See also TEditSearch::DoSearch

CmEditReplace

void CmEditReplace();

Opens a *TReplaceDialog* in response to an incoming Replace command with a CM_EDITREPLACE command.

DoSearch

void DoSearch();

Performs a search or replace operation base on information in *SearchData*.

See also TFindReplaceDialog::TData

EvFindMsg

LRESULT EvFindMsg(WPARAM, LPARAM);

Responds to a message sent by the modeless find or replace dialog box. Calls *DoSearch* to continue searching if text is not found or the end of the document has not been reached.

See also TEditSearch::DoSearch

SetupWindow

void SetupWindow();

Posts a CM_EDITFIND or a CM_EDITREPLACE message to re-open a find or replace modeless dialog box. Calls *TEdit::SetupWindow*.

See also TEdit::SetupWindow

Response table entries

Response table entry	Member function
EV_COMMAND(CM_EDITFIND, CmEditFind)	*CmEditFind*
EV_COMMAND(CM_EDITFINDNEXT, CmEditFindNext)	*CmEditFindNext*
EV_COMMAND(CM_EDITREPLACE, CmEditReplace)	*CmEditReplace*
EV_REGISTERED(FINDMSGSTRING, EvFindMsg)	*EvFindMsg*

TEditView class

editview.h

Derived from *TView* and *TEditSearch*, *TEditView* provides a view wrapper for ObjectWindows text edit class (*TEdit*). A streamable class, *TEditView* includes several event-handling functions that handle messages between a document and its views.

Public constructor and destructor

Constructor

TEditView(TDocument& doc, TWindow* parent = 0);

Creates a *TEditView* object associated with the specified document and parent window. Sets *AttrAccelTable* to IDA_EDITVIEW to identify the edit view. Sets *TView::ViewMenu* to the new *TMenuDescr* for this view.

Destructor

~TEditView()

Destroys a *TEditView* object.

Public member functions

CanClose

bool CanClose();

Returns nonzero if the view can be closed.

Create

bool Create()

Overrides *TWindow::Create* and calls *TEditSearch::Create* to create the view's window. Calls *GetDocPath* to determine if the file is new or already has data. If there is data, calls *LoadData* to add the data to the view. If the view's window can't be created, *Create* indicates the view is invalid.

GetViewName

const char far* GetViewName();

Overrides *TView::GetViewName* and returns the descriptive name of the class (*StaticName*).

See also TEditView::StaticName, TView::GetViewName

GetWindow

TWindow* GetWindow();

GetWindow overrides *GetWindow* in *TView* and returns **this** as a *TWindow*.

See also TView::GetWindow

PerformCreate

void PerformCreate(int menuOrId);

Allocates memory as necessary so that *TEditView* can handle files up to and including 30,000 bytes.

SetDocTitle

bool SetDocTitle(const char far* docname, int index)

Overrides *TView::SetDocTitle* and forwards the title to its base class, *TEditSearch*. *index* is the number of the view displayed in the caption bar. *docname* is the name of the document displayed in the view window.

See also TWindow::SetDocTitle, TView::SetDocTitle

StaticName

static const char far* StaticName();

Returns "Edit View," the descriptive name of the class for the ViewSelect menu.

Protected data member

Origin

long Origin;

Holds the file position at the beginning of the display.

Protected member functions

EvNCDestroy

void EvNCDestroy();

EvNcDestroy is used internally by *TEditView* to manage memory.

This member is not available under Presentation Manager.

LoadData

bool LoadData();

LoadData reads the view from the stream and closes the file. It returns nonzero if the view was successfully loaded. If the file can't be read, posts an error and returns 0.

VnCommit

bool VnCommit(bool force);

VnCommit commits changes made in the view to the document. If *force* is nonzero, all data, even if it's unchanged, is saved to the document.

See also TEditView::vnRevert, vnxxxx view notification constants

VnDocClosed

bool VnDocClosed(int omode);

VnDocClosed indicates that the document has been closed. *mode* is one of the *ofxxxx document open* constants.

See also ofxxxx document open enum, vnxxxx view notification constants

VnIsDirty

bool VnIsDirty();

VnIsDirty returns nonzero if changes made to the data in the view have not been saved to the document; otherwise, it returns 0.

See also vnxxxx view notification constants

VnIsWindow

bool VnIsWindow(HWND hWnd);

VnIsWindow returns nonzero if the window's handle passed in *hWnd* is the same as that of the view's display window.

VnRevert

bool VnRevert(bool clear);

VnRevert is nonzero if changes made to the view should be erased, and the data from the document should be restored to the view. If *clear* is nonzero, the data is cleared instead of restored to the view.

See also TEditView::VnCommit

Response table entries

Response table entry	Member function
EV_VN_COMMIT	*VnCommit*
EV_VN_DOCCLOSED	*VnDocClosed*
EV_VN_ISDIRTY	*VnIsDirty*
EV_VN_ISWINDOW	*VnIsWindow*
EV_WM_NCDESTROY	*EvNcDestroy*
EV_VN_REVERT	*VnRevert*

TEqualOperator typedef

eventhan.h

typedef bool(*TEqualOperator)(TGenericTableEntry __RTFAR&, TEventInfo&);

TEqualOperator is used to perform special kinds of searches and to faciliate finding response table entries. *TEqualOperator* compares a particular message event (*TEventInfo*&) with a response table entry (*TGenericTableEntry*) to determine if they match.

See also

TResponseTableEntry

TEventHandler class

eventhan.h

TEventHandler is a base class from which you can derive classes that handle messages. Specifically, *TEventHandler* performs the following event-handling tasks:

1 Analyzes a window message

2 Searches the class's response table entries for an appropriate event-handling function

3 Dispatches the message to the designated event-handling function

Most of ObjectWindows' classes are derived from *TEventHandler* and, therefore, inherit this event-handling behavior. In addition, any user-defined class derived from *TEventHandler* can handle message response functions that are associated with a particular window message.

See also DECLARE_RESPONSE_TABLE macro

Public member functions

Dispatch

virtual LRESULT Dispatch(TEventInfo&, WPARAM, LPARAM = 0);

Takes the message data from *TEventInfo's Msg* data member and dispatches it to the correct event handling function.

Find

virtual bool Find(TEventInfo&, TEqualOperator = 0);

Searches the list of response table entries looking for a match. Because *TEventHandler* doesn't have any entries, *TEventHandler*'s implementation of this routine returns false.

Protected member function

SearchEntries

bool SearchEntries(TGenericTableEntry __RTFAR* entries, TEventInfo&, TEqualOperator);

Searches the entries in the response table for an entry that matches *TEventInfo* or, if so designated, an entry that *TEqualOperator* specifies is a match.

TEventHandler::TEventInfo class

eventhan.h

A nested class, *TEventInfo* provides specific information about the type of message sent, the class that contains the function to be handled, the corresponding response table entry, and the dispatch function that processes the message.

Public constructor

Constructor

TEventInfo(uint msg, uint id = 0) : Msg(msg), Id(id);

Constructs a *TEventInfo* object with the specified ID and message type.

Public data members

Entry
TGenericTableEntry __RTFAR *Entry;
Points to the response table entry (for example, *EvActivate*).

Id
const uint Id;
Contains the menu or accelerator resource ID (CM_xxxx) for the message response member function.

Msg
const uint Msg;
Contains the type of message sent. These can be command messages, child ID messages, notify-based messages such as LBN_SELCHANGE, or windows messages such as LBUTTONDOWN.

Object
GENERIC *Object;
Points to the object that contains the function to be handled.

TEventStatus enum — window.h

TEventStatus
enum {esPartial, esComplete};
Event status constants indicate the status of a mix-in window event implementation, for example, a keyboard event. The constants indicate whether additional handlers are needed.

Event status constants

Constant	Meaning
esPartial	Additional handlers can be invoked.
esComplete	No additional handlers are needed.

TFileDocument class — filedoc.h

Derived from *TDocument*, *TFileDocument* opens and closes views and provides stream support for views. *TFileDocument* has member functions that continue to process FileNew and FileOpen messages after a view is constructed. You can add support for specialized file types by deriving classes from *TFileDocument*. *TFileDocument* makes this process easy by hiding the actual processs of storing file types.

Public constructor and destructor

Constructor

TFileDocument(TDocument* parent = 0);

Constructs a *TFileDocument* object with the optional parent document.

Destructor

~TFileDocument(TDocument* parent = 0);

Destroys a *TFileDocument* object.

Type definitions

TFileDocProp

enum TFileDocProp {PrevProperty, CreateTime, ModifyTime, AccessTime, StorageSize, FileHandle, NextProperty};

Contains constants that define the following properties of the document:

Value	Description
PrevProperty	*TDocument::NextProperty*–1. This is the first value for view and document objects.
CreateTime	The time the view or document was created.
ModifyTime	The time the view or document was modified.
AccessTime,	The time the view or document was last accessed.
StorageSize	An unsigned long containing the storage size.
FileHandle	The platform file handle.
NextProperty	The is the terminating value for the property.

Classes derived from *TDocument* and *TView* can use these generic file property values.

Public member functions

Close

bool Close();

Closes the document but does not delete or detach any associated views. Before closing the document, *Close* calls *TDocument's Close* to make sure all child documents are closed. If any children are open, *Close* returns 0 and doesn't close the document. If all children are closed, checks to see if any associated streams are open, and if so, returns 0 and doesn't close the document. If there are no open streams, closes the file.

See also TDocument::Close

Commit

bool Commit(bool force = false);

Calls *TDocument_Commit* and clears *TDocument's DirtyFlag* data member, thus indicating that there are no unsaved changes made to the document.

See also TDocument::Commit

FindProperty

int FindProperty(const char far* name);

Gets the property index, given the property name (*name*). Returns 0 if the name isn't found.

See also pfxxxx property attribute constants

GetProperty

int GetProperty(int index, void far* dest, int textlen=0);

Overrides *TDocument_GetProperty* and gets the property ID for the current file document.

See also pfxxxx property attribute constants

InStream

TInStream* InStream(int mode, const char far* strmId = 0);

Overrides *TDocument_InStream* and provides generic input for the particular storage medium. *InStream* returns a pointer to a *TInStream*. *mode* is a combination of the ios bits defined in iostream.h. *strmId* is not used for file documents. The view reads data from the document as a stream or through stream functions.

See also TFileDocument::OutStream

IsOpen

bool IsOpen();

Is nonzero if the document or any streams are open.

Open

Form 1 bool Open(HFILE fhdl);

Opens a file document using an existing file handle. Sets *TDocument_OpenMode* to PREV_OPEN and read/write. Sets the document path to 0. Sets *FHd* to *fhdl*. Always returns nonzero.

Form 2 bool Open(int mode, const char far* path=0);

Overrides *TDocument_Open* and opens the file using the specified path. If the file is already open, returns 0. Calls *TDocument_SetDocPath* to set the directory path. If *omode* isn't 0, sets *TDocument_OpenMode* to *omode*. If the file can't be opened, returns 0.

See also TDocument::SetDocPath, TDocument::Open, ofxxxx document open enum

OutStream

TOutStream* OutStream (int mode, const char far* strmId = 0);

Overrides *TDocument_OutStream* and provides generic input for the particular storage medium. *OutStream* returns a pointer to a *TOutStream*. *mode* is a combination of the ios bits defined in iostream.h. *strmId* is not used for file documents. Instead, the view reads data from the document through stream functions.

See also TFileDocument::InStream

PropertyFlags

int PropertyFlags(int index);

Returns the property attribute constants (*pfGetText*, *pfHidden*, and so on).

See also pfxxxx property attribute constants

PropertyName

const char* PropertyName(int index);

Returns the text name of the property given the index value.

See also pfxxxx property attribute constants

Revert

bool Revert(bool clear = false);

Calls *TDocument_Revert* to notify the views to refresh their data. If *clear* is **false**, the data is restored instead of cleared.

See also TFileDoc::Commit

SetProperty

bool SetProperty(int index, const void far* src);

Sets the property data, which must be in the native data type (either string or binary).

See also pfxxxx property attribute constants

Protected data member

FHdl

HFILE FHdl;

Holds the file handle to an open file document.

Protected member functions

CloseThisFile

void CloseThisFile(HFILE fhdl, int omode);

Closes the file handle if the associated file was opened by *TFileDocument*. Calls *TDocument_NotifyView* to notify all views that the file document has closed.

See also ofxxxx document open enum

OpenThisFile

HFILE OpenThisFile(int omode, const char far* name, streampos* pseekpos);

Opens the file document after checking the file sharing mode (*omode*). If a file mode is not specified as read, write, or read and write, *OpenThisFile* returns 0.

See also ofxxxx document open enum, shxxxx document sharing modes

TFileOpenDialog class

opensave.h

TFileOpenDialog is a modal dialog box that lets you specify the name of a file to open. Use this dialog box to respond to a CM_FILEOPEN command that's generated when a user selects File | Open from a menu. *TFileOpenDialog* uses the *TOpenSave::TData* structure to initialize the file open dialog box.

Public constructor

Constructor

TFileOpenDialog(TWindow* parent, TData& data, TResID templateID = 0, const char far* title = 0, TModule* module = 0);

Constructs and initializes the *TFileOpen* object based on information in the *TOpenSaveDialog::TData* data structure. The *parent* argument points to the dialog box's parent window. *data* is a reference to the *TData* object. *templateID* is the ID for a custom template. *title* is an optional title. *module* points to the module instance.

See also TOpenSaveDialog::TData, TOpenSaveDialog, TResID, TModule

Public member function

DoExecute

int DoExecute();

Creates the *TFileOpenDialog* object.

See also TDialog::DoExecute

TFileSaveDialog class

opensave.h

TFileSaveDialog is a modal dialog box that lets you enter the name of a file to save. Use *TFileSaveDialog* to respond to a CM_FILESAVEAS command generated when a user selects File | Save from a menu. *TFileSaveDialog* uses the *TOpenSave::TData* structure to initialize the file save dialog box.

Public constructor

Constructor

TFileSaveDialog(TWindow* parent, TData& data, TResID templateID = 0, const char far* title = 0, TModule* module = 0);

Constructs and initializes the *TFileOpen* object based on the *TOpenSaveDialog::TData* structure, which contains information about the file name, file directory, and file name search filers.

See also TOpenSaveDialog::TData structure, TModule, TResID, TWindow

Public member function

DoExecute

int DoExecute();

Creates the *TFileSaveDialog* object.

See also TDialog::DoExecute, TOpenSaveDialog

TFilterValidator class

validate.h

A streamable class, *TFilterValidator* checks an input field as the user types into it. The validator holds a set of allowed characters. When the user enters a character, the filter validator indicates whether the character is valid or invalid. See *TValidator* for an example of an input validation screen.

Public constructor

Constructor

TFilterValidator(const TCharSet& validChars);

Constructs a filter validator object by first calling the constructor inherited from *TValidator*, then setting *ValidChars* to *validChars*.

Public member functions

Error

void Error();

Error overrides *TValidator's* virtual function and displays a message box indicating that the text string contains an invalid character.

See also TValidator::Error

IsValid

bool IsValid(const char far* str);

IsValid overrides *TValidator's* **virtual**s and returns **true** if all characters in *str* are in the set of allowed characters, *ValidChar*; otherwise, it returns **false**.

IsValidInput

bool IsValidInput(char far* str, bool suppressFill);

IsValidInput overrides *TValidator's* virtual function and checks each character in the string *str* to ensure it is in the set of allowed characters, *ValidChar*. *IsValidInput* returns **true** if all characters in *str* are valid; otherwise, it returns **false**.

See also TValidator::IsValidInput

Protected data members

ValidChars

TCharSet ValidChars;

Contains the set of all characters the user can type. For example, to allow only numeric digits, set *ValidChars* to "0–9". *ValidChars* is set by the *validChars* parameter passed to the constructor.

TFindDialog class

findrepl.h

TFindDialog objects represents modeless dialog box interface elements that let you specify text to find. *TFindDialog* communicates with the owner window using a registered message. Derived from *TFindReplaceDialog*, *TFindDialog* uses the *TFindReplaceDialog::TData* structure to initialize the dialog box with user-entered values (such as the text string to find).

Public constructor

Constructor

```
TFindDialog(TWindow* parent, TData& data, TResId templateId = 0, const char far* title = 0,
  TModule* module = 0);
```

Constructs a *TFindDialog* object with the given parent window, resource ID, and caption. Sets the attributes of the dialog box based on*TFindReplaceDialog::TData* structure, which contains information about the text string to search for.

See also TFindReplaceDialog::TData

Protected member functions

DoCreate

```
HWND DoCreate();
```

Creates the modeless interface element of a find dialog.

TFindReplaceDialog class

findrepl.h

TFindReplaceDialog is an abstract base class for a modeless dialog box that lets you search for and replace text. This base class contains functionality common to both derived classes, *TFindDialog* which lets you specify text to find, and *TReplaceDialog* which lets you specify replacement text. *TFindReplaceDialog* communicates with the owner window using a registered message.

Public constructor

Constructor

```
TFindReplaceDialog(TWindow* parent, TData& data, TResId templateId = 0, const char far* title = 0,
  TModule* module = 0);
```

Constructs a *TFindReplaceDialog* object with a parent window, resource ID, and caption. Sets the attributes of the dialog box with the specified data from the *TFindReplaceDialog::TData* structure.

See also TFindReplaceDialog::TData struct, TModule, TResID, TWindow

Public member functions

CmCancel
void CmCancel();
Responds to a click of the Cancel button.

CmFindNext
void CmFindNext();
Responds to a click of the Find Next button.

CmReplace
void CmReplace();
Responds to a click of the Replace button.

CmReplaceAll
void CmReplaceAll();
Responds to a click of the Replace All button.

EvNCDestroy
void EvNCDestroy();
Calls *TWindow::EvNCDestroy*, which responds to an incoming EV_WM_NCDESTROY message which tells the owner window that is nonclient area is being destroyed.

Protected data members

Data
TData& Data;
Data is a reference to the *TData* object passed in the constructor.

See also TFindReplaceDialog::TData class

fr
FINDREPLACE fr;
A **struct** that contains find-and-replace attributes, such as the size of the find buffer and pointers to search and replace strings, used for find-and-replace operations.

See also FINDREPLACE class

Protected member functions

DoCreate
HWND DoCreate()=0;
DoCreate is a virtual function that is overridden in derived classes to create a modeless find or replace dialog box.

DialogFunction
bool DialogFunction(uint message, WPARAM, LPARAM);
Returns **true** if a message is handled.

See also TDialog::DialogFunction

Init

void Init(TResId templateId);

Used by constructors in derived classes, *Init* initializes a *TFindReplaceDialog* object with the current resource ID and other members.

Response table entries

Response table entry	Member function
EV_WM_NCDESTROY	*EvNCDestroy*

TFindReplaceDialog::TData class

findrepl.h

The *TFindReplaceDialog::TData* class encapsulates information necessary to initialize a *TFindReplace* dialog box. The *TFindDialog* and *TReplaceDialog* classes use the *TFindReplaceDialog::TData* class to initialize the dialog box and to accept user-entered options such as the search and replacement text strings.

Public constructor and destructor

Constructor

TFindReplaceDialog(uint32 flags = 0, int bufferSize = 81);

Constructs a *TData* object with the specified flag value that initializes the status of the dialog box control buttons and the buffer size for the find and replace search strings.

Destructor

~TData();

Destroys a *~TData* object.

Public data members

BuffSize

int BuffSize;

BuffSize contains the size of the text buffer.

Error

uint32 Error;

If the dialog box is successfully created, *Error* is 0. Otherwise, it contains one or more of the following error codes:

Constant	Meaning
CDERR_LOCKRESOURCEFAILURE	Failed to lock a specified resource.
CDERR_LOADRESFAILURE	Failed to load a specified resource.

Constant	Meaning
CDERR_LOADSTRFAILURE	Failed to load a specified string.
CDERR_REGISTERMSGFAIL	The window message (a value used to communicate between applications) cannot be registered. This message value is used when sending or posting window messages.

FindWhat

char* FindWhat;

Contains the search string.

Flags

uint32 Flags;

Flags, which indicates the state of the control buttons and the action that occurred in the dialog box, can be a combination of the following constants that indicate which command the user wants to select:

Constant	Meaning
FR_DOWN	The Down button in the Direction group of the Find dialog box is selected.
FR_HIDEMATCHCASE	The Match Case check box is hidden.
FR_HIDEWHOLEWORD	The Whole Word check box is hidden.
FR_HIDEUPDOWN.	The Up and Down buttons are hidden.
FR_MATCHCASE	The Match Case check box is checked.
FR_NOMATCHCASE	The Match Case check box is disabled. This occurs when the dialog box is first initialized.
FR_NOUPDOWN	The Up and Down buttons are disabled. This occurs when the dialog box is first initialized.
FR_NOWHOLEWORD	The Whole Word check box is disabled. This occurs when the dialog box is first initialized.
FR_REPLACE	The Replace button was pressed in the Replace dialog box.
FR_REPLACEALL	The Replace All button was pressed in the Replace dialog box.
FR_WHOLEWORD	The Whole Word check box is checked.

ReplaceWith

char* ReplaceWith;

ReplaceWith contains the replacement string.

See also TEditSearch::SearchData, TFindReplaceDialog::Data

TFloatingFrame class

floatfra.h

Derived from *TFrameWindow* and *TTinyCaption*, *TFloatingFrame* implements a floating frame that can be positioned anywhere in the parent window. Except for the addition of a tiny caption bar, the default behavior of *TFrameWindow* and *TFloatingFrame* is the same. Therefore, an application that uses *TFrameWindow* can easily gain the functionality of *TFloatingFrame* by just changing the name of the class to *TFloatingFrame*.

If there is a client window, the floating frame shrinks to fit the client window, leaving room for margins on the top, bottom, left, and right of the frame. Because the floating frame expects the client window to paint its own background, it does nothing in response to a WM_ERASEBKGND message. However, if there is no client window, the floating frame erases the client area background using COLOR_BTNFACE.

See PAINT.CPP, the sample program on your distribution disk, for an example of a floating frame.

Public constructor

Constructor

```
TFloatingFrame(TWindow *owner, char *title = 0, TWindow* clientWnd = 0, bool shrinkToClient = false,
  int CaptionHeight = DefaultCaptionHeight, bool popupPalette = false, Module* module = 0);
```

Constructs a *TFloatingFrame* object attached to the specified parent window. By default, the floating frame window doesn't shrink to fit the client window, and the floating palette style isn't enabled.

Set *popupPalette* to true if you want to enable a floating palette style for the window. The floating palette is a popup window with a tiny caption, a standard window border, and a close box instead of a system menu box. There are no maximize or minimize buttons. A one pixel border is added around the client area in case a toolbox is implemented. This style must be turned on before the window is created. After the window is created, its style can't be changed.

See also TFrameWindow::TFrameWindow, TTinyCaption::TTinyCaption

Public member functions

SetMargins

```
void SetMargins(const TSize& margin);
```

Sets the margins of the floating palette window to the size specified in *margin* and sets the height of the tiny caption bar.

See also TTinyCaption::EnableTinyCaption

Protected member functions

DoNCHitTest

```
TEventStatus  DoNCHitTest(TPoint& screenPt, uint& evRes);
```

If the floating palette is not enabled, returns *esPartial*. Otherwise, sends a message to the floating palette that the mouse or the cursor has moved, and returns *esComplete*.

Response table entries

Response table entry	Member function
EV_WM_SYSCOMMAND	*EvSysCommand*
EV_WM_NCCALCSIZE	*EvNCCalcSize*
EV_WM_NCPAINT	*EvNCPaint*
EV_WM_NCHITTEST	*EvNcHitTest*

TFont class

gdiobjec.h

TFont derived from *TGdiObject* provides constructors for creating font objects from explicit information or indirectly.

Public constructors

Constructors

Form 1 `TFont(HFONT handle, TAutoDelete autoDelete = NoAutoDelete);`

Creates a *TFont* object and sets the *Handle* data member to the given borrowed *handle*. The *ShouldDelete* data member defaults to **false**, ensuring that the borrowed handle will not be deleted when the C++ object is destroyed.

Form 2
```
TFont(const char far* facename = 0, int height = 0, int width = 0, int escapement = 0,
  int orientation = 0 int weight = FW_NORMAL, uint8 pitchAndFamily = DEFAULT_PITCH|
  FF_DONTCARE, uint8 italic = false, uint8 underline = false, uint8 strikeout = false, uint8 charSet = 1,
  uint8 outputPrecision = OUT_DEFAULT_PRECIS, uint8 clipPrecision = CLIP_DEFAULT_PRECIS,
  uint8 quality = DEFAULT_QUALITY);
```

Creates a *TFont* object with the given values.

Form 3
```
TFont(int height, int width, int escapement = 0, int orientation = 0, int weight = FW_NORMAL, uint8 italic = false,
  BYTE underline = false, uint8 strikeout = false, uint8 charSet = 1, uint8 outputPrecision =
  OUT_DEFAULT_PRECIS, uint8 clipPrecision = CLIP_DEFAULT_PRECIS, uint8 quality =
  DEFAULT_QUALITY uint8 pitchAndFamily = DEFAULT_PITCH|
  FF_DONTCARE, const char far* facename = 0);
```

Creates a font object with the given values. The constructor parameter list and default values match the Windows API *CreateFont* call.

Form 4 `TFont(const LOGFONT far* logFont);`

Creates a *TFont* object from the given *logFont*.

Form 5 `TFont(const TFont& font);`

The *TFont* copy constructor.

See also ::CreateFont (Windows API), ::CreateFontIndirect (Windows API), TGdiObject::Handle, TGdiObject::ShouldDelete,

Public member functions

GetObject

bool GetObject(LOGFONT far& logFont) const;

Retrieves information about this pen object and places it in the given *LOGFONT* structure. Returns **true** if successful and **false** if unsuccessful.

See also TGdiObject::GetObject, LOGFONT struct

operator HFONT()

operator HFONT() const;

Typecasting operator that converts this font's *Handle* to type *HFONT* (the data type representing the handle to a physical font).

TFrameWindow class

framewin.h

Derived from *TWindow*, *TFrameWindow* controls such window-specific behavior as keyboard navigation and command processing for client windows. For example, when a window is reactivated, *TFrameWindow* is responsible for restoring a window's input focus and for adding menu bar and icon support. *TFrameWindow* is a streamable class.

In terms of window areas, the frame area consists of the border, system menus, toolbars and status bars whereas the client area excludes these areas. Although frame windows can support a client window, the frame window remains separate from the client window so that you can change the client window without affecting the frame window.

ObjectWindows uses this frame and client structure for both *TFrameWindow* and *TMDIChild* classes. Both these classes can hold a client class. Having a separate class for the client area of the window adds more flexibility to your program. For example, this separate client area, which might be a dialog box, can be moved into another frame window, either a main window or an MDI child window.

See *TFloatingFrame* for a description of a floating frame with the same default functionality as a frame window.

Public constructors and destructor

Constructors

Form 1 TFrameWindow(TWindow* parent, const char far *title = 0, TWindow* clientWnd = 0, bool shrinkToClient = false, TModule* module = 0);

Constructs a window object with the parent window supplied in *parent*, which is zero if this is the main window. *title*, which by default is zero, contains the title displayed in the window's caption bar. *clientWnd* is the client window for this frame window or zero if none exists. *shrinkToClient* controls whether the client window will size to fit the frame or the frame window will fit the client. Note that this parameter only affects the size of the main window. When a client window is used in a frame window that doesn't have *shrinktoClient* set, the client window resizes to fit the frame window. When a client window is used in a frame window that has the *shrinktoClient* set, the frame window shrinks to fit the size of the client window.

Form 2 TFrameWindow(HWND hWnd, TModule* module = 0);
Constructor for a *TFrameWindow* that is being used as an alias for a non-ObjectWindows window. *hWnd* is the handle to the existing window object that *TFrameWindow* controls; *module* contains the module passed to the base class's contructor.

Destructor
~TFrameWindow();
Deletes any associated menu descriptor.

See also TWindow::TWindow, TFloatingFrame::TFloatingFrame

Public data members

KeyboardHandling
bool KeyboardHandling;
Indicates if keyboard navigation is required.

Public member functions

AssignMenu
virtual bool AssignMenu(TResId menuResId);
Sets *Attr.Menu* to the supplied *menuResId* and frees any previous strings pointed to by *Attr.Menu*. If *HWindow* is nonzero, loads and sets the menu of the window, destroying any previously existing menu.

See also TMDIFrameWindow::SetMenu, TFrameWindow::SetMenuDescr

EnableKBHandler
void EnableKBHandler();
Sets a flag indicating that the receiver has requested keyboard navigation (translation of keyboard input into control selections). By default, the keyboard interface, which lets users use the Tab and arrow keys to move between the controls, is disabled for windows and dialog boxes.

GetClientWindow
virtual TWindow* GetClientWindow();
Returns a pointer to the client window. If you are trying to access a window-based object in a *TMDIChild* (which is a frame window), you can use this function.

See also TMDIChild

GetCommandTarget
virtual HWND GetCommandTarget();
Locates and returns the child window that is the target of the command and command enable messages. If the current application does not have focus or if the focus is within a toolbar in the application, *GetCommandTarget* returns the most recently active child window.

If an alternative form of command processing is desired, a user's main window class can override this function. TFrameWindow's *EvCommand* and *EvCommandEnable* functions

use *GetCommandTarget* to find the command target window. This member is not available under Presentation Manager.

GetMenuDescr
const TMenuDescr* GetMenuDescr();
Returns a pointer to the menu descriptor.

See also TFrameWindow::SetMenuDescr, TMenuDescr

HoldFocusHWnd
bool HoldFocusHWnd(HWND hWndLose, HWND hWndGain);
Overrides *TWindow's* virtual function. Responds to a request by a child window to hold its HWND when it is losing focus. Stores the child's HWND in *HwndRestoreFocus*.

See also TWindow::HoldFocusHwnd

IdleAction
void IdleAction(long idleCount);
Overrides *TWindow's* virtual function. *TApplication* calls the main window's *IdleAction* when no messages are waiting to be processed. *TFrameWindow* uses this idle time to perform command enabling for the menu bar. It also forwards *IdleAction* to each of its children. *IdleAction* can be overridden to do background processing.

See also TApplication::IdleAction

MergeMenu
bool MergeMenu(const TMenuDescr& childMenuDescr);
Merges the given menu descriptor with this frame's own menu descriptor and displays the resulting menu in this frame. See *TMenuDescr* for a description of menu bar types that can be merged.

See also TMenuDescr

PreProcessMsg
bool PreProcessMsg(MSG& msg);
Overrides *TWindow's* virtual function. Performs preprocessing of window messages. If the child window has requested keyboard navigation, *PreProcessMsg* handles any accelerator key messages and then processes any other keyboard messages.

See also TWindow::PreProcessMsg

RestoreMenu
bool RestoreMenu();
Restores the default menu of the frame window.

SetClientWindow
virtual TWindow* SetClientWindow(TWindow* clientWnd);
Sets the client window to the specified window. Users are responsible for destroying the old client window if they want to eliminate it.

SetDocTitle
bool SetDocTitle(const char far* docname, int index);

Overrides *TWindow's* virtual function. Pastes the number of the view into the caption and then shows the number on the screen. This function can be overridden if you don't want to use the default implementation, which displays a number on the screen. That is, you might want to write "Two" instead of "2" on the screen. For an example of the behavior of this function, see step 12 of the ObjectWindows tutorial, which renumbers the views if one of them is closed.

SetIcon

bool SetIcon(TModule* iconModule, TResId iconResId);

Sets the icon in the module specified in *iconModule* to the resource ID specified in *iconResId*. See the sample file bmpview.cpp for an example of painting an icon from a bitmap. You can set the *iconResId* to one of these pre-defined values as well as user-defined values:

IDI_APPLICATION	Default icon used for applications
IDI_ASTERISK	Asterisk used for an informative message
IDI_EXCLAMATION	Exclamation mark used for a warning message
IDI_HAND	Hand used for warning messages
IDI_QUESTION	Question mark used for prompting a response

See also TFrameWindow::EvQueryDragIcon

SetMenu

virtualBOOL SetMenu(HMENU newMenu);

Overrides *TWindow's* non-virtual *SetMenu* function, thus allowing derived classes the opportunity to implement this function differently from *TWindow*. *SetMenu* sets the window's menu to the menu indicated by *newMenu*. If *newMenu* is 0, the window's current menu is removed. *SetMenu* returns 0 if the menu remains unchanged; otherwise, it returns a nonzero value.

SetMenuDescr

void SetMenuDescr(const TMenuDescr& menuDescr);

Sets the menu descriptor to the new menu descriptor.

See also TFrameWindow::GetMenuDescr, TMenuDescr

Protected data members

ClientWnd

TWindow* ClientWnd;

ClientWnd points to the frame's client window.

DocTitleIndex

int DocTitleIndex;

Holds the index number for the document title.

HWndRestoreFocus

HWND HWndRestoreFocus;

Stores the handle of the child window whose focus gets restored.

See also TFrameWindow::HoldFocusHwnd

MergeModule

TModule* MergeModule;

Tells the frame window which module the menu comes from. *TDecoratedFrame* uses this member to get the menu hints it displays at the bottom of the screen. It assumes that the menu hints come from the same place the menu came from.

Protected constructor

Constructor

TFrameWindow();

Protected constructor used in conjunction with *Init* function for initializing virtually derived classes.

Protected member functions

EvCommand

LRESULT EvCommand(uint id, HWND hWndCtl, uint notifyCode);

Provides extra processing for commands and lets the focus window and its parent windows handle the command first.

EvCommandEnable

void EvCommandEnable(TCommandEnabler& ce);

Handles checking and unchecking of the frame window's menu items.

EvEraseBkgnd

bool EvEraseBkgnd(HDC);

EvEraseBkgnd erases the background of the window specified in *HDC*. It returns **true** if the background is erased; otherwise, it returns **false**.

EvInitMenuPopup

HANDLE EvInitMenuPopup(HMENU hPopupMenu, uint index, bool sysMenu);

Sent before a pop-up menu is displayed, *EvInitMenuPopup* lets an application change the items on the menu before the menu is displayed. *EvInitMenuPopup* controls whether the items on the pop-up menu are enabled or disabled, checked or unchecked, or strings. *HMENU* indicates the menu handle. *index* is the index of the pop-up menu. *sysMenu* indicates if the pop-up menu is the system menu.

EvPaint

void EvPaint();

Responds to a WM_PAINT message in the client window in order to paint the iconic window's icon or to allow client windows a change to paint the icon.

See also TWindow::Paint, TScroller::BeginView, TScroller::EndView

EvParentNotify

void EvParentNotify(uint event, uint childHandleOrX, uint childIDOrY);

Responds to a message to notify the parent window that a given event has occurred. If the client window is destroyed, closes the parent window. If *shrinkToClient* is set and the child window has changed size, the frame is adjusted.

When a *TFrameWindow's* client window is destroyed, the *TFrameWindow* object sees the WM_PARENTNOTIFY message and posts a close message to itself. Without this message, an empty frame would remain and the client window would then have to determine how to destroy the frame. If you don't want this to happen, you can derive from the frame window and have your application handle the *EvParentNotify* or *EvClose* messages.

EvQueryDragIcon

HANDLE EvQueryDragIcon();

Responds to a WM_QUERYDRAGICON message sent to a minimized (iconic) window that is going to be dragged. Instead of the default icon, *EvQueryDragIcon* uses the icon that was set using *SetIcon*.

This member is not available under Presentation Manager.

See also TFrameWindow::SetIcon

EvSetFocus

void EvSetFocus(HWND hWndLostFocus);

Restores the focus to the active window. *hWndLostFocus* contains the handle for the window that lost focus.

EvSize

void EvSize(uint sizeType, TSize& size);

Resizes the client window's size so that it is equivalent to the client rectangle's size. Calls *TWindow::EvSize)* in response to an incoming WM_SIZE message.

See also TSize

Init

void Init (TWindow* clientWnd, bool shrinkToClient);

This initialize function is for use with virtually derived classes, which must call *Init* before construction is completed. This procedure provides necessary data to virtually derived classes and takes care of providing the data in the appropriate sequence.

SetupWindow

void SetupWindow();

Calls *TWindow::SetUpWindow* to create windows in a child list. *SetupWindow* performs the initial adjustment of the client window if one exists, assigns the frame's menu based on the menu descriptor, and initializes *HwndRestoreFocus*.

See also TWindow::SetUpWindow

Response table entries

Response table entry	Member function
EV_WM_ERASEBKGND	*EvEraseBkgnd*
EV_WM_INITMENUPOPUP	*EvInitMenuPopup*
EV_WM_PAINT	*EvPaint*
EV_WM_PARENTNOTIFY	*EvParentNotify*
EV_WM_QUERYDRAGICON	*EvQueryDragIcon (Windows only)*
EV_WM_SETFOCUS	*EvSetFocus*
EV_WM_SIZE	*EvSize*

TGadget class

gadget.h

TGadget is the base class for the following derived gadget classes:

Class	Description
TBitmapGadget	Displays a bitmap.
TButtonGadget	Uses a bitmap to similate a button gadget.
TControlGadget	Encapsulates inserting a control such as an edit control or a combobox, into a gadget window.
TTextGadget	Displays text.
TSeparatorGadget	Separates logical groups of gadgets.

TGadget interface objects belong to a gadget window, have borders and margins, and have their own coordinate system. The margins are the same as those for *TGadgetWindow* and borders are always measured in border units.

To set the attributes for the gadget, you can either choose a border style (which automatically sets the individual border edges) or set the borders and then override the member function *PaintBorder* to create a custom look for your gadget. If you change the borders, margins, or border style, the gadget window's *GadgetChangedSize* member function is invoked.

Although, by default, gadgets shrink-wrap to fit around their contents, you can control this attribute by setting your own values for *ShrinkWrapWidth* and *ShrinkWrapHeight*.

A gadget window, being an actual window, receives messages from the mouse. After the gadget window receives the message, it decides which gadget should receive the message by calling the member function directly instead of sending or posting a message.

See also

TBitmapGadget, TButtonGadget, TControlGadget, TTextGadget, TSeparatorGadget

Public constructors and destructor

Constructor

TGadget(int id = 0, TBorderStyle = None);

Constructs a *TGadget* object with the specified ID and border style.

Destructor

virtual ~TGadget();

Destroys a *TGadget* interface object and removes it from its associated window.

Public data members

Clip

bool Clip;

If *Clip* is **false**, clipping borders have not been established. If *Clip* is **true**, the drawing for each gadget is restrained by the gadget's border.

WideAsPossible

bool WideAsPossible;

Initially set to false, *WideAsPossible* indicates whether the gadget width will be adjusted by the gadget window to be as wide as possible in the remaining space.

See also TGadgetWindow::WideAsPossible

Public enums and structs

TBorders

```
struct TBorders {
  unsigned  Left;
  unsigned  Right;
  unsigned  Top;
  unsigned  Bottom;
  TBorders(){Left=Right=Top=Bottom=0;}
  };
```

TBorders structure holds the values for the left, right, top, and bottom measurements of the gadget.

TBorderStyle

enum TBorderStyle {None, Plain, Raised, Recessed, Embossed};

Enumerates an exclusive list of border styles: none, plain, raised (the gadget appears to be raised above the gadget window), recessed (the gadget appears to be recessed into the gadget window), or embossed (the gadget appears to have an embossed border). For an example of border styles, see the sample ObjectWindows program, MDIFILE.CPP, on your distribution disk.

TMargins

```
struct TMargins {
  enum TUnits {Pixels, LayoutUnits, BorderUnits};
  TUnits  Units;
```

```
  int Left;
  int Right;
  int Top;
  int Bottom;
  TMargin(){Units=LayoutUnits;Left=Right=Top=Bottom=0;}
  };
```

Used by the *TGadgetWindow* and *TGadget* classes, *TMargins* contains the measurements of the margins for the gadget. The constructor initializes *Units* to *LayoutUnits* and sets *Left*, *Right*, *Top*, and *Bottom* equal to 0.

See also TGadgetWindow::SetMargins

Public member functions

CommandEnable

```
virtual void CommandEnable();
```

CommandEnable is provided so that the gadget can perform command enabling (so it can handle an incoming message, if it's appropriate to do so).

GetBorders

```
TBorders& GetBorders();
```

Gets the gadget's borders measured in border units that are based on SM_CXBORDER and SM_CYBORDER.

See also TGadget::SetBorders

GetBorderStyle

```
TBorderStyle GetBorderStyle();
```

Gets the style for the gadget's borders.

See also TGadget::SetBorderStyle

GetBounds

```
TRect& GetBounds();
```

Returns the boundary rectangle for the gadget.

See also TButtonGadget::SetNotchCorners

GetDesiredSize

```
virtual void GetDesiredSize(TSize& size);
```

GetDesiredSize determines how big the gadget can be. The gadget window sends this message to query the gadget's size. If shrink-wrapping is requested, *GetDesiredSize* returns the size needed to accommodate the borders and margins. If shrink-wrapping is not requested, it returns the gadget's current width and height. *TGadgetWindow* needs this information to determine how big the gadget needs to be, but it can adjust these dimensions if necessary. If *WideAsPossible* is true, then the width parameter (*size.cx*) is ignored.

GetEnabled

```
bool GetEnabled();
```

Determines whether keyboard and mouse input have been enabled for the specified gadget. If the gadget is enabled, *GetEnabled* returns true; otherwise, it returns false. By default, keyboard and mouse input are enabled.

See also TGadget::SetEnabled

GetId

```
int GetId();
```

Gets the ID for the gadget.

GetMargins

```
TMargins& GetMargins();
```

Gets the margin dimensions.

GetOuterSizes

```
void GetOuterSizes(int& left, int& right, int& top, int& bottom);
```

Returns the amount of space (in pixels) taken up by the borders and margins.

NextGadget

```
TGadget* NextGadget();
```

Returns the next gadget in the list of gadgets.

SetBorders

```
void SetBorders(TBorders& borders);
```

Sets the borders for the gadget. If the borders are changed, *SetBorders* calls *TGadgetWindow::GadgetChangedSize* to notify the gadget window of the change.

See also TGadget::GetBorders, TGadgetWindow::GadgetChangedSize

SetBorderStyle

```
void SetBorderStyle(TBorderStyle);
```

Sets the border style for the gadget.

See also TGadget::GetBorderStyle

SetBounds

```
virtual void SetBounds(TRect& rect);
```

SetBounds informs the gadget of a change in its bounding rectangle. Although the default behavior updates only the instance variable *Bounds*, you can override this method to also update the internal state of the gadget.

SetEnabled

```
virtual void SetEnabled(bool);
```

Enables or disables keyboard and mouse input for the gadget. By default, the gadget is disabled when it is created and must be enabled before it can be activated.

See also TGadget::GetEnabled

SetMargins

```
void SetMargins(TMargins& margins);
```

Sets the margins of the gadget. If the margins are changed, *SetMargins* calls *TGadgetWindow::GadgetChangedSize* to notify the gadget window.

See also TGadget::GetMargins

SetShrinkWrap

void SetShrinkWrap(bool shrinkWrapWidth, bool shrinkWrapHeight);

Sets the *ShrinkWrapWidth* and *ShrinkWrapHeight* data members. Your derived class can call *TGadgetWindow's GadgetChangedSize* member function if you want to change the size of the gadget.

See also TGadgetWindow::GadgetChangedSize

SetSize

void SetSize(TSize& size);

SetSize alters the size of the gadget and then calls *TGadgetWindow::GadgetChangedSize* for the size change to take effect.

This function is needed only if you have turned off shrink-wrapping in one or both dimensions; otherwise, use the *GetDesiredSize* member function to return the shrink-wrapped size.

SysColorChange

virtual void SysColorChange();

SysColorChange is called when the system colors have been changed so that gadgets can rebuild and repaint, if necessary.

Protected data members

Bounds

TRect Bounds;

Contains the bounding rectangle for the gadget in gadget window coordinates.

See also TGadget::GetInnerRect

Borders

TBorders Borders;

Contains the border measurements *fTGadget::GetInnerRect*

BorderStyle

TBorderStyle BorderStyle;

Contains the border style for the gadget.

Id

int Id;

Contains the gadget's ID.

Margins

TMargins Margins;

Contains the margin measurements of the rectangle or the gadget.

See also TGadget::GetInnerRect

ShrinkWrapHeight

bool ShrinkWrapHeight;

Indicates if the gadget is to be shrink-wrapped to fit around its contents.

ShrinkWrapWidth
bool ShrinkWrapWidth;
Indicates if the gadget is to be shrink-wrapped to fit around its contents.

TrackMouse
bool TrackMouse;
Initialized to false. When *TrackMouse* is **true**, the gadget captures and releases the mouse on *LButtonDown* and *LButtonUp* by calling *TGadgetWindow's GadgetSetCapture* and *GadgetReleaseCapture*.

See also TGadget::LButtonDown, TGadget::LButtonUp

Window
TGadgetWindow* Window;
References the owning or parent window for the gadget.

Protected member functions

GetInnerRect
void GetInnerRect(TRect& rect);
Computes the area of the gadget's rectangle excluding the borders and margins.

Inserted
virtual void Inserted();
Called after a gadget is inserted into a window.

Invalidate
void Invalidate(bool erase = true);
Used to invalidate the active (usually nonborder) portion of the gadget, *Invalidate* calls *InvalidateRect* and passes the boundary width and height of the area to erase.

InvalidateRect
void InvalidateRect(const TRect& rect, bool erase = true);
Invalidates the gadget-relative rectangle in the parent window.

LButtonDown
virtual void LButtonDown(uint modKeys, TPoint& point);
Captures the mouse if *TrackMouse* is set. *point* is located in the gadget's coordinate system.

See also TGadget::TrackMouse

LButtonUp
virtual void LButtonUp(uint modKeys, TPoint& point);
Releases the mouse capture if *TrackMouse* is set. *point* is located in the gadget's coordinate system.

See also TGadget::TrackMouse

MouseEnter
virtual void MouseEnter(uint modKeys, TPoint& point);
Called when the mouse enters the gadget.

See also TGadget::MouseLeave

MouseLeave
virtual void MouseLeave(uint modKeys, TPoint& point);
Called when the mouse leaves the gadget.

See also TGadget::MouseEnter

MouseMove
virtual void MouseMove(uint modKeys, TPoint& point);
If mouse events are captured, *EvMouseMove* responds to a mouse dragging message. *point* is located in the receiver's coordinate system.

See also TGadget::MouseEnter, TGadget::MouseLeave

Paint
virtual void Paint(TDC&);
Calls *PaintBorder* to paint the indicated device context.

See also TTextGadget::Paint

PaintBorder
virtual void PaintBorder(TDC& dc);
Used to paint the border, *PaintBorder* calls *::GetSystemMetrics* to obtain the width and height of the gadget and uses the color returned by *GetSyscolor* to paint or highlight the area with the specified brush. Depending on whether the border style is raised, embossed, or recessed, *PaintBorder* paints the specified boundary. You can override this function if you want to implement a border style that isn't supported by ObjectWindows's gadgets.

PtIn
virtual bool PtIn(TPoint& point);
PtIn determines if the point is within the receiver's bounding rectangle and returns true if this is the case; otherwise, returns false.

Removed
virtual void Removed();
Called after a gadget is removed from a window.

Update
void Update();
Repaints the gadget if possible.

TGadgetWindowFont class — gadgetwi.h

Derived from *TFont*, *TGadgetWindowFont* is a specific font used in gadget windows for sizing and default text. You can specify the point size of the font (not the size in pixels)

and whether it is bold or italic. You can use one of the following *FF_xxxx* constants to indicate the font family type:

Value	Meaning
FF_DECORATIVE	Speciality fonts such as Old English.
FF_DONTCARE	The font type does not matter.
FF_MODERN	Fonts such as Pica, Elite or Courier with a constant stroke, width, and with or without serifs.
FF_ROMAN	Fonts such as Times New Roman and New Century Schoolbook with varied stroke and with serifs.
FF_SCRIPT	Fonts such as Script that are designed to resemble handwriting.
FF_SWISS	Fonts such as MS Sans Serif with variable stroke width and without serifs.

Depending on the typeface, the font weight can be one of the following constants:

Value	Meaning
FW_DONTCARE	The font weight does not matter.
FW_THIN	Thin
FW_EXTRALIGHT	Extra light
FW_ULTRALIGHT	Extra light
FW_LIGHT	Light
FW_NORMAL	Normal
FW_REGULAR	Normal font weight
FW_MEDIUM	Medium
FW_SEMIBOLD	Somewhat bold
FW_DEMIBOLD	Somewhat bold
FW_BOLD	Bold
FW_EXTRBOLD	Extra bold
FW_ULTRBOLD	Extra bold
FW_BLACK	Heavy weight
FW_HEAVY	Heavy weight

The font's appearance depends on the typeface so that some fonts only have FW_NORMAL, FW_REGULAR and FW_BOLD available. If FW_DONTCARE is indicated, the default font weight is used.

Public constructor

Constructor

```
TGadgetWindowFont(int pointSize = 10, bool bold = false, bool italic = false);
```

Constructs a *TGadgetWindowFont* interface object with a default point size of 10 picas without bold or italic typeface. By default, the constructor creates the system font: a variable-width, sans-serif Helvetica.

See also TFont

TGadgetWindow class

gadgetwi.h

Derived from *TWindow*, *TGadgetWindow* maintains a list of tiled gadgets for a window and lets you dynamically arrange tool bars. You can specify the following attributes of these gadgets:

- Horizontal or vertical tiling. Positions the gadgets horizontally or vertically within the inner rectangle (the area excluding borders and margins).
- Gadget font. Default font to use for gadgets and for calculating layout units. For font information, see the description of *TGadgetWindowFont*.
- Left, right, top, and bottom margins. Specified in pixels, layout units (based on the window font), or border units (the width or height of a thin window border).
- Measurement units. Specified in pixels, layout units, or border units.
- Gadget window size. A gadget window can shrink-wrap its width, height, or both to fit around its gadgets. By default, horizontally tiled gadgets shrink-wrap to fit the height of the window and vertically tiled gadgets shrink-wrap to fit the width of the window.

TGadgetWindow is the base class for the following derived classes: *TControlBar*, *TMessageBar*, *TToolBox*, and *TStatusBar*.

Public constructor and destructor

Constructor

```
TGadgetWindow(TWindow* parent = 0, TTileDirection direction = Horizontal,
  TFont *font = new TGadgetWindowFont, TModule* module = 0);
```

Creates a *TGadgetWindow* interface object with the default tile direction and font and passes *module* with a default value of 0.

Destructor

```
~TGadgetWindow();
```

Destructs the *TGadgetWindow* object by deleting all of its gadgets and fonts.

Type definitions

THintMode

```
enum THintMode{NoHints, PressHints, EnterHints};
```

Enumerates the hint mode settings of the gadget—either no hints, hints when a button is pressed, or hints when the mouse passes over a gadget.

See also TGadgetWindow::GetHintMode

Public member functions

FirstGadget

```
TGadget* FirstGadget() const;
```

Returns the *FirstGadget* in the list.

See also TGadgetWindow::FirstGadget

GadgetChangedSize

void GadgetChangedSize(TGadget& gadget);

Used to notify the gadget window that a gadget has changed its size, *GadgetChangedSize* calls *LayoutSession* to re-layout all gadgets.

See also TGadget::SetShrinkWrap, TGadgetWindow::GadgetChangedSize

GadgetFromPoint

TGadget* GadgetFromPoint(TPoint& point);

Returns the gadget at the given window coordinates.

GadgetReleaseCapture

void GadgetReleaseCapture(TGadget& gadget);

Releases the capture so that other windows can receive mouse messages.

See also TGadgetWindow::GadgetSetCapture

GadgetSetCapture

bool GadgetSetCapture(TGadget& gadget);

GadgetSetCapture reserves all mouse messages for the gadget window until the capture is released. Although gadgets are always notified if a left button-down event occurs within the rectangle, the derived gadget class must call *GadgetSetCapture* if you want the gadget to be notified when a mouse drag and a mouse button-up event occurs.

See also TGadgetWindow::GadgetReleaseCapture

GadgetWithId

TGadget* GadgetWithId(int id) const;

Returns a pointer to the gadget associated with the given ID (*id*).

GetDirection

TTileDirection GetDirection() const;

Gets the horizontal or vertical orientation of the gadgets.

See also TGadgetWindow::SetDirection

GetFont

TFont& GetFont();

Returns the font (which is *Sans Serif* by default).

See also TGadgetWindowFont::TGadgetWindowFont

GetFontHeight

uint GetFontHeight() const;

Gets the height of the window's font.

GetHintMode

THintMode GetHintMode();

Returns the hint mode.

IdleAction

void IdleAction();

While no messages are waiting to be processed, *IdleAction* is called and iterates through the gadgets, invoking their *CommandEnable* member function.

See also TGadget::CommandEnable

Insert

virtual void Insert(TGadget& gadget, TPlacement = After, TGadget *sibling = 0);

Inserts a gadget before or after a sibling gadget (*TPlacement*). If *sibling* is 0, then the new gadget is inserted at either the beginning or the end of the gadget list. If this window has already been created, *LayoutSession* needs to be called after inserting gadgets.

See also TGadgetWindow::LayoutSession, TGadgetWindow::Remove

LayoutSession

virtual void LayoutSession();

LayoutSession is typically called when a change occurs in the size of the margins or gadgets or when gadgets are added or deleted. *LayoutSession* calls *TileGadgets* to tile the gadgets in the specified direction and *Invalidate* to mark the area as invalid (needs repainting).

See also TGadgetWindow::Insert, TGadgetWindow::Remove, TWindow::Invalidate

NextGadget

TGadget* NextGadget(TGadget& gadget) const;

Returns the next gadget after *gadget* or 0 if none exists.

Remove

virtual TGadget* Remove(TGadget& gadget);

Removes a gadget from the gadget window. The gadget is returned but not destroyed. *Remove* returns 0 if the gadget is not in the window.

If this window has already been created, the calling application must call *LayoutSession* after any gadgets have been removed.

See also TGadgetWindow::Insert, TGadgetWindow::LayoutSession

SetDirection

virtual void SetDirection(TTileDirection direction);

Sets the horizontal or vertical orientation of the gadgets. If the gadget window is already created, *SetDirection* readjusts the dimensions of the gadget window to fit around the gadgets.

The setting of the direction parameter is also related to the setting of the second parameter (*Tlocation*) in *TDecoratedFrame's Insert* function,. which specifies where the decoration is added in relation to the frame window's client window. If the second parameter in *TDecoratedFrame::Insert* is set to top or bottom, the direction parameter in *SetDirection* must be horizontal. If the second parameter in *TDecoratedFrame::Insert* is set to left or right, the direction parameter in *SetDirection* must be vertical.

See also TGadgetWindow::GetDirection

SetHintCommand

void SetHintCommand(int id);

Simulates menu selection messages so that ObjectWindows command processing can display command hints for the given command *id* (CM_xxxx).

See also CM_xxxx edit constants, CM_xxxx edit file constants

SetHintMode

void SetHintMode(THintMode hintMode);

Sets the mode of the hint text. Defaults to *PressHints* (displays hint text when a button is pressed).

See also THintMode enum

SetMargins

void SetMargins(TMargins& margins);

Sets or changes the margins for the gadget window and calls *LayoutSession*.

See also TGadgetWindow::Margins

SetShrinkWrap

void SetShrinkWrap(bool shrinkWrapWidth, bool shrinkWrapHeight);

Sets the width and height of the data members. By default, if the tile direction is horizontal, *ShrinkWrapWidth* is false and *ShrinkWrapHeight* is **true**. Also by default, if the direction is vertical, *ShrinkWrapWidth* is true and *ShrinkWrapHeight* is **false**.

Protected data members

AtMouse

TGadget* AtMouse;

The last gadget at the mouse position.

BkgndBrush

TBrush* BkgndBrush;

The color of the background brush.

Capture

TGadget* Capture;

Points to the gadget that currently has the mouse capture; otherwise, if no gadget has the mouse capture, *Capture* is 0.

See also TGadgetWindow::GadgetSetCapture

Direction

TTileDirection GetDirection() const;

Gets the horizontal or vertical orientation of the gadgets.

See also TGadgetWindow::SetDirection

DirtyLayout

bool DirtyLayout;

Indicates the layout has changed and gadgets need to be re-tiled. Using *DirtyLayout* avoids redundant tiling when gadget windows are created.

See also TGadgetWindow::LayoutSession

Font

TFont* Font;

Points to the font used to calculate layout units.

See also TGadgetWindow::GetFont

FontHeight

uint GetFontHeight() const;

Gets the height of the window's font.

Gadgets

TGadget* Gadgets;

Points to the first gadget in the gadget list.

HintMode

THintMode HintMode;

Holds the hint text mode.

See also THintMode enum

Margins

TMargins Margins;

Holds the margin values for the gadget window.

See also TGadgetWindow::SetMargins

NumGadgets

uint NumGadgets;

The number of gadgets in the window.

ShrinkWrapHeight

bool ShrinkWrapHeight;

If *ShrinkWrapHeight* is true, the window will shrink its width to fit the tallest gadget for horizontally tiled gadgets.

See also TGadgetWindow::SetShrinkWrap

ShrinkWrapWidth

bool ShrinkWrapWidth;

If *ShrinkWrapWidth* is true, the window will shrink its width to fit the widest gadget for vertically tiled gadgets.

See also TGadgetWindow::SetShrinkWrap

WideAsPossible

uint WideAsPossible;

The number of gadgets that are as wide as possible.

Protected member functions

Create

bool Create();

Overrides *TWindow* member function and chooses the initial size of the gadget if shrink-wrapping was requested.

See also TGadgetWindow::SetShrinkWrap

EvLButtonDown

void EvLButtonDown(uint modKeys, TPoint& point);

Responds to a left button-down mouse message by forwarding the event to the gadget positioned under the mouse.

EvLButtonUp

void EvLButtonUp(uint modKeys, TPoint& point);

Responds to a left button-up mouse message by forwarding the event to the gadget that has the capture.

EvMouseMove

void EvMouseMove(uint modKeys, TPoint& point);

If mouse events are captured, *EvMouseMove* responds to a mouse move message by forwarding the event to the gadget that has the capture.

EvSize

void EvSize(uint sizeType, TSize& size);

Calls *TWindow::EvSize* to perform any default processing. If *DirtyLayout* **true** and *WideAsPossible* is greater than 0, *EvSize* sets *DirtyLayout* to **true** and calls *TileGadgets* to readjust the size and *Invalidate* to mark the area for redrawing.

EvSysColorChange

void EvSysColorChange();

EvSysColorChange, which is called when any system colors have changed, forwards the event to all gadgets.

GetDesiredSize

virtual void GetDesiredSize(TSize& size);

If shrink-wrapping was requested, *GetDesiredSize* returns the size needed to accommodate the borders and the margins of the widest and highest gadget; otherwise, it returns the width and height in the window's *Attr* structure.

If you want to leave extra room for a specific look (for example, a separator line between gadgets, a raised line, and so on), you can override this function. However, if you override *GetDesiredSize*, you will probably also need to override *GetInnerRect* to calculate your custom inner rectangle.

See also TGadgetWindow::GetInnerRect

GetInnerRect

virtual void GetInnerRect(TRect& rect);

GetInnerRect computes the rectangle inside of the borders and margins of the gadget.

If you want to leave extra room for a specific look (for example, a separator line between gadgets, a raised line, and so on), you can override this function. If you override *GetInnerRect*, you will probably also need to override *GetDesiredSize* to calculate your custom total size.

See also TGadgetWindow::GetDesiredSize

GetMargins

```
void GetMargins(TMargins& margins, int& left, int& right, int& top, int& bottom);
```

Returns the left, right, top, and bottom margins in pixels.

LayoutUnitsToPixels

```
int LayoutUnitsToPixels(int units);
```

Converts layout units to pixels. A layout unit is determined by dividing the window font height by eight.

See also TGadgetWindow::LayoutSession

Paint

```
void Paint(TDC& dc, bool erase, TRect& rect);
```

Puts the font into the device context and calls *PaintGadgets*.

See also TGadgetWindow::PaintGadgets

PaintGadgets

```
virtual void PaintGadgets(TDC& dc, bool erase, TRect& rect);
```

Called by *Paint* to repaint all of the gadgets, *PaintGadgets* iterates through the list of gadgets, determines the gadget's area, and repaints each gadget.

You can override this function to implement a specific look (for example, separator line, raised, and so on).

PositionGadget

```
virtual void PositionGadget(TGadget* previous, TGadget* next, TPoint& point);
```

PositionGadget is called to allow spacing adjustments to be made before each gadget is positioned.

See also TGadgetWindow::TileGadgets

TileGadgets

```
virtual void TileGadgets();
```

Tiles the gadgets in the direction requested (horizontal or vertical).

Calls *PositionGadget* to give derived classes an opportunity to adjust the spacing between gadgets in their windows.

See also TGadgetWindow::PositionGadget

Response table entries

Response table entry	Member function
EV_WM_LBUTTONDOWN	*EvLButtonDown*
EV_WM_LBUTTONUP	*EvLButtonUp*
EV_WM_MOUSEMOVE	*EvMouseMove*
EV_WM_SIZE	*EvSize*
EV_WM_SYSCOLORCHANGE	*EvSysColorChange*

TGauge class

gauge.h

A streamable class derived from *TControl*, *TGauge* defines the basic behavior of gauge controls. Gauges are display-only horizontal or vertical controls that provide duration or analog information about a particular process. A typical use of a gauge occurs in installation programs where a control provides a graphical display indicating the percentage of files copied. In general, horizontal gauges with a broken (dashed-line) bar are used to display short-duration, process information whereas horizontal gauges with a solid bar are used to illustrate long-duration, process information. Usually, vertical gauges are preferred for displaying analog information.

Public constructor

Constructor

```
TGauge(TWindow* parent, const char far* title, int id, int X, int Y, int W, int H, bool isHorizontal = true,
  int margin = 0, TModule* module = 0);
```

Constructs a *TGauge* object with borders that are determined by using the value of SM_CXBORDER. Sets *IsHorizontal* to *isHorizontal*. Sets border thickness and spacing between dashed borders (LEDs) to 0. Sets the range of possible values from 0 to 100.

Public member functions

GetRange

```
void GetRange(int& min, int& max) const;
```

Gets the minimum and maximum values for the gauge.

GetValue

```
int GetValue() const;
```

Gets the current value of the gauge.

SetColor

```
void SetColor(TColor color);
```

Sets the *BarColor* data member to the value specified in color.

SetLed

```
void SetLed(int spacing, int thick = 90);
```

Sets the *LedSpacing* and *LedThick* data members to the values *spacing* and *thick*.

SetRange

```
void SetRange(int min, int max);
```

Sets the *Min* and *Max* data members to *min* and *max* values returned by the constructor. If *Max* is less than or equal to *Min*, *SetRange* resets *Max* to *Min* plus 1.

SetValue

```
void SetValue(int value);
```

Restricts the value so that it is within the minimum and maximum values established for the gauge. If the current value has changed, *SetValue* marks the old position for repainting. Then, it sets the data member *Value* to the new value.

Protected data members

BarColor

```
TColor BarColor;
```

Holds the bar or LED color, which defaults to blue.

IsHorizontal

```
int IsHorizontal;
```

Set to the *ishorizontal* argument of the constructor. *IsHorizontal* is **true** if the gauge is horizontal and **false** if it is vertical.

LedSpacing

```
int LedSpacing;
```

Holds the integer value (in gauge units) of the spacing between the broken bars of the gauge. Note that *TGauge* does not paint the title while using LED spacing.

LedThick

```
int LedThick;
```

Holds the thickness of the broken bar.

Margin

```
int Margin;
```

Contains the border width and height of the gauge.

Max

```
int Max;
```

Holds the maximum value (in gauge units) displayed on the gauge.

Min

```
int Min;
```

Holds the minimum value (in gauge units) displayed on the gauge.

Value

```
int Value;
```

Holds the current value of the gauge.

Protected member functions

EvEraseBkgnd

bool EvEraseBkgnd(HDC);

Overrides *TWindow's EvEraseBkgnd* function and erases the background of the gauge. Whenever the background is repainted, *EvEraseBkgnd* is called to avoid flickering.

Paint

void Paint(TDC& dc, bool erase, TRect& rect);

Overrides *TWindow's* Paint function and paints the area and border of the gauge. Paints the given rectangle on the given device context. Uses the values in *LedSpacing* and *IsHorizontal* to paint draw a horizontal or vertical gauge with solid or broken bars.

See also TGauge::LedSpacing, TGauge::IsHorizontal, TDC, TRect

PaintBorder

virtual void PaintBorder(TDC& dc);

Paints the gauge border using the specified device context. Depending on whether the border style is raised, embossed, or recessed, *PaintBorder* paints the specified boundary. You can override this function if you want to implement a border style that isn't supported by ObjectWindows' gauges.

Response table entries

Response table entry	Member function
EV_WM_ERASEBKGND	EvEraseBkgnd

TGdiObject class

gdiobjec.h

GdiObject is the root, pseudo-abstract base class for ObjectWindows' GDI (Graphics Device Interface) wrappers. The *TGdiOject*-based classes let you work with a GDI handle and construct a C++ object with an aliased handle. Some GDI objects are also based on *TGdiObject* for handle management. Generally, the *TGdiObject*-based class hierarchy handles all GDI objects apart from the DC (Device Context) objects handled by the *TDC*-based tree.

The five DC selectable classes *(TPen, TBrush, TFont, TPalette,* and *TBitmap),* and the *TIcon, TCursor, TDib,* and *TRegion* classes, are all derived directly from *TGdiObject.*

TGdiObject maintains the GDI handle and a *ShouldDelete* flag that determines if and when the handle and object should be destroyed. Protected constructors are provided for use by the derived classes: one for borrowed handles, and one for normal use.

An optional orphan control mechanism is provided. By default, orphan control is active, but you can turn it off by defining the NO_GDI_ORPHAN_CONTROL identifier:

```
#define NO_GDI_ORPHAN_CONTROL
```

With orphan control active, the following static member functions are available:

RefAdd, *RefCount*, *RefDec*, *RefFind*, *RefInc*, and *RefRemove*.

These maintain object reference counts and allow safe orphan recovery and deletion. Macros, such as OBJ_REF_ADD, let you deactivate or activate your orphan control code by simply defining or undefining NO_GDI_ORPHAN_CONTROL. When NO_GDI_ORPHAN_CONTROL is undefined, for example, *OBJ_REF_ADD(handle, type)* expands to *TGdiObject::RefAdd((handle),(type))*, but when NO_GDI_ORPHAN_CONTROL is defined, the macro expands to *handle*.

Public destructor

Destructor

~TGdiObject();

If *ShouldDelete* is **false** no action is taken. Otherwise with *ShouldDelete* **true**, the action of the destructor depends on whether orphan control is active or not. If orphan control is inactive (that is, if NO_ORPHAN_CONTROL is defined) *~TGdiObject* deletes the GDI object. If orphan control is active (the default) the object is deleted only if the reference count is 0.

Type definitions

TAutoDelete

enum TAutoDelete{NoAutoDelete, AutoDelete};

This enum, which is defined in the private base class, gdibase.h, enumerates the flag values for GDI Handle constructors. This flag is used to control GDI object deletion in the destructors.

TType

enum TType {None, Pen, Brush, Font, Palette, Bitmap, TextBrush};

This enumeration is used to store the object type in the **struct** *TObjInfo*. This internal structure is used to track object reference counts during debugging sessions.

See also TGdiObject::RefCount

Public member functions

GetObject

int GetObject(int count, void far* object) const;

Obtains information about this GDI object and places it in the *object* buffer. If the call succeeds and *object* is not 0, *GetObject* returns the number of bytes copied to the object buffer. If the call succeeds and *object* is 0, *GetObject* returns the number of bytes needed in the object buffer for the type of object being queried. Depending on what type of GDI object is derived, this function retrieves a *LOGPEN*, *LOGBRUSH*, *LOGFONT*, or *BITMAP* structure through *object*.

See also TPen::GetObject, BITMAP struct, LOGBRUSH struct, LOGFONT struct, LOGPEN struct

RefAdd

static void RefAdd(HANDLE handle, TType type);

Available only if orphan control is active (that is, if NO_GDI_ORPHAN_CONTROL is undefined). *RefAdd* adds a reference entry for the object with the given *handle* and *type* to the *ObjInfoBag* table and sets the reference count to 1. If the table already has a matching entry, no action is taken.

See also TGdiObject::RefCount, macro OBJ_REF_ADD

RefCount

static int RefCount(HANDLE handle);

Available only if orphan control is active, that is, if NO_GDI_ORPHAN_CONTROL is undefined. *RefCount* returns this object's current reference count or –1 if the object is not in the *ObjInfoBag* table.

See also macro OBJ_REF_COUNT

RefDec

static void RefDec(HANDLE handle);
static void RefDec(HANDLE handle, bool wantDelete);

Available only if orphan control is active, that is, if NO_GDI_ORPHAN_CONTROL is undefined. *RefDec* decrements this object's reference count by 1 and deletes the object when the reference count reaches zero. A warning is issued if the deletion was supposed to happen but didn't. Likewise, a warning is issued if the deletion wasn't supposed to happen but did. The deleted object is also detached from the *ObjInfoBag* table.

The second version of *RefDec* is available only if the __TRACE identifier is defined. You can vary the normal deletion strategy by setting *wantDelete* to **true** or **false**.

See also TGdiObject::RefCount, macro OBJ_REF_DEC

RefFind

static TObjInfo* RefFind(HANDLE object);

Available only if orphan control is active (that is, if NO_GDI_ORPHAN the given object is undefined). If found, the object's type and reference count are returned in the specified *TObjInfo* object. *RefFind* returns 0 if no match is found.

See also TGdiObject::RefCount

RefInc

static void RefInc(HANDLE handle);

Available only if orphan control is active (that is, if NO_GDI_ORPHAN_CONTROL is undefined). *RefInc* increments by 1 the reference count of the object associated with *handle*.

See also TGdiObject::RefCount, macro OBJ_REF_INC

RefRemove

static void RefRemove(HANDLE handle);

Available only if orphan control is active (that is, if NO_GDI_ORPHAN_CONTROL is undefined). *RefRemove* removes the reference entry to the object with the given *handle* from the *ObjInfoBag* table. If the given handle is not found, no action is taken.

See also TGdiObject::RefCount, macro OBJ_REF_REMOVE

operator HGDIOBJ()

operator HGDIOBJ() const

Type casting operator that converts this GDI object handle to type HGDIOBJ.

Protected data members

Handle

HANDLE Handle;

The GDI handle of this object.

See also TGdiObject protected constructors

ShouldDelete

bool ShouldDelete;

Set true if the Destructor needs to delete this object's GDI handle.

See also TGdiObject Protected Constructors

Protected member functions

CheckValid

void CheckValid(uint resId=IDS_GDIFAILURE)
static void CheckValid(HANDLE handle, uint resId=IDS_GDIFAILURE)

Both versions of *CheckValid* check for a valid GDI object handle. If one is not found a GDI exception is thrown for the given resource id.

Protected constructors

Constructors

Form 1 TGdiObject();

This default constructor sets *Handle* to 0 and *ShouldDelete* to **true**. This constructor is intended for use by derived classes that must set the *Handle* member.

Form 2 TGdiObject(HANDLE handle, TAutoDelete autoDelete = NoAutoDelete);

This constructor is intended for use by derived classes only. The *Handle* data member is "borrowed" from an existing handle given by the argument *handle* The *ShouldDelete* data member defaults to false ensuring that the borrowed handle will not be deleted when the object is destroyed.

See also TGdiObject::enumTAutoDelete, TGdiObject::Handle, TGdiObject::RefCount, TGdiObject::ShouldDelete

Macros

OBJ_REF_ADD

OBJ_REF_ADD(handle, type)

If orphan control is active (the default), *OBJ_REF_ADD(handle, type)* is defined as *TGdiObject::RefAdd((handle), (type))*. The latter adds to the *ObjInfoBag* table a reference

entry for the object with the given *handle* and *type*, and sets its count to 1. If orphan control is inactive, *OBJ_REF_ADD(handle)* is defined as *handle*. This macro lets you write orphan control code that can be easily deactivated with the single statement #define NO_GDI_ORPHAN_CONTROL.

See also TGdiObject::RefAdd

OBJ_REF_COUNT

OBJ_REF_COUNT(handle)

If orphan control is active (the default), *OBJ_REF_COUNT(handle)* is defined as *TGdiObject::RefCount((handle))*. The latter returns the reference count of the object with the given handle, or –1 if no such object exists. If orphan control is inactive, *OBJ_REF_COUNT(handle)* is defined as –1. This macro lets you write orphan control code that can be easily deactivated with the single statement #define NO_GDI_ORPHAN_CONTROL.

See also TGdiObject::RefCount

OBJ_REF_DEC

OBJ_REF_DEC(handle, wantDelete)

If orphan control is active (the default), *OBJ_REF_DEC(handle, wantDelete)* is defined as either *TGdiObject::RefDec((handle))* or *TGdiObject::RefDec((handle), (wantDelete))*. The latter format occurs only if _ _TRACE is defined. *RefDec(handle)* decrements the reference count of the object associated with *handle* and optionally deletes orphans or warns you of their existence. If orphan control is inactive, *OBJ_REF_DEC(handle)* is defined as *handle*. This macro lets you write orphan control code that can be easily deactivated with the single statement #define NO_GDI_ORPHAN_CONTROL.

See also TGdiObject::RefDec

OBJ_REF_INC

OBJ_REF_INC(handle)

If orphan control is active (the default), *OBJ_REF_INC(handle)* is defined as *TGdiObject::RefInc((handle))*. The latter increments the reference count of the object associated with *handle*. If orphan control is inactive, *OBJ_REF_DEC(handle)* is defined as *handle*. This macro lets you write orphan control code that can be easily deactivated with the single statement #define NO_GDI_ORPHAN_CONTROL.

See also TGdiObject::RefInc

OBJ_REF_REMOVE

OBJ_REF_REMOVE(handle)

If orphan control is active (the default), *OBJ_REF_REMOVE(handle)* is defined as *TGdiObject::RefRemove((handle))*. The latter removes from the *ObjInfoBag* table the reference entry for the object associated with *handle*. If orphan control is inactive, *OBJ_REF_REMOVE(handle)* is defined as *handle*. This macro lets you write orphan control code that can be easily deactivated with the single statement #define NO_GDI_ORPHAN_CONTROL.

See also TGdiObject::RefRemove

TGdiObject::TXGdi class

gdibase.h

Describes an exception resulting from GDI failures such as creating too many *TWindow* DCs. This exception occurs, for example, if a DC driver can't be located or if a DIB file can't be read.

The following code from the PAINT.CPP sample program on your distribution disk throws a *TXGdi* exception if a new DIB can't be created.

```
void TCanvas::NewDib(int width, int height, int nColors)
{
  TDib* dib;
  try {
     dib = new TDib(width, height, nColors);
  }
  catch (TGdiObject::TXGdi& x) {
     MessageBox("Could Not Create DIB", GetApplication()->Name,
  MB_OK);
     return;
  }
```

Public constructor

Constructor

TXGdi(uint resId = IDS_GDIFAILURE, HANDLE = 0);

Constructs a *TXGdi* object with a default IDS_GDIFAILURE message.

Public member functions

Clone

TXOwl* Clone();

Makes a copy of the exception object. *Clone* must be implemented in any class derived from *TXOwl*.

Msg

static string Msg(uint resId, HANDLE);

Converts the resource ID to a string and returns the string message.

Throw

void Throw();

Throws the exception object. *Throw* must be implemented in any class derived from *TXOwl*.

TGroupBox class

groupbox.h

An instance of a *TGroupBox* is an interface object that represents a corresponding group box element. Generally, *TGroupBox* objects are not used in dialog boxes or dialog windows (*TDialog*), but are used when you want to create a group box in a window.

Although group boxes don't serve an active purpose onscreen, they visually unify a group of selection boxes such as check boxes and radio buttons or other controls. Behind the scenes, however, they can take an important role in handling state changes for their group of controls (normally check boxes or radio buttons).

For example, you might want to respond to a selection change in any one of a group of radio buttons in a similar manner. You can do this by deriving a class from *TGroupBox* that redefines the member function *SelectionChanged*.

Alternatively, you could respond to selection changes in the group of radio buttons by defining a response for the group box's parent. To do so, define a child-ID-based response member function using the ID of the group box. The group box will automatically send a child-ID-based message to its parent whenever the radio button selection state changes. This class is streamable.

Public data members

NotifyParent

bool NotifyParent;

Flag that indicates whether parent is to be notified when the state of the group box's selection boxes has changed. *NotifyParent* is **true** by default.

Public constructors

Public constructors

Form 1 TGroupBox(TWindow* parent, int Id, const char far *text, int x, int y, int w, int h, TModule* module = 0);

Constructs a group box object with the supplied parent window (*Parent*), control ID (*Id*), associated text (*text*), position (*x, y*) relative to the origin of the parent window's client area, width (*w*), and height (*h*). Invokes the *TControl* constructor with similar parameters, then modifies *Attr.Style*, adding BS_GROUPBOX and removing WS_TABSTOP. *NotifyParent* is set to true; by default, the group box's parent is notified when a selection change occurs in any of the group box's controls.

Form 2 TGroupBox(TWindow* parent int resourceId, TModule* module = 0);

Constructs a *TGroupBox* object to be associated with a group box control of a *TDialog*. Invokes the *TControl* constructor with identical parameters *resourceID* must correspond to a group box resource that you define.

See also TControl::TControl, TWindow::DisableTransfer

Public member functions

GetClassName

char far* GetClassName();

GetClassName returns the name of *TGroupBox's* Windows registration class, "BUTTON." If BWCC is enabled, *GetClassName* returns BUTTON_CLASS.

SelectionChanged

virtual void SelectionChanged(int controlId);

If *NotifyParent* is **true**, *SelectionChanged* notifies the parent window of the group box that one of its selections has changed by sending it a child-ID-based message. This member function can be redefined to allow the group box to handle selection changes in its group of controls.

THatch8x8Brush class

gdiobjec.h

Derived from *TBrush*, *THatch8x8Brush* defines a small, 8x8, monochrome, configurable hatch brush (a brush that fills an area with a pattern created from hatch marks). Because the hatch brush is a logical brush created from device independent bitmaps (DIBs), it can be passed to any DC, which then renders the brush into the appropriate form for the particular device.

Although the default brush's color is a white foreground and a black background, you can vary the colors of the hatched brush. The colors can be any one of the *TColor* object encapsulated colors, namely the standard RGB values.

THatch8x8Brush contains static arrays that define common hatched brush patterns. The hatched brush patterns you can select include

Forward Diagonal / / / / /
// // // // //

Backward Diagonal \ \ \ \ \
\\ \\ \\ \\ \\

You can use *THatch8x8Brush* to design a variety of hatched brush border patterns around a simple rectangle or an OLE container. You can also use *THatch8x8Brush* in conjunction with *TUIHandle*.

Public data members

Hatch11F1[8]

const static uint8 Hatch11F1[8];

The static array, *Hatch11F1[8]* holds the logical hatched brush pattern of 1 pixel on and 1pixel off in monochrome, offset 1 per row as the following pattern illustrates.

Hatch13B1[8]

const static uint8 Hatch13B1[8];

The static array, *Hatch13B1[8]* holds a hatched brush pattern of 1 pixel on and 3 pixels off in backward diagonal hatch marks, offset 1 per row as the following pattern illustrates.

Hatch13F1[8]

const static uint8 Hatch13F1[8];

The static array, *Hatch13F1[8]* holds a hatched brush pattern of 1 pixel on and 3 pixels off in forward diagonal hatch marks, offset 1 per row as the following pattern illustrates.

Hatch22B1[8]

const static uint8 Hatch22B1[8];

The static array, *Hatch22B1[8]* holds a hatched brush pattern of 2 pixels on and 2 off in backward diagonal hatch marks, offset 1 per row as the following pattern illustrates.

Hatch22F1[8]

const static uint8 Hatch22F1[8];

The static array, *Hatch22F1[8]* holds a hatched brush pattern of 2 pixels on and 2 off in forward diagonal hatch marks, offset 1 per row as the following pattern illustrates.

Public constructors

Constructor

THatch8x8Brush(const uint8 hatch[], TColor fgColor=TColor::White, TColor bgColor=TColor::Black);

Constructs a *THatchBrush* object with the specified hatched pattern and colors. Although the hatched brush is, by default, white on a black background, you can control the displayed colors by passing different colors in the constructor where *fgColor* represents the foreground color and *bgColor* represents the background color of a *TColor* object type. Colors can be specified in either palette (a reference to a corresponding color palette entry in the currently realized palette) or RGB mode (an actual red, green, or blue color value).

Public member functions

Reconstruct

void Reconstruct(const uint8 hatch[], TColor fgColor, TColor bgColor);

Reconstructs the hatched brush with a new pattern or new set of colors.

See also TBrush, TColor, TUIHandle

TIC class

dc.h

Derived from *TDC*, *TIC* is a DC class that provides a constructor for creating a DC object from explicit driver, device, and port names.

Public constructor

Constructor

TIC(const char far* driver, const char far* device, const char far* output, const DEVMODE far* initData=0);

Creates a DC object with the given driver, device, and port names and initialization values.

See also TDC::GetDeviceCaps, DEVMODE struct

TIcon class

gdiobjec.h

TIcon, derived from *TGdiObject*, represents the GDI object icon class. *TIcon* constructors can create icons from a resource or from explicit information. Because icons are not real GDI objects, the *TIcon* destructor overloads the base destructor, *~TGdiObject*.

Public constructors and destructor

Constructors

Form 1 TIcon(HICON handle, TAutoDelete autoDelete = NoAutoDelete);

Creates a *TIcon* object and sets the *Handle* data member to the given borrowed handle. The *ShouldDelete* data member defaults to **false**, ensuring that the borrowed handle will not be deleted when the C++ object is destroyed.

Form 2 TIcon(HINSTANCE instance, const TIcon& icon);

Creates a copy of the given *icon* object.

Form 3 TIcon(HINSTANCE instance, TResID resID);

Creates an icon object from the given resource.

Form 4 TIcon(HINSTANCE instance, const char far* filename, int index);

Creates an icon object from the given resource file.

Form 5 TIcon(HINSTANCE instance, const TSize& size, int planes, int bitsPixel, const void far* andBits, const void far* xorBits);

Creates an icon object with the given values.

Form 6 TIcon(const void* resBits, uint32 resSize);

Creates an icon object of the given size from the bits found in the *resBits* buffer.

Form 7 TIcon(const ICONINFO* iconInfo);

Creates an icon object with the given *ICONINFO* information.

Destructor

~TIcon();

Destroys the icon and frees any memory that the icon occupied.

See also ~TGdiObject, TGdiObject::Handle, TGdiObject::ShouldDelete,, TResID, TSize, ICONINFO structure

Public member functions

GetIconInfo
bool GetIconInfo(ICONINFO* iconInfo) const;
Retrieves information about this icon and copies it into the given *ICONINFO* structure. Returns **true** if the call is successful; otherwise returns **false**.

See also ICONINFO structure

operator HICON()
operator HICON() const;
Typecasting operator that converts this icon's *Handle* to type *HICON* (the data type representing the handle to an icon resource).

TInputDialog class

inputdia.h

TInputDialog provides a generic dialog box to retrieve text input by a user. When the input dialog box is constructed, its title, prompt, and default input text are specified. *TInputDialog* is a streamable class.

Public data members

buffer
char far* buffer;
Pointer to the buffer that returns the text retrieved from the user. When passed to the constructor of the input dialog box, contains the default text to be initially displayed in the edit control.

BufferSize
int BufferSize;
Contains the size of the buffer that returns user input.

prompt
char far* prompt;
Points to the prompt for the input dialog box.

Public constructor

Constructor
TInputDialog(TWindow* parent, const char far *title, const char far *prompt, char far* buffer, int buffersize,
TModule* module = 0, TValidator* valid = 0)
Invokes *TDialog's* constructor, passing it *parent*, the resource identifier and *module*. Sets the caption of the dialog box to *title* and the prompt static control to *prompt*. Sets the *Buffer* and *BufferSize* data members to *buffer* and *bufferSize*.

See also TDialog::TDialog

Public member function

TransferData
void TransferData(TTransferDirection direction);
Transfers the data of the input dialog box. If *direction* is *tdSetData*, sets the text of the static and edit controls of the dialog box to the text in *prompt* and *buffer*. If *direction* is *tdGetData*, fills the *buffer* with the current text of the *Editor*.

Protected member function

SetupWindow
virtual void SetupWindow();
In setting up the window, *SetupWindow* calls *TDialog::SetupWindow*, then limits the number of characters the user can enter to *bufferSize* –1.

TInStream class

docview.h

Derived from *TStream* and *istream*, *TInStream* is a base class used for defining input streams for documents.

Public constructor

Constructor
TInStream(TDocument& doc, const char far* name, int mode);
Constructs a *TInStream* object. *doc* refers to the document object, *name* is the user-defined name of the stream, and *mode* is the mode of opening the stream.

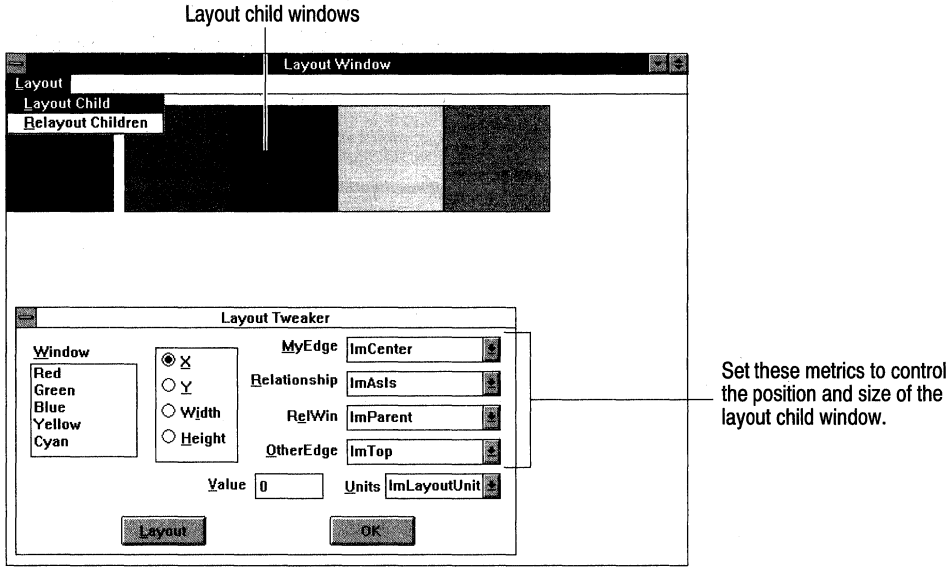
See also TOutStream, ofXXXX document open enum, shxxxx document sharing enum

TLayoutConstraint struct

layoutco.h

TLayoutConstraint is a structure that defines a relationship (a layout constraint) between an edge or size of one window and an edge or size of one of the window's siblings or its parent. If a parent-child relationship is established between windows, the dimensions of the child windows are dependent on the parent window. A window can have one of its sizes depend on the size of the opposite dimension. For example, the width can be twice the height. *TLayoutMetrics* lists the relationships you can have among size and edge constraints.

The sample file LAYOUT.CPP shows you the following example of how to set up layout constraints.



Public data members

MyEdge

uint MyEdge;

MyEdge contains the name of the edge or size constraint (*lmTop, lmBottom, lmLeft, lmRight, lmCenter, lmWidth,* or *lmHeight*) for your window.

See also TWidthHeight enum

OtherEdge

uint OtherEdge;

OtherEdge contains the name of the edge or size constraint (*lmTop, lmBottom, lmLeft, lmRight, lmCenter, lmWidth,* or *lmHeight*) for the other window.

See also TWidthHeight enum

Relationship

TRelationship Relationship;

Relationship specifies the type of relationship that exists between the two windows (that is, *lmRightOf, lmLeftOf, lmAbove, lmBelow, lmSameAs,* or *lmPercentOf*). A value of *lmAbsolute* actually indicates that no relationship exists.

See also TRelationship enum

RelWin

TWindow *RelWin;

RelWin is a pointer to the sibling windows or *lmParent* if the child is a proportion of the parent's dimensions. *RelWin* points to the window itself (**this**) if a child window's dimension is a proportion of one of its other dimensions (for example, its height is a proportion of its width).

See also TRelationship enum

Units

```
TMeasurementUnits Units;
```

Units enumerates the units of measurement (either pixels or layout units) used to measure the height and width of the windows. Unlike pixels, layout units are based on system font size and will be consistent in their perceived size even if the screen resolution changes.

See also TMeasurementUnits enum

union

```
union {
   int Margin;
   int Value;
   int Percent;
};
```

This union is included for the convenience of naming the layout constraints. Margin is used for the *lmAbove*, *lmLeftOf*, *lmLeftOf*, or *lmRightOf* enumerated values in *TRelationship*. Value is used for the *lmSameAs* or *lmAbsolute* enumerated values in *TRelationship*. Percent is used for the *lmPercentOf* enumerated value in *TRelationship*.

See also TMeasurementUnits enum

TLayoutMetrics class — layoutwi.h

TLayoutMetrics contains the four layout constraints used to define the layout metrics for a window. This table lists the constraints you can use for the *X*, *Y*, *Height*, and *Width* fields.

Field	Constraints
X	lmLeft, lmCenter, lmRight
Y	lmTop, lmCenter, lmBottom
Height	lmCenter, lmRight, lmWidth
Width	lmCenter, lmBottom, lmHeight

If the metrics for the child window are relative to the parent window, the relation window pointer (*lmParent*) needs to be *lmParent* (not the actual parent window pointer). For example,

```
TWindow* child = new TWindow(this, "");
TLayoutMetrics metrics;
metrics.X.Set(lmCenter, lmSameAs, lm()t, lmCenter);
metrics.Y.Set(lmCenter, lmSameAs, lm()t, lmCenter);
SetChildLayoutMetrics(*child, metrics);
```

The parent window pointer (**this**) should not be used as the relation window pointer of the child window.

Public data members

Height

TEdgeOrWidthConstraint Height;

Contains the height size constraint, center edge, or bottom edge constraint of the window.

Width

TEdgeOrWidthConstraint Width;

Contains the width size constraint, center edge, or right edge (*lmRight*) constraint of the window.

X, Y

TEdgeConstraint X, Y;

X contains the *X* (left, center, right) edge constraint of the window. *Y* contains the *Y* (top, center, bottom) edge constraint of the window.

Public constructor

Constructor

TLayoutMetrics();

Creates a *TLayoutMetrics* object and initializes the object by setting the units for the child and parent window to the specified layout units and the relationship between the two windows to what is defined in *ImAsIs* (of *TRelationship*). Sets the following default values:

```
X.RelWin = 0;
X.MyEdge = lmLeft;
X.Relationship = lmAsIs;
X.Units = lmLayoutUnits;
X.Value = 0;
Y.RelWin = 0;
Y.MyEdge = lmTop;
Y.Relationship = lmAsIs;
Y.Units = lmLayoutUnits;
Y.Value = 0;
Width.RelWin = 0;
Width.MyEdge = lmWidth;
Width.Relationship = lmAsIs;
Width.Units = lmLayoutUnits;
Width.Value = 0;
Height.RelWin = 0;
Height.MyEdge = lmHeight;
Height.Relationship = lmAsIs;
Height.Units = lmLayoutUnits;
Height.Value = 0;
```

The following program creates two child windows and a frame into which you can add layout constraints.

```
#include <owl\owl.h>
#include <owl\framewin.h>
#include <owl\applicat.h>
#include <owl\layoutwi.h>
#include <owl\decorate.h>
#include <owl\decmdifr.h>
#include <owl\layoutco.h>
#pragma hdrstop

// Create a derived class. //

class TMyDecoratedFrame : public TDecoratedFrame {
public:
TMyDecoratedFrame(TWindow* parent, const char far* title, TWindow& clientWnd, TWindow* MyC
hildWindow);
   void SetupWindow();
   {
      TDecoratedFrame::SetupWindow();
      MyChildWindow->ShowWindow(SW_NORMAL);
      MyChildWindow->BringWindowToTop();
   }
};

// Setup a frame window  //

TMyDecoratedFrame::TMyDecoratedFrame(TWindow * parent, const char far *
title, TWindow&  clientWnd)
   : TDecoratedFrame(parent, title, clientWnd),
      TFrameWindow(parent, title, &clientWnd),
      TWindow(parent, title)
{
// Create a new TMyChildWindow. //

   MyChildWindow = new TWindow(this, "");
   MyChildWindow->Attr.Style |=  WS_BORDER |WS_VISIBLE |WS_CHILD;
   MyChildwindow->SetBkgndColor(RGB(0,100,0));

// Establish metrics for the child window. //

   TLayoutMetrics  layoutMetrics;

   layoutMetrics.X.Absolute(lmLeft, 10);
   layoutMetrics.Y.Absolute(lmTop, 10);
   layoutMetrics.Width.Absolute( 80 );
   layoutMetrics.Height.Absolute( 80 );
}
   SetChildLayoutMetrics(*MyChildWindow, layoutMetrics);
class TMyApp : public TApplication {
public:
```

```
    virtual void InitMainWindow()
    {
      TWindow* client = new TWindow(0, "title");
      MainWindow = new TMyDecoratedFrame(0, "Layout Window Ex", *client);
    }
};
int OwlMain(int, char**) {
  return TMyApp.Run();
}
```

TLayoutWindow class

layoutwi.h

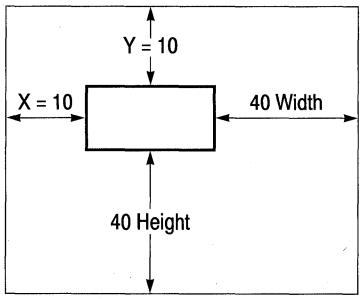
Derived from *TWindow*, *TLayoutWindow* provides functionality for defining the layout metrics for a window. By using layout constraints, you can create windows whose position and size are proportional to another window's dimensions. In other words, the one window constrains the size of the other window. Toolbars and status bars are examples of constrained windows. See *TLayoutConstraint* for a definition of the layout constraints and *TLayoutMetrics* for a description of the metrics you can use to set up layout constraints.

The following examples show how to set up various metrics using edge constraints. For purposes of illustration, these examples use a parent-child relationship, but you can also use a child-to-child (sibling) relationship. Keep in mind that, as usual, if you move the parent's origin (the left and top edges), the child will move with the parent window.

Examples

Example 1

To create growable windows, set the top and left edges of the child window's boundaries in a fixed relationship to the top and left edges of the parent's window. In this example, if you expand the bottom and right edges of the parent, the child's bottom and right edges grow the same amount. Both the X and Y constraints are 10 units from the parent window's edges. Both the Width and Height constraints are 40 layout units from the parent window's edges. Specifically, Width *(lmWidth)* is 40 units to the left of the parent's right edge (*lmLeftOf* = *lmSameAs* + offset or *sameas* – 40).

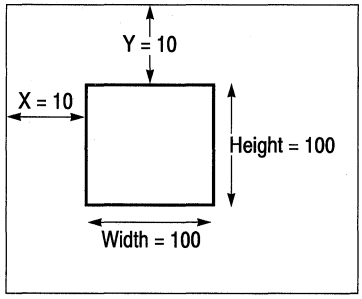


Use the following layout constraints:

```
layoutmetrics.X.Set(lmLeft, lmRightOf, lm()t, lmLeft, 10);
layoutmetrics.Y.Set(lmTop, lmBelow, lm()t, lmTop, 10);
layoutmetrics.Width.Set(lmRight, lmLeftOf, lmParent, lmRight, 40);
layoutmetrics.Height.Set(lmBottom, lmAbove, lmParent, lmBottom, 40);
SetChildLayoutMetrics(*MyChildWindow, layoutMetrics);
```

Example 2

To create fixed-size and fixed-position windows, set the child's right edge a fixed distance from parent's left edge and the child's bottom edge a fixed distance from the parent's top edge. In this example, both the X and Y edge constraints are set to 10 and both the Width and Height edge constraints are set to 100.

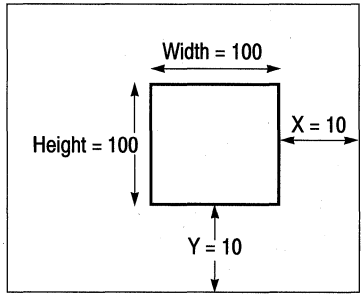


Use the following layout constraints:

```
layoutmetrics.X.Set(lmLeft, lmRightOf, lmParent, lmLeft, 10);
layoutmetrics.Y.Set(lmTop, lmBelow, lmParent, lmTop, 10);
layoutmetrics.Width.Absolute(100);
layoutmetrics.Height.Absolute(100);
SetChildLayoutMetrics(*MyChildWindow, layoutMetrics);
```

Example 3

To create a fixed-size window that remains a constant distance from the parent's right corner, set the child's top and bottom edges a fixed distance (*lmLayout* unit or pixels) from the parent window's bottom. Also, set the child's left and right edges a fixed distance from the parent's right edge. In this example, both the Width and the Height edge constraints are set to 10 and the X and Y edge constraints are set to 100. In this case, the child window, which stays the same size, moves with the lower right corner of the parent.



Use the following layout constraints:

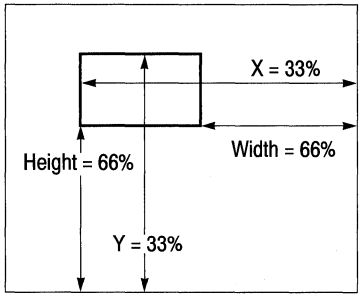
```
layoutmetrics.X.Set(lmRight, lmLeftOf, lmParent, lmRight, 10);
```

```
layoutmetrics.Y.Set(lmBottom, lmAbove, lmParent, lmBottom, 10);
layoutmetrics.Width.Absolute(100);
layoutmetrics.Height.Absolute(100);
SetChildLayoutMetrics(*MyChildWindow, layoutMetrics);
```

Example 4

To create a window in which the child's edges are a percentage of the parent's window, set the child's edges a percentage of the distance from the parent's edges. Specifically, the child's top and bottom edges are a percentage of the parent's bottom edge. The child's left and right edges are a percentage of the parent's right edge.

If you resize the parent window, the child window will change size and origin (that is, the top and left edges will also change).



Use the following layout constraints:

```
layoutmetrics.X.Set(lmLeft, lmPercentOf, lmParent, lmRight, 33);
layoutmetrics.Y.Set(lmTop, lmPercentOf, lmParent, lmBottom, 33);
layoutmetrics.Width.Set(lmRight, lmPercentOf, lm()t, lmRight, 66);
layoutmetrics.Height.Set(lmBottom, lmPercentOf, lm()t, lmBottom, 66);
SetChildLayoutMetrics(*MyChildWindow, layoutMetrics);
```

Public constructor and destructor

Constructor

TLayoutWindow(TWindow* parent, const char far *title = 0, TModule* module = 0);

Creates a *TLayoutWindow* object with specified parent, window caption, and library ID.

Destructor

~TLayoutWindow();

Deletes variables and frees the child metrics and constraints.

Public member functions

GetChildLayoutMetrics

bool GetChildLayoutMetrics(TWindow &child, TLayoutMetrics &metrics);

Gets the layout metrics of the child window.

Layout

void Layout();

Causes the window to resize and position its children according to the specified metrics. Call *Layout* to implement changes that occur in the layout metrics.

RemoveChildLayoutMetrics

bool RemoveChildLayoutMetrics(TWindow &child);

Removes the layout metrics for a child window.

SetChildLayoutMetrics

void SetChildLayoutMetrics(TWindow &child, TLayoutMetrics &metrics);

Sets the metrics for the window and removes any existing ones. Set the metrics as shown:

```
layoutMetrics->X.Absolute(lmLeft, 10);
layoutMetrics->Y.Absolute(lmTop, 10);
layoutMetrics->Width.Set(lmWidth, lmRightOf, GetClientWindow(), lmWidth, -40);
layoutMetrics->Height.Set(lmHeight, lmRightOf, GetClientWindow(), lmHeight, -40);
```

Then call *SetChildLayoutMetrics* to associate them with the position of the child window:

```
SetChildLayoutMetrics(* MyChildWindow, * layoutMetrics);
```

Protected data member

ClientSize

TSize ClientSize;

Contains the size of the client area.

Protected member functions

EvSize

void EvSize(uint sizeType, TSize& size);

Responds to a change in window size by calling *Layout* to resize the window.

Response table entries

Response table entry	Member function
EV_WM_SIZE	EvSize

TListBox class

listbox.h

A *TListBox* is an interface object that represents a corresponding list box element. A *TListBox* must be used to create a list box control in a parent *TWindow*. A *TListBox* can be used to facilitate communication between your application and the list box controls of a *TDialog*. *TListBox*'s member functions also serve instances of its derived class, *TComboBox*. From within MDI child windows, you can access a *TListBox* object by using *TFrameWindow::GetClientWindow*. *TListBox* is a streamable class.

Public constructors

Constructors

Form 1 TListBox(TWindow* parent, int Id, int x, int y, int w, int h, TModule* module = 0);

Constructs a list box object with the supplied parent window (*parent*) library ID (*module*), position (*x*, *y*) relative to the origin of the parent window's client area, width (*w*), and height (*h*). Invokes a *TControl* constructor. Adds LBS_STANDARD to the default styles for the list box to provide it with

- A border (WS_BORDER)
- A vertical scroll bar (WS_VSCROLL)
- Automatic alphabetic sorting of list items (LBS_SORT)
- Parent window notification upon selection (LBS_NOTIFY)

The *TListBox* member functions that are described as being for single-selection list boxes are inherited by *TComboBox* and can also be used by combo boxes. Also, these member functions return –1 for multiple-selection list boxes.

Form 2 TListBox(TWindow* parent, int resourceId, TModule* module = 0)

Constructs a *TListBox* object to be associated with a list box control of a *TDialog*. Invokes the *TControl* constructor with similar parameters. The *module* parameter must correspond to a list box resource that you define.

See also GetSelIndex, GetSelString, SetSelIndex, SetSelString, TControl::TControl

Public member functions

AddString

virtual int AddString(const char far* str);

Adds *string* to the list box, returning its position in the list (0 is the first position). Returns a negative value if an error occurs. The list items are automatically sorted unless the style LBS_SORT is not used for list box creation.

See also TListBox::DeleteString, TListBox::InsertString

ClearList

virtual void ClearList();

Clears all items in the list.

DeleteString

virtual int DeleteString(int index);

Deletes the item in the list at the position (starting at 0) supplied in *index*. *DeleteString* returns the number of remaining list items, or a negative value if an error occurs.

See also TListBox::AddString, TListBox::InsertString

DirectoryList

virtual int DirectoryList(uint attrs, const char far* fileSpec)

Adds a list of file names to a list box.

FindExactString

```
int FindExactString(const char far* str, int searchIndex) const;
```

Starting at the line number passed in *searchIndex*, searches the list box for an exact match with the string *str*. If a match is not found after the last string has been compared, the search continues from the beginning of the list until a match has been found or until the list has been completely traversed. Searches from the beginning of the list when –1 is supplied as *searchIndex*. Returns the index of the first string found if successful, a negative value if an error occurs.

See also TListBox::AddString, TListBox::DeleteString

FindString

```
virtual int FindString(const char far* str, int Index) const;
```

Searches the list box as described under *FindExactString*, but looks for the first entry that begins with *str*.

See also TListBox::AddString, TListBox::DeleteString, TListBox::InsertString

GetCaretIndex

```
int GetCaretIndex() const;
```

Returns the index of the currently focused list-box item. For single-selection list boxes, the return value is the index of the selected item, if one is selected.

See also TListBox::SetCaretIndex

GetCount

```
virtual int GetCount() const;
```

Returns the number of items in the list box, or a negative value if an error occurs.

GetHorizontalExtent

```
int GetHorizontalExtent() const;
```

Returns the number of pixels by which the list box can be scrolled horizontally.

See also TListBox::SetHorizontalExtent

GetItemData

```
virtual uint32 GetItemData(int index) const;
```

Returns the 32-bit value of the list box item set by *SetItemData*.

See also TListBox::SetItemData

GetItemHeight

```
virtual int GetItemHeight(int index) const;
```

Returns the height in pixels of the specified list box items.

See also TListBox::SetItemHeight

GetItemRect

```
int GetItemRect(int index, TRect& rect) const;
```

Returns the dimensions of the rectangle that surrounds a list-box item currently displayed in the list-box window.

GetSel

```
bool GetSel(int index) const;
```

Returns the selection state of the list-box item at location *index*. Returns true if the list-box item is selected, false if not selected.

See also TListBox::SetSel

GetSelCount

int GetSelCount() const;

Returns the number of selected items in the single- or multiple-selection list box or combo box.

GetSelIndex

virtual int GetSelIndex() const;

For single-selection list boxes. Returns the nonnegative index (starting at 0) of the currently selected item, or a negative value if no item is selected.

See also TListBox::SetSelIndex

GetSelIndexes

int GetSelIndexes(int* indexes, int maxCount) const;

For multiple-selection list boxes. Fills the *indexes* array with the indexes of up to *maxCount* selected strings. Returns the number of items put in *indexes* (–1 for single-selection list boxes and combo boxes).

See also TListBox::SetSelIndexes

GetSelString

int GetSelString(char far* str, int maxChars) const;

Retrieves the currently selected items, putting up to *maxChars* of them in *Strings*. Each entry in the *Strings* array should have space for *maxChars* characters and a terminating null. For single-selection list boxes, returns the string length, a negative value if an error occurs, or 1 if no string is selected. For multiple-selection list boxes, returns –1.

See also TListBox::SetSelString

GetSelStrings

int GetSelStrings(char far** strs, int maxCount, int maxChars) const;

Retrieves the total number of selected items for a multiselection list and copies them into the buffer. *str* is an array of pointers to chars. Each of the pointers to the buffers is of *maxChars*. *maxCount* is the size of the array.

See also TListBox::SetSelStrings

GetString

virtual int GetString(char far* str, int index) const;

Retrieves the item at the position (starting at 0) supplied in *index* and returns it in *str*. *GetString* returns the string length, or a negative value if an error occurs.

GetStringLen

virtual int GetStringLen(int Index) const;

Returns the string length (excluding the terminating NULL) of the item at the position index supplied in *Index*. Returns a negative value in the case of an error.

GetTopIndex

int GetTopIndex() const;

Returns the index of the first item displayed at the top of the list box.

See also TListBox::SetTopIndex

InsertString

virtual int InsertString(const char far* str, int index);

Inserts *str* in the list box at the position supplied in *index*, and returns the item's actual position (starting at 0) in the list. A negative value is returned if an error occurs. The list is not resorted. If *index* is –1, the string is appended to the end of the list.

See also TListBox::AddString, TListBox::DeleteString, TListBox::FindString

SetCaretIndex

int SetCaretIndex(int index, bool partScrollOk);

Sets the focus to the item specified at *index*. An item that is not visible is scrolled into view.

See also TListBox::GetCaretIndex

SetColumnWidth

void SetColumnWidth(int width);

Sets the width in pixels of the items in the list box.

SetHorizontalExtent

void SetHorizontalExtent(int horzExtent);

Sets the number of pixels by which the list box can be scrolled horizontally.

See also TListBox::GetHorizontalExtent

SetItemData

virtual int SetItemData(int index, uint32 itemData);

Sets the 32-bit value of the list box item at the specified *index* position.

See also TListBox::GetItemData

SetItemHeight

virtual int SetItemHeight(int index, int height);

Sets the height in pixels of the items in the list box.

See also TListBox::GetItemHeight

SetItemRect

int SetItemRect(int index, TRect& rect) const;

Sets the dimensions of the rectangle that surrounds a list-box item currently displayed in the list-box window.

SetSel

int SetSel(int index, bool select);

Selects an item at the position specified in *index*. For multiple-selection list boxes.

See also TListBox::GetSel

SetSelIndex

virtual int SetSelIndex(int index);

For single-selection list boxes. Forces the selection of the item at the position (starting at 0) supplied in *index*. If *index* is –1, the list box is cleared of any selection. *SetSelIndex* returns a negative number if an error occurs.

SetSelIndexes

int SetSelIndexes(int* indexes, int numSelections, bool shouldSet);

For multiple-selection list boxes. Selects/deselects the strings in the associated list box at the indexes specified in the *Indexes* array. If *ShouldSet* is true, the indexed strings are selected and highlighted; if *ShouldSet* is false the highlight is removed and they are no longer selected. Returns the number of strings successfully selected or deselected (–1 for single-selection list boxes and combo boxes). If *NumSelections* is less than 0, all strings are selected or deselected, and a negative value is returned on failure.

SetSelItemRange

int SetSelItemRange(bool select, int first, int last);

Selects the range of items specified from *first* to *last*.

SetSelString

int SetSelString(const char far* str, int searchIndex);

For single-selection list boxes. Forces the selection of the first item beginning with the text supplied in *str* that appears beyond the position (starting at 0) supplied in *SearchIndex*. If *SearchIndex* is –1, the entire list is searched, beginning with the first item. *SetSelString* returns the position of the newly selected item, or a negative value in the case of an error.

SetSelStrings

int SetSelStrings(const char far** prefixes, int numSelections, bool shouldSet);

For multiple-selection list boxes, selects the strings in the associated list box that begin with the prefixes specified in the *prefixes* array. For each string, the search begins at the beginning of the list and continues until a match is found or until the list has been completely traversed. If *shouldSet* is true, the matched strings are selected and highlighted; if *shouldSet* is false the highlight is removed from the matched strings and they are no longer selected. Returns the number of strings successfully selected or deselected (–1 for single-selection list boxes and combo boxes). If *numSelections* is less than 0, all strings are selected or deselected, and a negative value is returned on failure.

SetTabStops

bool SetTabStops(int numTabs, int far* tabs);

Sets tab stops. *numTabs* is the number of tabstops. *tabs* is the array of integers representing the tab positions.

SetTopIndex

int SetTopIndex(int index);

Sets *index* to the first item displayed at the top of the list box.

See also TListBox::GetTopIndex

Transfer

uint Transfer(void *buffer, TTransferDirection direction);

Transfers the items and selection(s) of the list box to or from a transfer buffer if *tdSetData* or *tdGetData*, respectively, is passed as the *direction*. *buffer* is expected to point to a pointer to a *TListBoxData* structure.

Transfer, which overrides the *TWindow* virtual member function, returns the size of *TListBoxData* (the pointer, not the structure). To retrieve the size without transferring data, pass *tdSizeData* as the *direction*.

You must use a pointer in your transfer buffer to these structures. You can't embed copies of the structures in your transfer buffer, and you can't use these structures as transfer buffers.

See also TListBoxData, TWindow::Transfer

Protected member function

GetClassName
char far* GetClassName();
Returns the name of *TListBox*'s registration class, "LISTBOX".

TListBoxData struct

listbox.h

Used to transfer the contents of a list box.

Public data members

ItemDatas
TDwordArray* ItemDatas;
Contains all **uint32** item data for each item in the list box.

SelCount
int SelCount;
Holds the number of selected items.

SelIndices
TIntArray* SelIndices;
Contains the indexes of all the selected strings in a multiple-selection list box.

SelStrings
TStringArray* SelStrings;
Pointer to an array of the strings to select when data is transferred into the list box. When data is transferred out of the list box, *SelStrings* returns the current selection(s).

Strings
TStringArray* Strings;
Pointer to an array of strings to be transferred into a *TListBox*.

Public constructor and destructor

Constructor

TListBoxData();
Constructs *Strings* and *SelStrings*. Initializes *SelCount* to 0.

Destructor

~TListBoxData();
Deletes the space allocated for *Strings* and *SelStrings*.

Public member functions

AddString

void AddString(const char *str, bool isSelected = false);
Adds the specified string to *Strings*. If *IsSelected* is **true**, adds the string to *SelStrings* and increments *SelCount*.

AddStringItem

void AddStringItem(const char* str, uint32 itemData, bool isSelected = false));
Adds a string to the *Strings* array, optionally selects it, and adds item data to the *ItemDatas* array.

Clear

void Clear();
Resets the list box by flushing the *ItemDatas* and *Strings* arrays and calling *ResetSelections*.

See also TListBoxData::Strings, TListBoxData::ItemDatas, TListBoxData::ResetSelections

GetItemDatas

TDwordArray& GetItemDatas();
Returns a pointer to the *ItemDatas* array.

See also TListBoxData::ItemDatas

GetSelIndices

TIntArray& GetSelIndices();
Returns a pointer to the *SelIndices* array.

See also TListBoxData::SelIndices

GetSelString

void GetSelString(char far* buffer, int bufferSize, int index = 0) const;
Locates the string at the specified *index* in *SelStrings* and copies it into *buffer*. *bufferSize* includes the terminating NULL.

GetSelStringLength

int GetSelStringLength(int index = 0) const;
Returns the length (excluding the terminating NULL) of the string at the specified *index* in *SelStrings*.

GetStrings
TStringArray& GetStrings();
Returns a pointer to the *Strings* array.

See also TListBoxData::Strings

ResetSelections
void ResetSelections();
Removes all strings from *SelStrings* and sets *SelCount* to 0.

Select
void Select(int index);
Selects the string at the given *index*.

SelectString
void SelectString(const char far* str);
Adds *str* to *SelStrings* and increments *SelCount*.

TListView class

listview.h

Derived from *TListBox* and *TView*, *TListView* provides views for list boxes. See *TView* for a description of view functions and *TListBox* for list box functions.

Public constructor and destructor

Constructor
TListView(TDocument& doc, TWindow* parent = 0);
Creates a *TListView* object associated with the specified document and parent window. Sets *Attr.AccelTable* to IDA_LISTVIEW to identify the edit view. Sets the view style to WS_HSCROLL | LBS_NOINTEGRALHEIGHT.

Sets *TView::ViewMenu* to the new *TMenuDescr* for this view.

Destructor
~TListView();
After checking to see if there is an open view, this destructor destroys the *TListView* object.

Public data member

DirtyFlag
bool DirtyFlag;
Is nonzero if the data in the list view has been changed; otherwise, is 0.

Public member functions

CanClose

bool CanClose();

Checks to see if all child views can be closed before closing the current view. If any child returns 0, *CanClose* returns 0 and aborts the process. If all children return nonzero, it calls *TDocManager::FlushDoc*.

See also TDocManager::FlushDoc

Create

virtual bool Create();

Overrides *TWindow::Create* and creates the view's window. Determines if the file is new or already has data. If there is data, calls *LoadData* to add the data to the view. If the view's window can't be created, *Create* throws a *TXInvalidWindow* exception.

GetViewName

const char far* GetViewName();

Overrides *TView's* virtual function and returns the descriptive name of the class (*StaticName*).

See also TView::GetViewName

GetWindow

TWindow* GetWindow();

Overrides *TView's* virtual function and returns the list view object as a *TWindow*.

See also TView::GetWindow

SetDocTitle

bool SetDocTitle(const char far* docname, int index);

Overrides *TView*'s virtual function and stores the document title. This name is forwarded up the parent chain until a *TFrameWindow* object accepts the data and displays it in its caption.

See also TView::SetDocTitle

StaticName

static const char far* StaticName();

Overrides *TView*'s function and returns a constant string, "ListView." This information is displayed in the user interface selection box.

See also TView::GetViewName

Protected data members

MaxWidth

int MaxWidth;

Holds the maximum horizontal extent (the number of pixels by which the view can be scrolled horizontally).

Origin
long Origin;
Holds the file position at the beginning of the display.

Protected member functions

CmEditAdd
void CmEditAdd();
Automatically responds to CM_LISTADD message by getting the length of the input string and calling *InsertString* to insert the text string into the list view. Sets the data member *DirtyFlag* to true.

CmEditClear
void CmEditClear();
Automatically responds to a menu selection with a menu ID of CM_EDITCLEAR by clearing the items in the list box using functions in *TListBox*.

CmEditCopy
void CmEditCopy();
Automatically responds to a menu selection with a menu ID of CM_EDITCOPY and copies the selected text to the Clipboard.

CmEditCut
void CmEditCut();
Automatically responds to a menu selection with a menu ID of CM_EDITCUT by calling *CmEditCopy* and *CmEditDelete* to delete a text string from the list view. Sets the data member *DirtyFlag* to true.

CmEditDelete
void CmEditDelete();
Automatically responds to a menu selection with a menu ID of CM_EDITDELETE and deletes the selected text.

CmEditItem
void CmEditItem();
Automatically responds to a CM_LISTEDIT message by getting the input text and inserting into the list view. Sets the *DirtyFlag* to nonzero to indicate that the view has been changed and not saved.

CmEditPaste
void CmEditPaste();
Automatically responds to a menu selection with a menu ID of CM_EDITPASTE by inserting text into the list box using functions in *TListBox*.

CmEditUndo
void CmEditUndo();
Automatically responds to a menu selection with a menu ID of CM_EDITUNDO by calling *TListBox::Undo*.

CmSelChange

void CmSelChange();

Automatically responds to a LBN_SELCHANGE message (which indicates that the contents of the list view have changed) by calling *DefaultProcessing*.

EvGetDlgCode

uint EvGetDlgCode(MSG far* msg);

Overrides *TWindow's* response to a WM_GETDLGCODE message (an input procedure associated with a control that isn't a check box) by calling *DefaultProcessing*. The *msg* parameter indicates the kind of message, for example a control or a command message, sent to the dialog box manager.

EvGetDlgCode returns a code that indicates how the list box control message is to be treated.

See also TButton::EvGetDlgCode, TWindow::DefaultProcessing, DLGC_xxxx dialog control message constants

LoadData

bool LoadData(int top, int sel);

Reads the view from the stream and closes the file. Returns true if the view was successfully loaded.

Throws an *xmsg* exception and displays the error message "TListView initial read error" if the file can't be read. Returns false if the view can't be loaded.

SetExtent

void SetExtent(LPSTR str);

Sets the maximum horizontal extent for the list view window.

VnCommit

bool VnCommit(bool force);

VnCommit commits changes made in the view to the document. If *force* is nonzero, all data, even if it's unchanged, is saved to the document.

See also TListView::vnRevert, vnxxxx view notification constants

VnDocClosed

bool VnDocClosed(int omode);

VnDocClosed indicates that the document has been closed.

See also vnxxxx view notification constants

VnIsDirty

bool VnIsDirty();

VnIsDirty returns nonzero if changes made to the data in the view have not been saved to the document; otherwise, returns 0.

See also vnxxxx view notification constants

VnIsWindow

bool VnIsWindow(HWND hWnd);

VnIsWindow returns nonzero if the window's handle passed in *hWnd* is the same as that of the view's display window.

See also vnxxxx view notification constants

VnRevert

bool VnRevert(bool clear);

VnRevert indicates if changes made to the view should be erased, and the data from the document should be restored to the view. If *clear* is nonzero, the data is cleared instead of restored to the view.

See also TListView::vnCommit

Response table entries

Entry	Member function
EV_COMMAND(CM_LISTUNDO, CmEditUndo)	CmEditUndo
EV_COMMAND(CM_LISTCUT, CmEditCut)	CmEditcut
EV_COMMAND(CM_LISTCOPY, CmEditCopy)	CmEditCopy
EV_COMMAND(CM_LISTPASTE, CmEditPaste)	CmEditPaste
EV_COMMAND(CM_LISTCLEAR, CmEditClear)	CmEditClear
EV_COMMAND(CM_LISTDELETE, CmEditDelete)	CmEditDelete
EV_COMMAND(CM_LISTADD, CmEditAdd)	CmEditAdd
EV_COMMAND(CM_LISTEDIT, CmEditItem)	CmEditItem
EV_WM_GETDLGCODE	EvGetDlgCode
EV_NOTIFY_AT_CHILD(LBN_DBLCLK, CmEditItem)	CmEditItem
EV_NOTIFY_AT_CHILD(LBN_SELCHANGE, CmSelChange)	CmSelchange
EV_VN_DOCCLOSED	VnDocClosed
EV_VN_ISWINDOW	VnIsWindow
EV_VN_ISDIRTY	VnIsDirty
EV_VN_COMMIT	VnCommit
EV_VN_REVERT	VnRevert

TLocaleString struct

locale.h

Designed to provide support for localized registration parameters, the *TLocaleString* struct defines a localizable substitute for **char*** strings.These strings, which describe both OLE and non-OLE enabled objects to the user, are available in whatever language the user needs. This struct supports ObjectWindows' Doc/View as well as ObjectComponents, OLE-enabled applications.The public member functions, which supply information about the user's language, the native language, and a description of the string marked for localization, simplify the process of translating and comparing strings in a given language.

To localize the string resource, *TLocaleString* uses several user-entered prefixes to determine what kind of string to translate. Each prefix must be followed by a valid

resource identifier (a standard C identifier).The following table lists the the prefixes *TLocaleString* uses to localize strings. Each prefix is followed by a sample entry.

Prefix	Description
@ TXY	The string is a series of characters interpreted as a resource ID and is accessed only from a resource file. It is never used directly.
# 1045	The string is a series of digits interpreted as a resource ID and is accessed from a resource file. It is never used directly.
! MyWindow	The string is translated if it is not in the native language; otherwise, this string is used directly.

See the section on localizing symbol names in the *ObjectWindows Programmer's Guide* for more information about localizing strings.

Public member functions

Compare

int Compare(const char far* str, TLangId lang);

Using the specified language (*lang*), *Compare* compares *TLocaleString* with another string. It uses the standard string compare and the language-specific collation scheme. It returns one of the following values.

Return value	Meaning
0	There is no match between the two strings.
1	This string is greater than the other string.
–1	This string is less than the other string.

GetSystemLangId

static TLangId GetSystemLangId();

Returns the system language ID, which can be the same as the *UserLangId*.

See also TLocaleString::GetUserLangId

GetUserLangId

static TLangId GetUserLangId();

Returns the user language ID. For single user systems, this is the same as *LangSysDefault*. The language ID is a predefined number that represents a base language and dialect.

See also TLocaleString::GetSystemLangId

IsNativeLangId

static int IsNativelangId(TLangId lang);

Returns **true** if *lang* equals the native system language.

operator const char*

operator const char* ();

Returns the current character string in the translation.

operator =

void operator = (const char* str);

Assigns the string (*str*) to this locale string.

Translate

const char* Translate(TLangId lang);

Translates the string to the given language. *Translate* follows this order of preference in order to choose a language for translation:

1. Base language and dialect.
2. Base language and no dialect
3. Base language and another dialect
4. The native language of the resource itself.
5. Returns 0 if unable to translate the string. (This can happen only if an @ or # prefix is used; otherwise, the ! prefix indicates that the string following is the native language itself.)

See also TLangId typedef, LangXxxx_ID constants

TLookupValidator class

validate.h

A streamable class, *TLookupValidator* compares the string typed by a user with a list of acceptable values. *TLookupValidator* is an abstract validator type from which you can derive useful lookup validators. You will never create an instance of *TLookupValidator*. When you create a lookup validator type, you need to specify a list of valid items and override the *Lookup* method to return true only if the user input matches an item in that list. One example of a working descendant of *TLookupValidator* is *TStringLookupValidator*.

Public constructor

Constructor

TLookupValidator();

Constructs a *TLookupValidator* object.

Public member functions

IsValid

bool IsValid(const char far* str);

IsValid overrides *TValidator's* virtual function and calls *Lookup* to find the string *str* in the list of valid input items. *IsValid* returns **true** if *Lookup* returns **true**, meaning *Lookup* found *str* in its list; otherwise, it returns **false**.

Lookup

virtual bool Lookup(const char far* str);

Searches for the string *str* in the list of valid entries and returns **true** if it finds *str*; otherwise, returns **false**. *TLookupValidator's Lookup* is an abstract method that always returns **false**. Descendant lookup validator types must override *Lookup* to perform a search based on the actual list of acceptable items.

TMDIChild class

mdichild.h

TMDIChild defines the basic behavior of all MDI child windows. Child windows can be created inside the client area of a parent window. Because child windows exist within, and are restricted to the parent window's borders, the parent window defined before the child is defined. For example, a dialog box is a window that contains child windows, often referred to as dialog box controls.

To be used as MDI children, classes must be derived from *TMDIChild*. MDI children can inherit keyboard navigation, focus handling, and icon support from *TFrameWindow*. *TMDIChild* is a streamable class.

Public constructors and destructor

Constructors

Form 1 `TMDIChild(TMDIClient& parent, const char far* title = 0, TWindow* clientWnd = 0, bool shrinkToClient = false, TModule* module = 0);`

Creates an MDI child window of the MDI client window specified by *parent*, using the specified *title*, client window (*clientWnd*) and instance (*inst*). Invokes the *TFrameWindow* base class constructor, supplying *parent*, *title*, *clientWnd*, *inst*, and indicating that the child window is not to be resized to fit. Invokes the *TWindow* base class constructor, specifying *parent*, *title*, and *inst*. The window attributes are then adjusted to include WS_VISIBLE, WS_CHILD, WS_CLIPSIBLINGS, WS_CLIPCHILDREN, WS_SYSMENU, WS_CAPTION, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX. The dimensions of the window are set to the system default values.

Form 2 `TMDIChild(HWND hWnd, TModule* module = 0);`

Creates an MDI child window object from a preexisting window, specified by *hWnd*. The base class *TFrameWindow* constructor is invoked, specifying this *hWnd*, as well as the specified *inst*. The base class *TWindow* constructor is invoked, supplying the *hWnd* and *inst* parameters.

Destructor

`~TMDIChild();`

Destructs the MDI child window object.

Public member functions

Destroy

`void Destroy(retVal = 0);`

Destroys the interface element associated with the *TMDIChild*. Calls *EnableAutoCreate* for each window in the child list so that the children are also re-created when the parent window is re-created.

See also TWindow::EnableAutoCreate

EnableWindow

bool EnableWindow(bool enable);

Overrides *TWindow's* virtual function. *Enables* a child window.

PreProcessMsg

bool PreProcessMsg (MSG& msg);

Performs preprocessing of window messages for the MDI child window. If keyboard handling is enabled the parent client window's *TMDIClient_PreProcessMsg* member function is called to preprocess messages. In this case, the return value is true. Otherwise, *TFrameWindow::PreProcessMsg* is called and its return value becomes the return value of this member function.

See also TMDIClient::PreProcessMsg, TFrameWindow::PreprocessMsg

ShowWindow

bool ShowWindow(int cmdShow);

Overrides *TWindow's* virtual function. Displays a child window according to the value of *cmdShow*.

Protected member functions

DefWindowProc

LRESULT DefWindowProc (uint msg, WPARAM wParam, LPARAM lParam);

Overrides *TWindow::DefWindowProc* to provide default processing for any incoming message the MDI child window does not process. In addition, DefWindowProc handles the following messages: WM_CHILDACTIVATE, WM_GETMINMAXINFO, WM_MENUCHAR, WM_MOVE, WM_SETFOCUS, WM_SIZE, and WM_SYSCOMMAND.

See also TWindow::DefWindowProc

EvMDIActivate

void EvMDIActivate(HWND hWndActivated, HWND hWndDeactivated);

Instructs a client window to activate or deactivate an MDI child window and then sends a message to the child window being activated and the child window being deactivated.

EvNCActivate

void EvNCActivate(bool activate);

Responds to a request to change a title bar or icon.

PerformCreate

void PerformCreate(int menuOrId);

Creates the interface element associated with the MDI child window. The supplied *menuOrId* parameter is ignored because MDI child windows cannot have menus.

Response table entries

Response table entry	Member function
EV_WM_MDIACTIVATE	EvMDIActivate
EV_WM_NCACTIVATE	EvNCActivate

TMDIClient class

mdi.h

Multiple Document Interface (MDI) client windows (represented by a *TMDIClient* object) manage the MDI child windows of a *TMDIFrame* parent. *TMDIClient* is a streamable class.

Public constructor and destructor

Constructor

TMDIClient(TModule* module = 0);

Creates an MDI client window object by invoking the base class *TWindow* constructor, passing it a null parent window, a null title, and the specified library ID. Sets the default client window identifier (IDW_MDICLIENT) and sets the style to include MDIS_ALLCHILDSTYLES, WS_GROUP, WS_TABSTOP, WS_CLIPCHILDREN, WS_VSCROLL, and WS_HSCROLL. Initializes the *ClientAttr* data member, setting its *idFirstChild* member to IDW_FIRSTMDICHILD.

Destructor

~TMDIClient();

Frees the *ClientAttr* structure.

See also TWindow::TWindow, TWindow::~TWindow

Public data member

ClientAttr

LPCLIENTCREATESTRUCT ClientAttr;

ClientAttr holds a pointer to a structure of the MDI client window's attributes.

Public member functions

ArrangeIcons

virtual void ArrangeIcons();

Arranges the MDI child window icons at the bottom of the MDI client window.

CascadeChildren

virtual void CascadeChildren();

Sizes and arranges all of the non-iconized MDI child windows within the MDI client window. The children are overlapped, although each title bar is visible.

CloseChildren

virtual bool CloseChildren();

First calls *CanClose* on each of the MDI child windows owned by this MDI client. Returns true if all MDI children are closed; otherwise returns false.

See also TWindow::CanClose

Create

bool Create();

Creates the interface element associated with the MDI client window. Calls *TWindow::Create* after first setting the child window menu in *ClientAttr* to the parent frame window's child menu.

See also TWindow_Create, TFrameWindow_GetMenuDescr

CreateChild

virtual TWindow* CreateChild();

Overrides member function defined by *TWindow*. Constructs and creates a new MDI child window by calling *InitChild* and *Create*. Returns a pointer to the new MDI child window.

See also TMDIClient::InitChild, TModule::MakeWindow, TWindow::Create

GetActiveMDIChild

TMDIChild *GetActiveMDIChild();

GetActiveMDIChild points to the *TMDIClient's* active MDI child window. *GetActiveMDIChild* is set by the child in its *EvMDIActivate* message response member function. *TMDIClient's* constructors initialize *GetActiveChild*.

InitChild

virtual TMDIChild *InitChild();

Constructs an instance of *TWindow* as an MDI child window and returns a pointer to it. Children must be created with MDI client as the parent window. Redefine this member function in your derived MDI window class to construct an instance of a derived MDI child class. For example,

```
PTWindowsObject TMyMDIClient::InitChild()
{
   return new TMyMDIChild(this, "");
}
```

See also TMDIClient::CreateChild

PreProcessMsg

bool PreProcessMsg(MSG &msg);

If the specified *msg* is one of WM_KEYDOWN or WM_SYSKEYDOWN, then the keyboard accelerators are translated for the MDI client.

See also TWindow::PreProcessMsg

TileChildren

virtual void TileChildren(int tile = MDITILE_VERTICAL);

Sizes and arranges all of the non-iconized MDI child windows within the MDI client window. The children fill up the entire client area without overlapping.

See also TMDIClient::TileChildren

Protected member functions

CmArrangeIcons

void CmArrangeIcons();

Calls *ArrangeIcons* in response to a menu selection with an ID of CM_ARRANGEICONS.

See also TMDIClient::ArrangeIcons

CmCascadeChildren

void CmCascadeChildren();

Calls *CascadeChildren* in response to a menu selection with an ID of CM_CASCADECHILDREN.

See also TMDIClient::CascadeChildren

CmChildActionEnable

void CmChildActionEnable(TCommandEnabler& commandEnabler);

If there are MDI child windows, *CmChildActionEnable* enables any one of the child window action menu items.

CmCloseChildren

void CmCloseChildren();

Calls *CloseChildren* in response to a menu selection with an ID of CM_CLOSECHILDREN.

See also TMDIClient::CloseChildren

CmCreateChild

void CmCreateChild();

Calls *CreateChild* to produce a new child window in response to a menu selection with a menu ID of CM_CREATECHILD.

See also TMDIClient::CreateChild

CmTileChildren

void CmTileChildren();

Calls *TileChildren* in response to a menu selection with an ID of CM_TILECHILDREN.

See also TMDIClient::TileChildren

CmTileChildrenHoriz

void CmTileChildrenHoriz();

Calls *TileChildren* in response to a menu selection with an ID of CM_TILECHILDREN and passes MDI child tile flag as MDITILE_HORIZONTAL.

EvMDICreate

LRESULT EvMDICreate(MDICREATESTRUCT far& createStruct);

Intercepts the WM_MDICREATE message sent when MDI child windows are created, and, if the client's style includes MDIS_ALLCHILDSTYLES, and the child window's specified style is 0, then changes the child window style attributes to WS_VISIBLE, WS_CHILD, WS_CLIPSIBLINGS, WS_CLIPCHILDREN, WS_SYSMENU, WS_CAPTION, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX.

See also TWindow::DefaultProcessing, MDICREATE struct

EvMDIDestroy

void EvMDIDestroy(HWND hWnd);

Intercepts the WM_MDIDESTROY message.

GetClassName

char far *GetClassName();

Returns *TMDIClient's* registration class name, "MDICLIENT."

Response table entries

Response table entry	Member function
EV_COMMAND (CM_ARRANGEICONS, CmArrangeIcons)	CmArrangeIcons
EV_COMMAND (CM_CASCADECHILDREN, CmCascadeChildren)	CmCascadeChildren
EV_COMMAND (CM_CLOSECHILDREN, CmCloseChildren)	CmCloseChildren
EV_COMMAND (CM_CREATECHILD, CmCreateChild)	CmCreateChild
EV_COMMAND (CM_TILECHILDREN, CmTileChildren)	CmTileChildren
EV_COMMAND(CM_TILECHILDRENHORIZ,CmTileChildrenHoriz)	CmTileChildrenHoriz
EV_COMMAND_ENABLE(CM_TILECHILDREN, CmChildActionEnable)	CmChildActionEnable
EV_COMMAND_ENABLE(CM_CASCADECHILDREN, CmChildActionEnable)	CmChildActionEnable
EV_COMMAND_ENABLE(CM_ARRANGEICONS, CmChildActionEnable)	CmChildActionEnable
EV_COMMAND_ENABLE(CM_CLOSECHILDREN, CmChildActionEnable)	CmChildActionEnalbe
EV_WM_MDICREATE	EvMDICreate
EV_WM_MDIDESTROY	EvMDIDestroy

TMDIFrame class

mdi.h

Multiple Document Interface (MDI) frame windows, represented by *TMDIFrame*, are overlapped windows that serve as main windows of MDI-compliant applications. *TMDIFrame* objects automatically handle the creation and initialization of an MDI client "window" (represented by a *TMDIClient* object) required by Windows. *TMDIFrame* sets window style WS_CLIPCHILDREN by default so that minimal flicker occurs when the MDI frame erases its background and the backgrounds of its children. *TMDIFrame* is a streamable class.

Because *TMDIFrame* is derived from *TFrameWindow*, it inherits keyboard navigation. As a result, all children of the MDI frame acquire keyboard navigation. However, it's best to enable keyboard navigation only for those children who require it.

To create an OLE-enabled MDI frame window, use *TOleMDIFrame*, which inherits functionality from both *TMDIFrame* and and *TOleFrame*.

See also
TOLEMDIFrame

Public constructors

Constructors

Form 1 `TMDIFrame(const char far *title, TResId menuResId, TMDIClient &clientWnd = *new TMDIClient, TModule* module = 0);`

Constructs an MDI frame window object using the caption (*title*) and resource ID (*menuResId*). If no client window is specified (*clientWnd*), then an instance of *TMDIClient* is created automatically and used as the client window of the frame. The supplied library ID (*module*) is passed to the *TFrameWindow* constructor along with a null parent window pointer, caption, client window, and a flag indicating that the client window is not to be resized to fit. The *TWindow* constructor is also invoked; it passes the supplied caption and library ID, as well as a null parent window pointer. Then the child menu position is initialized to be the leftmost menu item, and the supplied menu resource ID is used in a call to *AssignMenu*.

Form 2 `TMDIFrame(HWND hWindow, HWND clientHWnd, TModule* module = 0);`

Constructs an MDI frame window using an already created non-ObjectWindows window. Invokes the *TFrameWindow TWindow* constructors passing in the window handle (*hWindow*) library ID (*module*). Initializes the child menu position to the leftmost menu item and constructs a *TMDIClient* object that corresponds to the supplied *clientHWnd*.

See also TFrameWindow::AssignMenu, TFrameWindow::TFrameWindow, TMDIClient::TMDIClient, TWindow::TWindow

Public member functions

FindChildMenu

`static HMENU FindChildMenu(HMENU newMenu);`

FindChildMenu searches, from right to left, the pop-up child menus contained in the *newMenu* menu resource for a child menu containing command items with one of the following identifiers: CM_CASCADECHILDREN, CM_TITLECHILDREN, or CM_ARRANGEICONS. The return value of *FindChildMenu* is the *HMENU* of the first child menu containing one of these identifiers. If one of these identifiers is not found, *FindChildMenu* returns 0.

FindChildMenu is used to locate the menu to which the MDI child window list will be appended. This call to *FindChildMenu* is made from within the *TMDIClient* class.

GetClientWindow

TMDIClient *GetClientWindow();

Returns a pointer to the MDI client window.

See also TFrameWindow::GetClientWindow

GetCommandTarget

virtual HWND GetCommandTarget();

Locates and returns the child window that is the target of the command and command enable messages. If the current application does not have focus or if the focus is within a toolbar in the application, *GetCommandTarget* returns the most recently active child window. If an alternative form of command processing is desired, a user's main window class can override this function.

PerformCreate

void PerformCreate(int menuOrId);

Overrides *TWindow*'s virtual function. Creates the interface element associated with the MDI frame window.

SetMenu

bool SetMenu (HMENU);

Looks for the MDI submenu in the new menu bar. Searches for the MDI child menu in the new menu bar and updates the child menu position with the specified menu *index*. Then sends the client window an WM_MDISETMENU message to set the new menu and invokes *TWindow::DrawMenuBar* to redraw the menu. Returns false if the MDI client indicates that there was no previous menu; otherwise, returns true.

See also TWindow::DrawMenuBar

Protected member functions

DefWindowProc

LRESULT DefWindowProc (uint message, WPARAM wParam, LPARAM lParam);

Overrides *TWindow::DefWindowProc* and provides default processing for any incoming message the MDI frame window does not process.

See also TWindow::DefWindowProc

Response table entries

The *TMDIFrame* response table has no entries.

TMeasurementUnits enum

layoutco.h

enum TMeasurementUnits{lmPixels,lmLayoutUnits};

Used by the *TLayoutConstraint* struct, *TMeasurementUnits* enumerates the measurement units (*lmPixels* or *lmLayoutUnits*) that control the dimension of the window. These can be either pixels or layout units that are obtained by dividing the font height into eight vertical and eight horizontal segments.

See also TLayoutConstraint struct

TMemoryDC class

dc.h

A DC class derived from *TDC*, *TMemoryDC* provides access to a memory DC.

Public constructors

Constructors

Form 1 TMemoryDC();
Default constructor for a memory DC object.

Form 2 TMemoryDC(const TDC& DC);
Creates a memory DC object compatible with the given *DC* argument.

Form 3 TMemoryDC(HDC handle, TAutoDelete AutoDelete);
Creates a memory DC object from an existing *DC* handle.

See also TDC::TDC

Public member functions

RestoreBitmap

void RestoreBitmap();
Restores the originally selected bitmap object for this DC.

See also TDC::RestoreObjects

RestoreObjects

void RestoreObjects();
Restores the originally selected brush, pen, font, palette, and bitmap objects for this DC.

See also TDC::RestoreObjects, TMemoryDC::RestoreBitmap

SelectObject

void SelectObject(const TBrush& brush);
void SelectObject(const TPen& pen);
void SelectObject(const TFont& font);
void SelectObject(const TPalette& palette, bool forceBackground=false);
void SelectObject(const TBitmap& bitmap);
Selects the given GDI object into this DC.

See also TDC::SelectObject, TMemoryDC::RestoreBitmap, TMemoryDC::RestoreObjects

Protected data member

OrgBitmap

HBITMAP OrgBitmap;

The original bitmap selected into this DC.

See also TMemoryDC::SelectObject, TMemoryDC::RestoreBitmap

TMenu class

menu.h

The *TMenu* class encapsulates window menus. You can use *TMenu* member functions to construct, modify, query, and create menu objects. You can also use *TMenu* to add bitmaps to your menu or to specify if a menu item is checked or unchecked. *TMenu* includes two versions of a helper function, *DeepCopy*, designed to make copies of menus and insert them at a specified position on the menu bar. See the *ObjectWindows Programmer's Guide* for information about how to create menu objects.

Public constructors and destructor

Constructors

Form 1 `TMenu(TAutoDelete autoDelete = AutoDelete);`
Creates an empty menu and sets *autoDelete*, by default, so that the menu is automatically deleted when the object is destroyed.

Form 2 `TMenu(const TMenu& original, TAutoDelete autoDelete = NoAutoDelete);`
Creates a complete copy of an existing menu and sets *autoDelete*, by default, so that the menu is not automatically deleted when the object is destroyed.

Form 3 `TMenu(HWND wnd, TAutoDelete autoDelete = NoAutoDelete);`
Creates a menu object representing the window's current menu and sets *autoDelete*, by default, so that the menu is not automatically deleted when the object is destroyed.

Form 4 `TMenu(HMENU handle, TAutoDelete autoDelete = NoAutoDelete);`
Creates a menu object from an already loaded menu and sets *autoDelete*, by default, so the menu is not automatically deleted when the object is destroyed.

Form 5 `TMenu(const void far* menuTemplate);`
Creates a menu object from a menu template in memory. This constructor is not available under Presentation Manager.

Form 6 `TMenu(HINSTANCE instance, TResId resId);`
Creates a menu object from a specified resource ID.

Destructor

`virtual ~TMenu();`
Destroys the menu.

See also TResId class

Public member functions

AppendMenu

Form 1 `bool AppendMenu(uint flags, uint newItem, const TBitmap& newBmp);`

Adds a bitmap menu item at the end of the menu. See *TMenu::GetMenuState* for a description of the flag values that specify the attributes of the menu, for example, menu item is checked, menu item is disabled, and so on.

Form 2 bool AppendMenu(uint flags, uint newItem = –1, const char far* newItem = 0);
Adds a text menu item to the end of the menu. See *TMenu::GetMenuState* for a description of the flag values that specify the attributes of the menu, for example, menu item is checked, menu item is a a bitmap, and so on.

See also TBitmap class, TMenu::GetMenuState

CheckMenuItem
bool CheckMenuItem(uint item, uint check);
Checks or unchecks the menu item. By combining flags with the bitwise OR operator (|) *check* specifies both the position of *item* (MF_BYCOMMAND, MF_BYPOSITION) and whether *item* is to be checked (MF_CHECKED) or unchecked (MF_UNCHECKED).

CheckValid
void CheckValid(uint resId = IDS_MENUFAILURE);
Throws a *TXMenu* exception if the menu object is invalid.

See also TMenu::TXMenu

DeleteMenu
bool DeleteMenu(uint item, uint flags);
Removes the menu item (*item*) from the menu or deletes the menu item if it's a pop-up menu. *flags* is used to identify the position of the menu item by its relative position in the menu (MF_BYPOSITION) or by referencing the handle to the top-level menu (MF_BYCOMMAND).

See also TMenu::RemoveMenu

DrawItem
virtual void DrawItem(DRAWITEMSTRUCT far& drawItem);
DrawItem responds to a message forwarded to a drawable control by *TWindow* when the control needs to be drawn.

See also DRAWITEMSTRUCT struct

EnableMenuItem
bool EnableMenuItem(uint item, uint enable);
Enables, disables, or grays the menu item specified in the *item* parameter. If a menu item is enabled (the default state), it can be selected and used as usual. If a menu item is grayed, it appears in grayed text and cannot be selected by the user. If a menu item is disabled, it is not displayed. Returns **true** if successful.

GetHandle
virtual HMENU GetHandle();
Returns the handle to the menu.

See also TMenu::IsOK

GetMenuCheckMarkDimensions
static bool GetMenuCheckMarkDimensions(TSize& size);

Gets the size of the bitmap used to display the default checkmark on checked menu items.

See also TMenu::SetMenuItemBitmaps, TSize class

GetMenuItemCount

uint GetMenuItemCount() const;

Returns the number of items in a top-level or pop-up menu.

GetMenuItemID

uint GetMenuItemID(int posItem) const;

Returns the ID of the menu item at the position specified by *posItem*. If this is a pop-up menu, returns the ID of the menu's first item minus one.

GetMenuState

uint GetMenuState(uint item, uint flags) const;

Returns the menu flags for the menu item specified by *item*. *flags* specifies how the *item* is interpreted, and is one of the following values:

Flag	Description
MF_BYCOMMAND	Interpret *item* as a menu command ID. Default it neither MF_BYCOMMAND nor MF_BYPOSITION is specified.
MF_BYPOSITION	Interpret *item* as the zero-base relative postion of the menu item within the menu.

If *item* is found, and is a pop-up menu, the low-order byte of the return value contains the flags associated with *item*, and the high-order byte contains the number of items in the pop-up menu. If *item* is not a pop-up menu, the return value specifies a combination of these flags:

Flag	Description
MF_BITMAP	Menu item is a a bitmap.
MF_CHECKED	Menu item is checked (pop-up menus only).
MF_DISABLED	Menu item is disabled.
MF_ENABLED	Menu item is enabled. Note: this constant's value is 0.
MF_GRAYED	Menu item is disabled and grayed.
MF_MENUBARBREAK	Same as MF_MENUBREAK except pop-up menu columns are separated by a vertical dividing line.
MF_MENUBREAK	Static menu items are placed on a new line, pop-up menu items are placed in a new column, without separating columns.
MF_SEPARATOR	A horizontal dividing line is drawn, which cannot be enabled, checked, grayed, or highlighted. Both *item* and *flags* are ignored.
MF_UNCHECKED	Menu item check mark is removed (default). Note: this constant value is 0. Returns –1 if item doesn't exist.

GetMenuString

uint GetMenuString(uint item, char* str, int count, uint flags) const;

Returns the label (*str*) of the menu item (*item*).

GetSubMenu

HMENU GetSubMenu(int posItem) const;

Returns the handle of the menu specified by *posItem*.

InsertMenu

Form 1 bool InsertMenu(uint item, uint flags, uint newItem, const TBitmap& newBmp);

Adds a bitmap menu item after the menu item specified in *item*. The *flags* parameter contains either the MF_BYCOMMAND or MF_BYPOSITION values that indicate how to interpret the item parameter. If MF_BYCOMMAND, item is a command ID; if MF_BYPOSITION, item holds a relative position within the menu.

Form 2 bool InsertMenu(uint item, uint flags, uint newItem = –1, const char far* newItem = 0);

Inserts a new text menu item or pop-up menu into the menu after the menu item specified in *item*. The *flags*parameter contains either the MF_BYCOMMAND or MF_BYPOSITION values that indicate how to interpret the item parameter. If MF_BYCOMMAND, item is a command ID; if MF_BYPOSITION, item holds a relative position within the menu.

See also TMenu::GetMenuState

IsOK

bool IsOK() const;

Returns **true** if the menu has a valid handle.

See also TMenu::GetHandle

MeasureItem

virtual void MeasureItem(MEASUREITEMSTRUCT far& measureItem);

measureItem is used by owner-drawn controls to store the dimensions of the specified item.

See also MEASUREITEMSTRUCT struct

ModifyMenu

Form 1 bool ModifyMenu(uint item, uint flags, uint newItem, const TBitmap& newBmp);

Changes an existing menu item into a bitmap. The *flags* parameter contains either the MF_BYCOMMAND or MF_BYPOSITION values that indicate how to interpret the item parameter. If MF_BYCOMMAND, item is a command ID; if MF_BYPOSITION, item holds a relative position within the menu.

Form 2 bool ModifyMenu(uint item, uint flags, uint newItem = –1, const char far* newItem = 0);

Changes an existing menu item from the item specified in *item* to *newItem*. The *flags* parameter contains either the MF_BYCOMMAND or MF_BYPOSITION values that indicate how to interpret the item parameter. If MF_BYCOMMAND, item is a command ID; if MF_BYPOSITION, item holds a relative position within the menu.

See also TMenu::GetMenuState

operator HMENU()

operator HMENU();

Returns the menu's handle.

See also TMenu::operator uint

operator =

```
TMenu& operator =(const TMenu& original);
```

Returns the *TMenu* object.

See also TMenu::operator uint

operator uint()

```
operator uint();
```

Returns the menu's handle. This function provides compatibility with functions that require a **uint** menu parameter.

See also TMenu::operator HMenu

RemoveMenu

```
bool RemoveMenu(uint item, uint flags);
```

Removes the menu item from the menu but does not delete it if it is a submenu.

See also TMenu::DeleteMenu

SetMenuItemBitmaps

```
bool SetMenuItemBitmaps(uint item, uint flags, const TBitmap* bmpUnchecked=0,
  const TBitmap* bmpChecked=0);
```

Specifies the bitmap to be displayed when the menu item is checked and unchecked. *item* indicates the menu item to be associated with the bitmap. *flags* indicates how the *size* parameter is interpreted (whether by MF_BYPOSITION or by MF_BYCOMMAND). *GetMenuCheckMarkDimensions* gets the size of the bitmap.

See also TMenu::GetMenuCheckmarkDimensions, TBitmap

Protected data members

Handle

```
HMENU Handle;
```

Holds the handle to the menu.

ShouldDelete

```
bool ShouldDelete;
```

ShouldDelete is set to **true** if the destructor needs to delete the handle to the menu.

See also TMenu::DeleteMenu, TMenu::RemoveMenu

Protected member functions

DeepCopy

Form 1

```
static void DeepCopy(TMenu& dest, const TMenu& source, int offset = 0, int count = –1);
```

Makes a deep copy (that is, an actual copy of the menu, not just a copy of pointers or handles to a menu) of the menu. This form of *DeepCopy* copies *count* number of pop-up menus or menu items from *src* beginning at *offset* and appends the items or menus to the destination menu. If *count* is passed as –1, all source menu items are copied.

Form 2

```
static void DeepCopy(TMenu& dst, int dstOff, const TMenu& src, int srcOff = 0, int count = –1);
```

Makes a deep copy (that is, an actual copy of the menu, not just a copy of pointers or handles to a menu) of the menu. This form of *DeepCopy* copies *count* number of pop-up menus or menu items from *source* beginning at *offset* and inserts the items or menus at the *dstOffset* position specified in the destination menu (*dest*). If *count* is passed as –1, all source menu items are copied.

TMenuDescr class

menu.h

Derived from *TMenu*, *TMenuDescr* describes your menu bar and its functional groups. *TMenuDescr* provides an easy way for you to group menus on your menu bar and to add new groups to an existing menu bar. It uses a resource ID to identify the menu resource and an array of count values to indicate the number of pop-up menus in each group on the menu bar.

The *TGroup* enum enumerates the six basic functional groups on a menu bar: File, Edit, Container, Object, Window, and Help. *TMenuDescr*'s constructors simply initialize the members based on the arguments passed: *TFrameWindow*'s *MergeMenu* function actually performs the real work of merging the menu groups.

One method you can use to create a menu involves invoking the *TMenuDescr* constructor and passing the number of group counts for each menu selection.

For example, if your original menu looked like this:

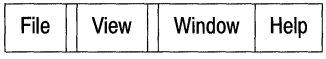
you might use the following group counts:

Group	Count	Menu
FileGroup	1	File
EditGroup	2	Edit Search
ContainerGroup	1	View
ObjectGroup	3	Page Paragraph Word
WindowGroup	1	Window
HelpGroup	1	Help

You would then invoke the constructor this way:

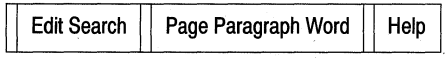
```
TMenuDescr(IDM_MYMENU, 1,2, 1, 3, 1, 1)
```

You can build the previous menu by merging two menus. Set your application's frame menu bar this way:




```
TMenuDescr(IDM_FRAME, 1, 0, 1, 0, 1, 1)
```

and the word-processor child menu bar this way:

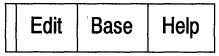



```
TMenuDescr(IDM_WPROC, 0, 2, 0, 3, 0, 1)
```

If no child is active, only the frame menu will be active. When the word processor's child window becomes active, the child menu bar is merged with the frame menu. Every group that is 0 in the child menu bar leaves the parent's group intact. The previous example interleaves every group except for the last group, the Help group, in which the child replaces the frame menu.

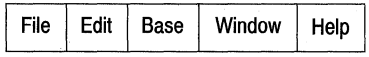
By convention, the even groups (File, Container, Window) usually belong to the outside frame or container, and the odd groups (Edit, Object, Help) belong to the child or contained group.

If a –1 is used in a group count, the merger eliminates the parent's group without replacing it. For example, another child menu bar, such as a calculator, could be added to your application in this way.




```
TMenuDescr(IDM_WCALC, 0, 1, -1, 1, 0, 1)
```

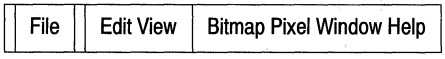
This produces a merged menu (with the View menu selection eliminated as a result of the –1) that looks like this:



You could add a paint window in this way:


```
TMenuDescr(IDM_WPAINT, 0, 1, 0, 2, 0, 1)
```

This produces the following merged menu:



The simplest way to add groups to a menu bar involves defining the menu groups and adding separators in a resource file. Insert the term MENUITEM SEPARATOR between each menu group and an additional separator if one of the menu groups is not present. For example, the resource file for Step14 of the ObjectWindows tutorial defines the following menu groups and separators:

```
IDM_MDICMNDS MENU
{
// Display a grayed File menu
 MENUITEM "File",   0,GRAYED  ;placeholder for File menu from DocManager
 MENUITEM SEPARATOR
 MENUITEM "Edit",   CM_NOEDIT ;placeholder for Edit menu from View
 MENUITEM SEPARATOR
```

```
 MENUITEM SEPARATOR
 MENUITEM SEPARATOR
 POPUP "&Window"
{
// Options within the Window menu group
  MENUITEM  "&Cascade",       CM_CASCADECHILDREN
  MENUITEM  "&Tile",          CM_TILECHILDREN
  MENUITEM  "Arrange &Icons", CM_ARRANGEICONS
  MENUITEM  "C&lose All",     CM_CLOSECHILDREN
  MENUITEM  "Add &View",      CM_VIEWCREATE
 }
 MENUITEM SEPARATOR
 POPUP "&Help"
 {
  MENUITEM "&About",   CM_ABOUT
 }
}
```

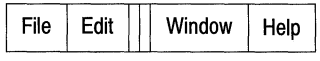
(You can see the separators by loading Step14.rc into Resource Workshop and disabling the View as Popup Option in the View menu.

This resource file defines an Edit group, a File group, a Window group, and a Help group, but no entries for Container or Object groups.

Then, step14.cpp uses these commands from the resource file to set the main window and its menu, passing IDM_MDICMNDS as the parmeter to *SetMenuDescr* function, thus

```
SetMainWindow(frame);
GetMainWindow()->SetMenuDescr(TMenuDescr(IDM_MDICMNDS));
```

and produces the following menu groups:



TMenuDescr's functions let you perform OLE2-like menu merging. That is, you can merge menus from a container's document (the MDI frame window) with those of an embedded object (the MDI child window). When the embedded object is activated in-place by double-clicking the mouse, the menu of the child window merges with that of the frame window.

Public constructors and destructor

Constructors

Form 1 **TMenuDescr();**

Default constructor for a *TMenuDescr* object. No menu resources or groups are specified. Constructs an empty menu bar.

Form 2 **TMenuDescr(TResId id, int fg, int eg, int cg, int og, int wg, int hg, TModule* module = ::Module);**

Constructs a menu descriptor from the resource indicated by the *id.* and *module* parameters. Places the pop-up menus in groups according the values of the *fg, eg, cg, of, wg,* and *hg* parameters. The *fg, eg, cg, of, wg,* and *hg* parameters represent the functional

groups identified by the *TGroup* enum. Calls the function *ExtractGroups* to extract the group counts based on the separator items in the menu bar.

Form 3 `TMenuDescr(const TMenuDescr& original);`
Copies the menu descriptor object specified in the *original* parameter.

Form 4 `TMenuDescr(TResId id, TModule* module = ::Module);`
Creates a menu descriptor from the menu resource specified in the *id* parameter. Calls the function *ExtractGroups* to extract the group counts based on the separator items in the menu bar.

Form 5 `TMenuDescr(HMENU hMenu, int fg, int eg, int cg, int og, int wg, int hg, TModule* module = ::Module);`
Constructs a menu descriptor from the menu handle indicated in the *hMenu* parameter. The menu descriptor can have zero or more pop-up menus in more than one functional group. The *fg, eg, cg, of, wg,* and *hg* parameters represent the functional groups identified by the *TGroup* enum. Calls the function *ExtractGroups* to extract the group counts based on the separator items in the menu bar or uses the group count parameters specified if there are no separators in the menubar.

Destructor

`~TMenuDescr();`
Destroys the *TMenuDescr* object.

Type definitions

TGroup enum

`enum TGroup{FileGroup, EditGroup, ContainerGroup, ObjectGroup, WindowGroup, HelpGroup, NumGroups};`
Used by *TMenuDescr*, the *TGroup* enum describes the following constants that define the index of the entry in the *GroupCount* array.

Constant	Meaning
FileGroup	Index of the File menu group count
EditGroup	Index of the Edit menu group count
ContainerGroup	Index of the Container menu group count
ObjectGroup	Index of the Object group count
WindowGroup	Index of the Window menu group count
HelpGroup	Index of the Help menu group count
NumGroups	Total number of groups

See also TMenuDescr::GroupCount

Public member functions

GetHandle

`HMENU GetHandle() const;`
Gets the handle to the menu, possibly causing any deferred menu acquisition to occur.

GetModule

TModule* GetModule() const;

Returns a pointer to the module object.

SetModule

void SetModule(TModule* module);

Sets the default module object for this menu descriptor.

GetId

TResId GetId() const;

Gets the menu resource ID used to construct the menu descriptor.

GetGroupCount

int GetGroupCount(int group) const;

Gets the number of menus in a specified group within the menu bar. There are a maximum of six functional groups as defined by the *TGroup* enum. These groups include *FileGroup*, *EditGroup*, *ContainerGroup*, *ObjectGroup*, *WindowGroup*, and *HelpGroup*.

See also TGroup enum

ClearServerGroupCount

void ClearServerGroupCount();;

Clears the odd groups (that is, 1, 3, 5 or Edit Group, Object Group, and Help Group) in the menu bar for a server application.

See also TGroup enum, TOcMenuDescr

ClearContainerGroupCount

void ClearContainerGroupCount();

Clears the even groups (that is, 0, 2, 4 or File Group, Container Group, Window Group) in the menu bar for a container application.

See also TGroup enum, TOcMenuDescr

Merge

Form 1 bool Merge(const TMenuDescr& sourceMenuDescr);

Merges the functional groups of another menu descriptor into this menu descriptor.

Form 2 bool Merge(const TMenuDescr& sourceMenuDescr, TMenu& destMenu);

Merges the functional groups of this menu descriptor and another menu descriptor into an empty menu.

See also TMenuDesc::TGroup enum

Protected data members

Id

TResId Id;

Resource ID for the menu. The resource ID is passed in the constructors to identify the menu resource.

GroupCount

int GroupCount[NumGroups];

An array of values indicating the number of pop-up menus in each group on the menu bar.

See also TGroup enum

Module

TModule* Module

Points to the *TModule* object that owns this *TMenuDescr*.

Protected member functions

ExtractGroups

bool ExtractGroups();

Extracts the group counts from the loaded menu bar by counting the number of menus between separator items. After the group counts are extracted, the separators are removed.

See also TMenu, TOcMenuDescr

TMenuItemEnabler class

framewin.h

Derived from *TCommandEnabler*, *TMenuItemEnabler* is a command enabler for menu items. The functions in this class modify the text, check state, and appearance of a menu item.

Public constructor

Constructor

TMenuItemEnabler(HMENU hMenu, uint id, HWND hWndReceiver, int position);

Constructs a *TMenuItemEnabler* with the specified command ID, for the menu item, message responder (*hWndReceiver*), and position on the menu.

Protected data member

HMenu

HMENU HMenu;

The menu that holds the item being enabled or disabled.

Position

int Position;

The position of the menu item.

Public member functions

Enable
void Enable(bool enable);
Overrides *TCommandEnable::Enable*. Enables or disables the menu options that control the appearance of the corresponding menu item.

GetMenu
HMENU GetMenu();
Returns the menu that holds the item being enabled or disabled.

GetPosition
int GetPosition();
Returns the position of the menu item.

SetText
void SetText(LPCSTR);
Overrides *TCommandEnable::SetText*. Changes the text of the corresponding menu item.

SetCheck
void SetCheck(int state);
Overrides *TCommandEnable::SetCheck*. Checks or unchecks the corresponding menu item. The state parameter reflects the menu item's state, which can be checked, unchecked, or indeterminate.

TMessageBar class

messageb.h

Derived from *TGadgetWindow*, *TMessageBar* implements a message bar with one text gadget as wide as the window and no border. Normally positioned at the bottom of the window, the message bar uses the default gadget window font and draws a highlighted line at the top.

Public constructor

Constructor
TMessageBar(TWindow* parent = 0, TFont* font = new TGadgetWindowFont, TModule* module = 0);
Constructs a *TMessageBar* object with the gadget window font. Sets IDW_STATUSBAR, *HighlightLine* to **true**, and *TTextGadget's* member *WideAsPossible* to **true**, making the text gadget as wide as the window.

See also TGadgetWindowFont::TGadgetWindowFont

Public member functions

SetHintText
virtual void SetHintText(const char* text);

Sets or clears the menu hint text for the message bar. Hint text is displayed over all other gadgets and is used for menu and control bar button help.

SetText
void SetText(const char* text);
Forwards the message in the message bar to the text gadget for formatting.

See also TTextGadget::SetText

Protected data members

HighlightLine
bool HighlightLine;
Is **true** if a highlighted line is drawn.

HintText
char* HintText;
Stores the command hint text, if any, that's currently being displayed.

Protected member functions

GetDesiredSize
void GetDesiredSize(TSize& rect);
Calls *TGadgetWindow's GetDesiredSize* to get the size of the message bar. Then, if a highlighting line is drawn, adjusts the size of the message bar.

See also TGadgetWindow::GetDesiredSize

GetInnerRect
void GetInnerRect(TRect& rect);
GetInnerRect computes the rectangle inside the borders and margins of the message bar.

See also TGadgetWindow::GetInnerRect

PaintGadgets
void PaintGadgets(TDC& dc, bool erase, TRect& rect);
Adjusts the message bar and paints a highlight line. Then, *PaintGadgets* either paints the hint text if any is set or calls *TGadgetWindow::PaintGadgets* to repaint each gadget.

See also TGadgetWindow::PaintGadgets

TMetaFileDC class dc.h

Derived from *TDC*, *TMetaFileDC* provides access to a device context with a metafile selected for drawing.

Public constructor and destructor

Constructor
TMetaFileDC(const char far* filename = 0);
Creates a *TMetaFileDC* object with the data written to the named file if one is provided.

Destructor
~TMetaFileDC();
Destroys this object.

See also TDC::TDC

Public member function

Close
HMETAFILE Close();
Closes this metafile DC object. Sets the *Handle* data member to 0 and returns a pointer to a new *TMetaFilePict* object.

See also TMetaFilePict

TMetaFilePict class

metafile.h

TMetaFilePict is a support class used with *TMetaFileDC* to simplify metafile operations, such as playing into a DC or storing data on the clipboard. *TMetaFilePict* automatically handles the conversion between a metafile and a metafilepict.

Public constructors and destructor

Constructors

Form 1 TMetaFilePict(HMETAFILE handle, TAutoDelete autoDelete);
Creates a *TMetaFilePict* object using the given *handle* argument.

Form 2 TMetaFilePict(const TClipboard& clipboard);
Creates a *TMetaFilePict* object from the contents of the specified clipboard.

Form 3 TMetaFilePict(const char* filename);
Creates a *TMetaFilePict* object for the metafile stored in the named file.

Form 4 TMetaFilePict(uint size, void far* data);
Creates a *TMetaFilePict* object for the memory-based metafile specified by *data*. The *data* buffer must hold a metafile of length *size* bytes.

Form 5 TMetaFilePict(HGLOBAL data);
Creates a *TMetaFilePict* object for the memory-based metafile specified by *data*.The *data* global memory block must hold a metafile.

Form 6 TMetaFilePict(const TMetaFilePict& orig, const char far* fileName = 0);
Copies the metafile, *orig*, to the named file. If *filename* is 0 (the default), the metafile is copied to a memory-based metafile.

Destructor

~TMetaFilePict()

Destroys this object.

See also TClipboard

Public member functions

CalcPlaySize

TSize CalcPlaySize(TDC& dc, const TSize& defSize) const;

Calculates the size of this metafile when played on a given DC.

See also TDC, TSize

GetMetaFileBits

HANDLE GetMetaFileBits();

Returns a handle to a global memory block containing this metafile as a collection of bits. The memory block can be used to determine the size of the metafile or to save the metafile as a file.

GetMetaFileBitsEx

uint32 GetMetaFileBitsEx(uint size, void* data);

32-bit only. Retrieves the contents of the metafile associated with this object and copies them (up to *size* bytes) to the *data* buffer. If *data* is nonzero and the call succeeds, the actual number of bytes copied is returned. If *data* is 0, a successful call returns the number of bytes required by the buffer. A return value of 0 always indicates a failure.

Height

int Height() const;

Retrieves the height of this metafile.

operator <<

TClipboard& operator <<(TClipboard& clipboard, TMetaFilePict& mfp);

Places the *TMetaFilePict* object onto the Clipboard. Returns a reference to the resulting Clipboard, allowing the usual chaining of << operations.

See also TClipboard

MappingMode

unsigned MappingMode()const;

Retrieves the mapping mode of this metafile.

operator HMETAFILE()

operator HMETAFILE() const;

Type-conversion operator.

PlayOnto

bool PlayOnto(TDC& dc, const TSize& defSize) const;

Plays the metafile into a DC.

See also TDC, TSize

SetMappingMode
void SetMappingMode(unsigned mm);
Sets the mapping mode of this metafile.

SetSize
void SetSize(const TSize& size);
Sets the size of this metafile.

Size
TSize Size() const;
Retrieves the size of this metafile.

ToClipboard
void ToClipboard(TClipboard& clipboard, unsigned mapMode = MM_ANISOTROPIC, const TSize& extent=TSize(0,0));
Puts this metafile onto the Clipboard.

See also TClipboard, TSize

Width
int Width() const;
Retrieves the width of this metafile.

Protected data members

Extent
TSize Extent;
Holds the extent or size of the metafile.

Mm
int Mm;
Stores the mapping mode for the metafile.

TModule class

module.h

ObjectWindows dynamic-link libraries (DLLs) construct an instance of *TModule*, which acts as an object-oriented stand-in for the library (DLL) module. *TModule* defines behavior shared by both library and application modules. ObjectWindows applications construct an instance of *TApplication*, derived from *TModule*. *TModule*'s constructors manage loading and freeing of external DLLs, and the member functions provide support for default error handling.

Public constructors and destructor

Constructors
Form 1 TModule(const char far* name, HINSTANCE hInstance, const char far* cmdLine);
Constructs a *TModule* object for an ObjectWindows DLL or program from within *LibMain* or *WinMain*. Calls *InitModule* to initialize *hInstance* and *cmdLine*.

Form 2 TModule(const char far* name, HINSTANCE hInstance);
Constructs a *TModule* object that is an alias for an already loaded DLL or program with an available *HInstance*. When the *TModule* is destructed, the instance isn't automatically freed. *name*, which is optional, can be 0.

Form 3 TModule(const char far* name, bool shouldLoad = true);
Constructs a *TModule* object that is used as an alias for a DLL. If *shouldLoad* is **true**, *TModule* will automatically load and free the DLL. If *shouldLoad* is **false**, then the *HInstance* needs to be set later using *InitModule*.

Destructor
virtual ~TModule();
Destroys a *TModule* object and deletes *lpCmdLine*.

Public data members

lpCmdLine
char far* lpCmdLine;
A null-terminated string, *lpCmdLine* points to a copy of the command-line arguments passed when the module is loaded. Notice that *lpCmdLine* is different from the WIN32 *lpCmdLine* in which the full path name of the module is appended to the command-line arguments. Whether running under WIN16 or WIN32, ObjectWindows *TModule::lpCmdLine* data member includes only the command-line arguments. Note that the run-time library global variables **_argv*[]** and **_argc** contain identical information for both WIN16 and WIN32 APIs, and that **_argv[0]** points to the full path name of the module.

See also TApplication

Module
extern TModule *Module;
Holds a global pointer to the current module.

See also TApplication

Status
TStatus Status;
Status contains the module status and is included for backward compatibility with ObjectWindows 1.0 applications. ObjectWindows 2.0 instead uses exceptions to handle errors. Setting *Status* to any nonzero value will throw a *TXCompatibility* exception.

See also TXCompatibility::MapStatusCodeToString

Public member functions

AccessResource
int AccessResource(HRSRC hRsrc) const;
Used for 16-bit applications, *AccessResource* finds the specified resource. The preferred method is to use *FindResource*.

See also TModule::FindResource

AllocResource

HGLOBAL AllocResource(HRSRC hRsrc, uint32 size) const;

Used for 16-bit applications, *AllocResource* loads a resource into memory. The preferred method is to use *LoadResource*.

See also TModule::LoadResource

CopyCursor

HCURSOR CopyCursor(HCURSOR hCursor) const;

Used for 16-bit applications, *CopyCursor* copies the cursor specified in *hCursor*. The return value is a handle to the duplicate cursor.

See also TIcon

CopyIcon

HICON CopyIcon(HICON hIcon) const;

Copies the icon specified in *hIcon*. The return value is a handle to the icon or 0 if unsuccessful. When no longer required, the duplicate icon should be destroyed.

Error

Form 1 virtual void Error(int errorCode);

Processes errors identified by the error value supplied in *errorCode*. *Error* displays the error code in a message box and asks the user if it is OK to continue. If the user does continue, the program might or might not be able to recover. If the user does not continue, the program terminates. *Error* can be overridden with another kind of exception handler. This function is included only for backward compatibility with ObjectWindows 1.0. If you are writing ObjectWindows 2.0 applications, use the following *Error* function instead.

Form 2 virtual int Error(xmsg& x, unsigned captionResId, unsigned promptResId=0);

Called when fatal exceptions occur, *Error* takes an *xmsg* exception object, a resource ID for a message box caption, and an optional resource ID for a user prompt. By default, *Error* calls *HandleGlobalException* with the *xmsg* object and the strings obtained from the resources. An application (derived from *TApplication* which is derived from *TModule*) can reimplement this function to provide alternative behavior.

A nonzero status code is returned to indicate that an error condition is to be propagated; a zero status indicates that the condition has been handled and that it is OK to proceed. ObjectWindows uses this status code inside its message loop to allow the program to resume. The global error handler (defined in except.h), which displays the message text, is

```
int _OWLFUNC HandleGlobalException(xmsg& x, char* caption, char* canResume);
```

ExecDialog

int ExecDialog(TDialog* dialog);

Executes a dialog box. This function is included only for backward compatibility. Use *TDialog::Execute* instead.

FindResource

HRSRC FindResource(TResId id, const char far* type) const;

Finds the resource indicated by *id* and *type* and, if successful, returns a handle to the specified resource. If the resource cannot be found, the return value is zero. The *id* and

type parameters either point to zero-terminated strings or specify an integer value. *type* can be one of the standard resource types defined below.

Value of type	Resource
RT_ACCELERATOR	Accelerator table
RT_BITMAP	Bitmap
RT_CURSOR	Cursor
RT_DIALOG	Dialog box
RT_FONT	Font
RT_FONTDIR	Font directory
RT_ICON	Icon
RT_MENU	Menu
RT_RCDATA	User-defined resource
RT_STRING	String

See also TResID

GetClientHandle

```
HWND GetClientHandle(HWND hWnd);
```

Gets the handle to the client window. This function is included only for backward compatibility with ObjectWindows 1.0.

GetClassInfo

```
bool GetClassInfo(const char far* name, WNDCLASS far* wndclass) const;
```

Used particularly for subclassing, *GetClassInfo* gets information about the window class specified in *wndclass*. *name* points to a zero-terminated string that contains the name of the class. *wndclass* points to the WNDCLASS structure that receives information about the class. If successful, *GetClassInfo* returns nonzero. If a matching class cannot be found, *GetClassInfo* returns zero.

See also WNDCLASS struct

GetInstance

```
HINSTANCE GetInstance() const;
```

Returns the instance handle for this module.

GetInstanceData

```
int GetInstanceData(void* data, int len) const;
```

Used only for 16-bit applications, *GetInstanceData* gets data from an already running instance of an application. *len* is the size of the buffer.

GetModuleFileName

```
int GetModuleFileName(char far* buff, int maxChars);
```

Returns the expanded file name (path and file name) of the file from which this module was loaded. *buff* points to a buffer that holds the path and file name. *maxChars* specifies the length of the buffer. The expanded file name is truncated if it exceeds this limit. *GetModeFileName* returns 0 if an error occurs.

GetModuleUsage

```
int GetModuleUsage() const;
```

Used only for 16-bit applications, *GetModuleUsage* returns the reference count of the module, if successful. The reference count is incremented by one each time a module is loaded, and decremented by one each time a module is freed.

GetName

```
const char far* GetName() const;
```

Gets the name of the module.

See also ::GetName (Windows API)

GetParentObject

```
TWindow* GetParentObject(HWND hWndParent);
```

Gets a handle to the parent window. This function is included only for backward compatibility with ObjectWindows 1.0.

See also ::GetParentObject (Windows API)

GetProcAddress

```
FARPROC GetProcAddress(const char far* fcnName) const;
```

Returns the entry-point address of the exported function *fcnName* if the function is found. Returns NULL otherwise.

InitModule

```
void InitModule(HINSTANCE hInstance, const char far* cmdLine);
```

Performs any instance initialization necessary for the module. If the module cannot be created, a *TXInvalidModule* exception is thrown.

IsLoaded

```
bool IsLoaded() const;
```

Returns a nonzero value if the instance handle is loaded. Use this function primarily to ensure that a given instance is loaded.

LoadAccelerators

```
HACCEL LoadAccelerators(TResId id) const;
```

Loads the accelerator table resource specified by *id*. *LoadAccelerators* loads the table only if it has not been previously loaded. If the table has already been loaded, *LoadAccelerators* returns a handle to the loaded table.

LoadBitmap

```
HBITMAP LoadBitmap(TResId id) const;
```

Loads the bitmap resource specified by *id*. If the bitmap cannot be found, *LoadBitmap* returns 0.

See also TBitMap, TResID, ::LoadBitmap (Windows API), OBM_XXXX values (Windows API)

LoadCursor

```
HCURSOR LoadCursor(TResId id) const
```

Loads the cursor resource specified by *id* into memory and returns a handle to the cursor resource. If the cursor resource cannot be found or identifies a resource that is not a cursor, *LoadCursor* returns 0.

See also TCursor, TResID, ::LoadCursor (Windows API)

LoadIcon

HICON LoadIcon(const char far* name) const;

Loads the icon resource indicated by the parameter, *name*, into memory. *LoadIcon* loads the icon only if it has not been previously loaded. If the icon resource cannot be found, *LoadIcon* returns 0.

LoadIcon can be used to load a predefined Windows icon if *name* points to one of the Windows IDI_XXXX values.

See also TIcon, ::LoadIcon (Windows API), IDI_XXXX values (Windows API)

LoadMenu

HMENU LoadMenu(TResId id) const;

Loads the menu resource indicated by *id* into memory. If the menu resource cannot be found, *LoadMenu* returns 0.

See also TMenu, ::LoadMenu (Windows API)

LoadResource

HGLOBAL LoadResource(HRSRC hRsrc) const;

Loads a resource indicated by *hRsrc* into memory and returns a handle to the memory block that contains the resource. If the resource cannot be found, the return value is 0. The *hRsrc* parameter must be a handle created by *FindResource*.

LoadResource loads the resource into memory only if it has not been previously loaded. If the resource has already been loaded, *LoadResource* increments the reference count by one and returns a handle to the existing resource. The resource remains loaded until it is discarded.

LoadString

Form 1 int LoadString(uint id, char far* buff, int maxChars) const;

Loads a string resource identified by *id* into the buffer pointed to by *buff*. *maxChars* indicates the size of the buffer to which the zero-terminated string is copied. A string longer than the length specified in *maxChars* is truncated. The return value is the number of characters copied into the buffer, or 0 if the string resource does not exist.

Form 2 string LoadString(uint id) const;

Loads a string resource identified by *id*

LowMemory

bool LowMemory();

This function, which is obsolete, always returns 0.

MakeWindow

TWindow* MakeWindow(TWindow* win);

This function is obsolete. Use the *TWindow::Create* function instead.

operator HINSTANCE()

operator HINSTANCE() const;

Returns the handle of the application or DLL module represented by this *TModule*. The handle must be supplied as a parameter to Windows when loading resources.

operator ==

bool operator ==(const TModule& other) const;

Returns **true** if this instance is equal to the other instance; otherwise, returns **false**.

RestoreMemory

void RestoreMemory();

This function, which is obsolete, restores memory.

SetInstance

void SetInstance(HINSTANCE hInstance);

Sets the instance handle for this *TModule*. *SetInstance* is used for special cases in which the *hInstance* is not known when the module is constructed.

SetName

void SetName(const char far* name);

Accessor function that sets the name of the module.

SetResourceHandler

const RSRCHDLRPROC SetResourceHandler(const char far* type, RSRCHDLRPROC loadProc) const;

Used for 16-bit applications, *SetResourceHandler* installs a callback function that loads resources. *type* points to a resource type. *loadProc* is the address of the callback procedure. If successful, *SetResourceHandler* returns a pointer to a previously installed resource handler. If no resource handler has been installed, *SetResourceHandler* returns a pointer to the default handler. This function is useful for handling user-defined resource types.

SizeofResource

uint32 SizeofResource(HRSRC hRsrc) const;

Returns the size, in bytes, of the resource indicated by *hRscr*. The resource must be a resource handle created by *FindResource*. If the resource cannot be found, the return value is 0.

Because of alignment in the executable file, the returned size might be larger than the actual size of the resource. An application cannot rely on *SizeofResource* for the exact size of a resource.

ValidWindow

TWindow* ValidWindow(TWindow* win);

This function, which is obsolete, returns a handle to the valid window.

Protected data members

HInstance

HINSTANCE HInstance;

Contains the executing instance of either the application or DLL module. The instance must be supplied as a parameter to Windows when loading resources.

Name
int GetModuleFileName(char far* buff, int maxChars);
Returns the expanded file name (path and file name) of the file from which this module was loaded. *buff* points to a buffer that holds the path and file name. *maxChars* specifies the length of the buffer. The expanded file name is truncated if it exceeds this limit. *GetModeFileName* returns 0 if an error occurs.

TModule::TXInvalidModule class — module.h

A nested class, *TXInvalidModule* describes an exception that results from an invalid module. A window throws this exception if it can't create a valid *TModule* object.

Public constructor

Constructor
TXInvalidModule();
Constructs a *TXInvalidModule* object.

Public member functions

Clone
TXOwl* Clone();
Copies the *TXInvalidModule* exception object.

Throw
void Throw();
Throws the *TXInvalidModule* exception object.

TOleClientDC class — olewindo.h

Derived from *TClientDC*, *TOleClientDC* is a helper class that translates between two different coordinate systems. For example, the window's logical points may be measured in HIMETRIC or *twips* whereas the actual ouput device's (the viewport) coordinates may be measured in pixels. Without the help of this class, you would need to create a client DC and then set up the window's logical coordinates (its *origin*) and its width and height (its *extent*) as well as the viewport's origin (measured in device coordinates) and extent. Instead, *TOleClientDC* performs these calculations for you by mapping logical points to device points and vice versa.

TOleClientDC works with a *TOleWindow* object. By default, *TOleClientDC* takes care of both scaling (adjusting the extents of the window and the viewport) and scrolling (adjusting the origins of the window and the viewport).

Public constructor

Constructor
TOleClientDC(TOleWindow& win, bool scale = true);
Constructs a *TOleClientDC* object . The parameter *win* references the window that *TOleClientDC* uses to create a DC. If the *scale* parameter is **true**, *TOleClientDC* takes care of scaling. However, if your application handles scaling, you can pass *scale* as **false**.

Scrolling is controlled by the presence of a scroller (*TScroller*). *TOleClientDC*, by default, takes care of both scaling and scrolling.

See also TClientDC, TOleWindow

TOleDocument class

oledoc.h

Derived from *TStorageDocument*, *TOleDocument* implements the Document half of the Doc/View pair. That is, *TOleDocument* manages the document's data while the corresponding *TOleView* object determines how the data is displayed on the screen. Basically, *TOleDocument* is a *TStorageDocument* with a knowledge of *TOcDocument* through its pointer to *TOcDocument*.

TOleDocument is responsible for creating compound documents (documents that can hold a variety of embedded objects from various source applications), closing documents, reading documents from storage (an area within a file where data is stored), and writing documents to storage. In the case of a server, the document consists of a single object. In the case of a container, the document can consist of one or more embedded objects (also referred to as *parts*).

To accomplish these tasks, *TOleDocument* talks to the underlying ObjectComponents classes through the use of functions such as *GetOcApp*, *GetOcDoc*, and *SetOcDoc*.

See also

TOleView, TStorageDocument, TOcDocument, TOcRemView

Public constructor and destructor

Constructor
TOleDocument(TDocument* parent = 0);
Constructs a *TOleDocument* object associated with the given parent *TDocument* object.

Destructor
~TOleDocument();
Destroys the *TOleDocument* object. In the case of an OLE container, the compound file remains open until all of the views shut down.

Public member functions

CanClose

virtual bool CanClose();

Prepares the document for closing. Before closing the current document, checks to see if all child documents can be closed. If any child returns **false**, *CanClose* returns **false** and aborts the process. If all children return **true**, *CanClose* checks to see if the document has been changed. If so, it asks the user to save the document, discard any changes, or cancel the operation. If the document has not been changed, and all child documents return **true**, this *CanClose* function returns **true**, thus indicating that the document can be closed.

CanClose also calls *ReleaseDoc* on its associated ObjectComponents document to make sure that all the embedded objects are closed properly.

See also TOleDocument::ReleaseDoc

Close

bool Close();

Ensures that the *IStorage* is released properly and disconnects any active server in the document. A compound file must be closed before it is reopened.

See also TTOleDocument::Open

Commit

bool Commit(bool force);

Commits the current document's data to storage. If *force* is **true** and the data is not dirty, all data is written to storage and *Commit* returns **true**. If *force* is **false**, the data is written only if it is dirty.

The data has been changed since the last save operation.

See also TTOleDocument::Revert

GetNewStorage

virtual IStorage*GetNewStorage();

Typically used in a SaveAs menu selection, *GetNewStorage* gets a new storage for the document.

If the document's path changes, for example, use this function to create a new storage.

GetOcApp

TOcApp* GetOcApp();

Returns the ObjectComponents application associated with this *TOleDocument* object.

See also TOcApp

GetOcDoc

TOcDocument*GetOcDoc();

Returns the ObjectComponents document associated with this *TOleDocument* object.

See also TOleDocument::SetOcDoc

InitDoc

virtual bool InitDoc();

Overrides *TDocument*'s *InitDoc* function and creates or opens a compound file so that there is an *IStorage* associated with this document's embedded objects. Uses a *TOcDocument* object to perform the the actual interaction with the OLE *IStorage* and *IStream* interfaces, which are ultimately responsible for establishing the relationship between a compound file and its storage.

See also TOcDocument, TDocument::InitDoc

Open

bool Open(int mode, const char far* path);

Loads the embedded objects, if any, using the path specified in *path*. *mode* is a combination of bits that specify how the embedded objects are opened (for example, read only, read/write, and so on).

See also TTOleDocument::Close

PathChanged

bool PathChanged();

Checks if the current document's path is the same as the *TOcDocument*'s path. If the paths are not the same, *PathChanged* returns **true**.

See also TOcDocument

PreOpen

virtual void PreOpen();

Before the document is actually opened, PreOpen gives the derived class a chance to perform a particular operation, for example, setting a different open mode for the compound document.

See also dtxxxx document constants

Read

virtual bool Read();

Loads the embedded objects from the compound file. A container should call this function when it wants to loads any embedded objects.

See also TOleDocument::Write

ReleaseDoc

virtual bool ReleaseDoc();

Releases the ObjectComponents document when the server is finished using the document.

See also TOleDocument::CanClose

Revert

bool Revert (bool clear);

Performs the reverse of Commit. Cancels any changes made to the document since the last time the document was saved to storage.

See also TTOleDocument::Commit

SetOcDoc

void SetOcDoc(TOcDocument* doc);

Sets the ObjectComponents document associated with this *TOleDocument* object.

See also TOleDocument::GetOcDoc

SetStorage

virtual bool SetStorage(IStorage* stg);

Attaches the *IStorage* pointer (*stg*) to this document. If successful, *SetStorage* returns **true**.

An OLE storage interface that ObjectComponents implements when it needs to assign storage to a document

Write

virtual bool Write();

Saves the embedded objects to the compound file. A container should call this function when it wants to save its embedded objects to storage.

See also TOleDocument::Read

TOleFactoryBase<> class — olefacto.h

A template class, *TOleFactoryBase<>* creates callback code for ObjectWindows classes. The main purpose of the factory code is to provide a callback function, *Create*, that ObjectComponents calls to create objects.

Just as a recipe consists of a list of instructions about how to make a particular kind of food, a template, such as *TOleFactoryBase<>*, contains instructions about how to make an object, in this case, a factory object. *TOleFactoryBase<>* includes two public member functions. The three additional functions are passed as template arguments. Although these template arguments actually belong to the class that is instantiated when you fill in the arguments to *TOleFactoryBase<>*, they are described here for convenience.

Use *TOleFactoryBase<>* to manufacture objects in general, whether or not they are embedded, OLE-enabled, or use the Doc/View model. These objects might or might not be connected to OLE.

The callouts are supplied through the arguments passed to the template class. The factory base class takes three template parameters: the application type, a set of functions to create the object, and a set of functions to create an automation object. Depending on the arguments passed, you can make the following OLE-enabled components:

- Doc/View components that are automated
- Doc/View components that are not automated
- Non-Doc/View components that are automated
- Non-Doc/View components that are not automated

ObjectWindows provides a standard implementation for object creation and automation. Factory Template Classes gives an overview of these classes.

By using *TOleFactoryBase<>* to obtain an OLE interface for your application, you can make objects that are accessible to OLE. That is, *TOleFactoryBase<>* handles any relationships with *IUnknown*, a standard OLE interface.

See also

TComponentFactory typedef, TOcRegistrar class, TAutoFactory class

Public member functions

TComponentFactory

operator TComponentFactory();

Converts the object into a pointer to the factory. ObjectComponents uses this pointer to create the object.

Create

static IUnknown* Create(IUnknown* outer, uint32 options, uint32 id);

A *TComponentFactory* callback function that creates or destroys the application or creates objects. If an application object does not already exist, *Create* creates a new one. The *outer* argument points to the OLE2 *IUnknown* interface with which this object aggregates itself. If *outer* is 0, If *outer* is 0, the object will become an independent object.

The *options* argument indicates the application's mode while it is running. The values for *options* are either set from the command line or set by ObjectComponents. They are passed in by the *Registrar* to this callback. The application looks at these flags in order to know how to operate, and the factory callback looks at them in order to know what to do. For example, a value of *amExeMode* indicates that the server is running as an .EXE either because it was built as an .EXE or because it is a .DLL that was launched by an .EXE stub and is now running as an executable program. The *TOcAppMode enum* description shows the possible values for the *options* argument.

If the application already exists and the object ID (*id*) equals 0, *Create* returns the application's OLE interface. Otherwise, it calls *OCInit* to create a new *TOcApp* and register the options from *TOcAppMode* enum, which contains OLE-related flags used in the application's command line. (These flags tell the application whether it has been run as a server, whether it needs to register itself, and so on.) If a component ID was passed, that becomes the component; otherwise, *Create* runs the application itself based on the values of the *TOcAppMode* enum and returns the OLE interface.

See also TOleFactoryBase::DestroyApp, TOcAppMode enum

Template arguments

CreateApp

static T* CreateApp(options);

Creates a new application. By default, it creates a new application of template type *T* with no arguments. This is a static function that you can override in your application.

The *options* are those passed to the factory. They can be one of the *TOcAppMode* enum values (for example, *amRun*, *amEmbedding*, and so on) that indicate the application's mode when running and indicate the options to the factory.

See also TOleFactoryBase::DestroyApp

CreateObject

static IUnknown* CreateObject(TApplication* app, TDocTemplate* tpl, IUnknown* outer);

Creates an object using the document template referred to in *tpl* and the application specified in *app*. The *outer* parameter refers to the controlling *IUnknown* interface of the object with which this object is going to be aggregated. If *outer* is 0, the object is an independent object.

You can override this static function to create your own object at run time.

See also TOleFactoryBase::DestroyApp

TOcView class

DestroyApp

static void DestroyApp(T* app);

Destroys the application (*app*) by unregistering the object and deleting it.

See also TOleFactoryBase::CreateApp

TOleFrame class — oleframe.h

Derived from *TDecoratedFrame*, *TOleFrame* provides user-interface support for the main window of a Single Document Interface (SDI) OLE application. Because it inherits *TDecoratedFrame*'s functionality, *TOleFrame* is able to position decorations, such as toolbars, around the client window. Because of its OLE frame functionality, you will always want to create a *TOleFrame* as a main frame. For example, *TOleFrame* supports basic container operations, such as

- Creating space in a container's frame window that the server has requested
- Merging the container's menu and the server's menu
- Processing accelerators and other messages from the server's message queue

In addition to supporting the customary frame window operations and event-handling, *TOleFrame* provides functionality that supports OLE 2 menu merging for pop-up menus.

Through the use of the EvOcXxxx event-handling member functions, *TOleFrame* responds to ObjectComponents messages sent to both the server and the container applications. Although most of the messages and functions provide container support, one message, *EvOcAppShutDown*, is server related, and one function, *GetRemViewBucket*, supplies server support. Whether *TOleFrame* functions as a container or a server, it always has a pointer to a corresponding *TOcApp*.

See also

TOcApp, TDecoratedFrame

Public constructor and destructor

Constructor

TOleFrame(const char far* title, TWindow* clientWnd, bool trackMenuSelection = false, TModule* module = 0);
Constructs a *TOleFrame* object with the specified caption for the frame window (*title*), the client window (*clientWnd*), and module instance. The *trackMenuSelection* parameter indicates whether or not the frame window should track menu selections (that is, display hint text in the status bar of the window when a menu item is highlighted).

Destructor

~TOleFrame();
Destroys a *TOleFrame* object.

Public member functions

AddUserFormatName

void AddUserFormatName(char far* name, char far* resultName, char far* id);
Adds user defined formats and the corresponding names to the list of Clipboard formats. Use this function if you want to associate a clipboard data format (*name*) with the description of the data format as it appears to users in the Help text of the Paste Special dialog box (*resultName*). To use a standard Clipboard format, set the *id* parameter to an appropriate constant (for example, CF_TEXT). Otherwise, if the format is identified by a string, pass the string as the name and omit the ID.

See also TOcApp::AddUserFormatName

GetOcApp

TOcApp* GetOcApp();
Gets the ObjectComponents application associated with this frame window.

See also TOcApp, TOleFrame::SetOcApp

GetRemViewBucket

TWindow* GetRemViewBucket();
Returns a pointer to the OLE frame's hidden helper window that holds all inactive server windows. It can also hold in-place tool bars and *TOleView* windows.

SetOcApp

void SetOcApp(TOcApp* app);
Sets the ObjectComponents application associated with this frame window to the applications specified in the *app* parameter.

See also TOcApp, TOleFrame::GetOcApp

Protected member functions

CanClose

bool CanClose();

Returns **true** if the frame window can be closed. Tests to see if both the *TOcApp* and all child windows can close. If the application and all child windows return **true**, *CanClose* closes the frame window.

See also TOcApp

CleanupWindow

void CleanupWindow();

Performs normal window cleanup of any HWND-related resources. For DLL servers, *CleanupWindow* destroys the idle timer.

See also TWindow::CleanupWindow, TOleFrame::EvTimer

Destroy

void Destroy(int retVal);

Checks with all the connected servers to ensure that they can close before destroying the frame window. If the user closes the application with objects still embedded, *Destroy* hides the frame window instead of destroying it.

DestroyStashedPopups

void DestroyStashedPopups();

Destroys the previously stored shared pop-up menus. Checks to see if *StashCount* is 0 before destroying the menus.

See also TOleFrame::StashContainerPopups, TOleFrame::StashCount

EvActivateApp

void EvActivateApp(bool active, HTASK hTask);

Responds to a WM_ACTIVATEAPP message sent when a window is activated or deactivated. If active is **true**, the window is being activated.

This message is sent to the top-level window being deactivated before it is sent to the top-level window being activated. *hTask* is a handle to the current process.

This event is forwarded to the *TOcApp* object, which activates an in-place server if one exists.

See also TOcApp

EvOcAppBorderSpaceReq

bool EvOcAppBorderSpaceReq(TRect far* rect);

Responds to an OC_APPBORDERSPACEREQ message sent to a container. The response is to ask the container if it can give border space in its frame window to the server.

See also TOcApp::BorderSpaceSet

EvOcAppBorderSpaceSet

bool EvOcAppBorderSpaceSet(TRect far* rect);

Responds to an OC_APPBORDERSPACESET message by making room in the container's frame window for the border space that the server has requested.

See also TOcApp::BorderSpaceReq

EvOcAppDialogHelp

void EvOcAppDialogHelp(TOcDialogHelp far& dh);

Responds to an OC_APPDIALOGHELP message. The *dh* parameter refers to one of the *TOcDialogHelp* enum constants that indicate the kind of dialog box the user has open. For example, *dhBrowseLinks* indicates that the Links dialog box is open. The *TOcDialogHelp* enum lists the help constants and their dialog box equivalents.

See also TOcDialogHelp enum

EvOcAppFrameRect

bool EvOcAppFrameRect(TRect far* rect);

Responds to an OC_APPFRAMERECT message sent to a container. The response is to get the coordinates of the client area rectangle of the application's main window.

EvOcAppInsMenus

bool EvOcAppInsMenus(TOcMenuDescr far& sharedMenu);

Responds to an OC_APPINSMENUS message by merging the container's menu into the shared menu. The *sharedMenu* parameter refers to this merged menu.

EvOcAppMenus

bool EvOcAppMenus(TOcMenuDescr far& md);

Responds to an OC_OCAPPMENUS sent to the container. The response is to install a merged menu bar.

EvOcAppProcessMsg

bool EvOcAppProcessMsg(MSG far* msg);

Responds to an OC_APPROCESSMSG message sent to the container asking the server to process accelerators and other messages from the container's message queue.

EvOcAppRestoreUI

void EvOcAppRestoreUI();

Responds to an OC_APPRESTOREUI message by restoring the container's normal menu and borders because in-place editing has finished.

EvOcAppShutdown

bool EvOcAppShutdown();

Responds to an OC_APPSHUTDOWN message indicating that the last embedded object has been closed. The response is to shut down the server.

EvOcAppStatusText

void EvOcAppStatusText(const char far*);

Responds to an OC_APPSTATUSTEXT message by displaying text from the server on this container's status bar.

EvOcEvent

LRESULT EvOcEvent(WPARAM wParam, LPARAM lParam);

Responds to a WM_OCEVENT message and subdispatches it to one of the EvOcXxxx event-handling functions based on the value of *wParam*. WM_OCEVENT messages are sent by ObjectComponents when it needs to communicate with an OLE-generated event; for example, if a server wants to display toolbars.

EvSize

void EvSize(uint sizeType, TSize& size);

Responds to an EV_WM_SIZE message indicating a change in the frame window's size and forwards this information to the *TOcApp*, which then notifies any in-place server. The server uses this information to change the size of its tool bar, if necessary.

See also TOcApp::EvActivate

EvTimer

void EvTimer(uint timerId);

If this is a .DLL server, *EvTimer* responds to a timer message by running an idle loop if the message queue is empty.

See also TOleFrame::CleanupWindow

SetupWindow

void SetupWindow();

Associates the ObjectComponents application with the window's HWND so that the *TOcApp* and the window can communicate. Prepares a place to insert the server's toolbars when in-place editing of the embedded object occurs.

See also TOcApp

StashContainerPopups

void StashContainerPopups(const TMenuDescr& shMenuDescr);

Stores a local copy of the pop-up menus so they can be used for menu merging and then destroyed later by *DestroyStashedPopups*. *shMenuDescr* is the shared menu descriptor to be stored. Increments *StashCount* each time the pop-up menus are saved.

See also TOleFrame::DestroyStashedPopups, TMenuDescr

Protected data members

HoldMenu

HMENU HoldMenu;

Holds the handle to the container's previously saved copy of the menu.

OcApp

TOcApp* OcApp;

Points to the ObjectComponents application associated with this frame window.

See also TOcApp

StashCount

int StashCount;

Holds the number of menu bars that have been stored. This number indicates how many active in-place editing sessions you have open.

See also StashContainerPopups, TOleFrame::DestroyStashedPopups

StashedContainerPopups

TMenu StashedContainerPopups;

Holds the stored, shared pop-up menus.

See also StashContainerPopups, TOleFrame::StashCount

Response table entries

Response table entry	Member function
EV_WM_SIZE	EvSize
EV_WM_ACTIVATEAPP	EvActivateApp
EV_MESSAGE(WM_OCEVENT, EvOcEvent)	EvOcEvent
EV_OC_APPINSMENUS	EvOcAppInsMenus
EV_OC_APPMENUS	EvOcAppMenus
EV_OC_APPROCESSMSG	EvOcAppProcessMsg
EV_OC_APPFRAMERECT	EvOcAppFrameRect
EV_OC_APPBORDERSPACEREQ	EvOcAppBorderspaceReq
EV_OC_APPBORDERSPACESET	EvOcAppBorderSpaceSet
EV_OC_APPSTATUSTEXT	EvOcAppStatusText
EV_OC_APPRESTOREUI	EvOcAppRestoreUI
EV_OC_APPSHUTDOWN	EvOcAppShutDown

TOleMDIFrame class

olemdifr.h

Derived from *TMDIFrame* and *TOleFrame*, *TOleMDIFrame* provides OLE user-interface support for the main window of a Multiple Document Interface (MDI) application. *TOleMDIFrame* also talks directly to the ObjectComponents classes through the use of a pointer to the *OcApp* object.

TOleMDIFrame inherits from *TMDIFrame* functionality that supports the use of MDI frame windows designed to serve as the main windows of an MDI-compliant application. From *TOleFrame*, *TOleMDIFrame* inherits decorated frame window functionality that supports the addition of decorations (such as toolbars and status lines) to the frame window. *TOleMDIFrame* also inherits the ability to

- Create space that the server has requested in a container's frame window
- Merge the container's menu into the server's menu
- Process accelerators and other messages from the server's message queue
- Support OLE 2 menu merging for pop-up menus

TOleMDIFrame also inherits from *TOleFrame* the ability to talk directly to the ObjectComponents classes through the use of a pointer to the *OcApp* object.

See Step 14 of the ObjectWindows tutorial for an example of a program that uses *TOleMDIFrame* to create an OLE-enabled decorated MDI frame window.

See also

TMDIFrame, TDecoratedMDIFrame, TOleFrame, TOleFrame::SetOcApp

Public constructor and destructor

Constructor

```
TOleMDIFrame(const char far* title, TResIdmenuResId, TMDIClient& clientWnd = *new TMDIClient,
  bool trackMenuSelection = false, TModule*module = 0);
```

Constructs a *TOleMDIFrame* object with the indicated title, menu resource ID, client window, and module instance. By default, because *trackMenuSelection* is false, menu hint text is not displayed. (These parameters coincide with those of *TMDIFrame*'s constructor.)

Destructor

```
~TOleMDIFrame();
```

Destroys the OLE MDI frame window object.

See also TDecoratedMDIFrame

Protected member functions

DefWindowProc

```
LRESULT DefWindowProc(uint message, WPARAM wParam, LPARAM lParam);
```

Allows default processing for all messages except for a resizing message concerning the frame window, in which case, *DefWindowProc* returns nothing.

EvActivateApp

```
void EvActivateApp(bool active, HTASK hTask);
```

Responds to a message indicating that the frame window of this application (*hTask*) is going to be either activated (*active* is **true**) or deactivated (*active* is **false**), and forwards this information to the *TOcApp* object.

See also TOcApp

EvOcAppInsMenus

```
bool EvOcAppInsMenus(TOcMenuDescr far*);
```

Inserts menus into a provided menu bar, or merges them with a child window and servers. To do this, *EvOcAppInsMenus* creates a temporary composite menu for the frame and MDI child windows, then copies the shared menu widths to the ObjectComponents structure. It saves the container popups so they can be destroyed later.

Response table entries

Response table entry	Member function
EV_WM_ACTIVATEAPP	*EvActivateApp*
EV_OC_APPINSMENUS	*EvOcAppInsMenus*

TOleView class

oleview.h

Derived from *TWindowView* and *TView*, *TOleView* supports the View half of the Doc/View pair and creates a window with a view that can display an associated document. Documents use views to display themselves to a user. Regardless of whether a view belongs to a server or a container, *TOleView* sets up a corresponding *TOcDocument* object (an entire compound document).

In the case of an OLE-enabled container application, *view* refers to the window where the container application draws the compound document, which may consist of one or more linked and embedded objects. To display these objects in different formats, a container can be associated with more than one view. Similarly, to display the data properly, each embedded object can also have its own view. Each container view creates a corresponding ObjectComponents *TOcView* object.

If the view belongs to an OLE-enabled server application, *TOleView* creates a remote view on the server's document (a *TOcRemView* object). TOleView takes care of transmitting messages from the server to the container, specifically in the case of merging menus and redrawing embedded objects. TOleView supports merging the server's and the container's pop-up menu items to form a composite menu. Because it knows the dimensions of the server's view, TOleView is responsible for telling the container how to redraw the embedded object.

Similarly to *TView*, *TOleView* supports the creation of views and provides several event handling functions that allow the view to query, commit, and close views. *TOleView* also manages the writing to storage of documents that belong to a container or a server.

See also

TOcDocument, TOcView, TOcRemView, TOleWindow, TView

Public constructor and destructor

Constructor

```
TOleView(TDocument& doc, TWindow* parent = 0);
```

Constructs a *TOleView* object associated with the given document object (*doc*) and parent window (*parent*).

Destructor

```
~TOleView();
```

Destroys the *TOleView* object and detaches the view from the associated document.

Public member functions

GetViewname

const char far* GetViewName();

Overrides *TView*'s virtual *GetViewName* function and returns the name of the class (*TOleView*).

See also TView::GetViewName

GetWindow

TWindow* GetWindow();

Overrides *TView*'s virtual *GetWindow* function and returns the *TWindow* instance associated with this view.

See also TView::GetWindow

SetDocTitle

bool SetDocTitle(const char far* docname, int index);

Overrides *TView*'s and *TWindow*'s virtual *SetDocTitle* function and stores the title of the document associated with this view.

See also TView::SetdocTitle

StaticName

static const char far* StaticName();

Returns the constant string "*OleView*" that is displayed in the user interface selection box.

See also TListView::StaticName

Protected member functions

CanClose

bool CanClose();

A view uses this function to verify whether or not it can shut down. If this is a server's view window , *CanClose* checks to see if any open-editing. is occurring on any of the embedded objects in the frame window. If so, *CanClose* closes this open-editing session by disconnecting the embedded object from its server. Then, hides the server's frame window and returns **true** when appropriate. If this is a container, *CanClose* queries all documents and views and returns **true** when all documents and views can be closed.

Unlike in-place editing, which takes place in the container's window, open-editing occurs in the server's frame window.

CreateOcView

TOcView* CreateOCView(TDocTemplate* tpl, bool isEmbedded, IUnknown* outer);

Creates an ObjectComponents view associated with the embedded object. Associates the view with the document template specified in *tpl*. The *isEmbedded* parameter is true if the view is an embedded object. The *outer* parameter refers to the *IUnknown* interface with which the view will aggregate itself.

EvOcViewAttachWindow

bool EvOcViewAttachWindow(bool attach);

Attaches this view to its ObjectWindows parent window so the embedded object can be either opened and edited or deactivated. To attach a view to an embedded object, set the *attach* parameter to true. To detatch the embedded object, set the *attach* parameter to false.

EvOcViewClose

bool EvOcViewClose();

Asks the server to close the view associated with this document. Tests to see if the document has been changed since it was last saved. Returns **true** if the document and its associated view are closed.

See also TOleView::EvOcViewSavePart

EvOcViewInsMenus

bool EvOcViewInsMenus(TOcMenuDescr far& sharedMenu);

Inserts the server's menu into the composite menu. Determines the number of groups and the number of pop-up menu items to insert within each group. The shared menu (*sharedMenu*) is the container's menu merged with the server's menu groups.

See also TMenuDescr has more information about menu merging, TOcMenuDescr struct

EvOcViewLoadPart

bool EvOcViewLoadPart(TOcSaveLoad far& ocLoad);

Asks the server to load itself from storage. Loads the document and its associated view.

EvOcViewOpenDoc

bool EvOcViewOpenDoc(const char far* path);

Asks the container application to open an existing document so the document can receive embedded and linked objects. (Actually, *TOleView* calls on the *TOleDocument* object to read the document from storage, using the standard OLE *IStorage* and *IStream* interfaces). Assigns a unique string identifier to the document and returns **true** if successful.

See also TOleView::EvOcViewClose

EvOcViewPartInvalid

bool EvOcViewPartInvalid(TOcChangeInfo far& changeInfo);

Notifies the active view of any changes made to the embedded object's data (*changeInfo*). Also, notifies any other views associated with this document that the bounding rectangle for the document is invalid and needs to be repainted. *EvOcViewPartInvalid* always returns **true**.

EvOcViewSavePart

bool EvOcViewSavePart(TOcSaveLoad far& ocSave);

Asks the server to save the embedded object's data to storage. To save the object, *EvOcViewSavePart* calls upon the *TOleDocument* object, which creates storage as necessary for each embedded object. Saves the dimensions of the server's view, which the server uses to tell the container how to redraw the embedded object in the container's window.

See also TOleView::EvOcViewClose, TOleView::EvOcViewOpenDoc, TOleDocument

VnDocOpened

bool VnDocOpened (int omode);

Ensures that TOleView's data members such as DragPart, Pos, and Scale are initialized properly after a revert operation, which cancels any changes made to the document since the last time the document was saved to storage.

VnInvalidateRect

bool VnInvalidateRect(LPARAM p);

Invalidates the view region specified by *p*. Use this function to invalidate the bounding rectangle surrounding an embedded object if the object has been changed, usually as a result of in-place editing. If successful, returns **true**.

Response table entries

Response table entry	Member function
EV_OC_VIEWOPENDOC	*EvOcViewOpenDoc*
EV_OC_VIEWINSMENUS	*EVOcViewInsMenus*
EV_OC_VIEWCLOSE	*EvOcViewClose*
EV_OC_VIEWSAVEPART	*EvOcViewSavePart*
EV_OC_VIEWLOADPART	*EvOcViewLoadPart*
EV_OC_VIEWATTACHWINDOW	*EvOcViewAttachWindow*

TOleWindow class

olewindo.h

Derived from *TWindow*, *TOleWindow* provides support for embedding objects in a compound document and serves as the client of a frame window. A compound document, such as the one *TOleWindow* supports, can contain many different types of embedded objects, from spreadsheets to bitmaps. In addition to providing support for a variety of basic window operations, *TOleWindow* also implements several OLE-related operations, among them,

- Responding to drag and drop events
- In-place editing (the process whereby an embedded object can be edited without having to switch to its associated server application)
- Activating an embedded object's server application
- Creating views for the container application.
- Transmitting a document's scaling information between a container's and a server's view windows.

TOleWindow has the ability to determine whether it's actiing as a server or a container. If it is a container, *TOleWindow* has a pointer to a*TOcView* or if it is a server, *TOleWindow* establishes a pointer to a *TOcRemView*. From the server's point of view, every remote view has a corresponding *TOleWindow*.

Through its many event-handling member functions, *TOleWindow* communicates with ObjectComponents to implement container and server support for embedded objects, update views, and respond to a variety of menu commands associated with the typical command identifiers (for example, CM_FILEMENU). It also supports OLE-specific verbs such as those activated from the Edit menu (for example, Edit and Open). These commands and verbs can originate from various sources such as a menu selection, a radio button, or even an internal program message.

Conversely, ObjectComponents talks to ObjectWindows by means of the various EV_OC_Xxxx messages. Some of these messages, such as EV_OC_VIEWPARTINVALID, implement container support while others, such as EV_OC_VIEWCLOSE, implement server support.

For any user-defined classes derived from *TOleWindow*, you need to choose which functions are appropriate. If you want to provide additional server support, you need to define only those functions that implement server messages; if you want to provide container support, you need to define only those functions that provide additional container support.

For example, the data embedded in the container application (a compound document having one or more embedded objects) and the data embedded in the server application (a single OLE object with or without other embedded objects) can be written to storage and loaded from storage. If you're using *TOleWindow* without *TOleView*, you have to manipulate the storage by talking directly to the ObjectComponents class, *TOcDocument*.

In addition to communicating with ObjectComponents classes, *TOleWindow* supports many transactions as a result of its interaction with other ObjectWindows classes. By virtue of its derivation from *TWindow*, naturally it inherits much of *TWindow*'s functionality.

See also

TOcView, TWindowView, TWindow

Public constructor and destructor

Constructor

```
Constructor TOleWindow(TWindow* parent = 0, TModule* module = 0);
```

Constructs a *TOleWindow* object associated with the specified parent window and module instance.

Destructor

```
~TOleWindow();
```

Checks to see if there are any open views, and, if no open views exist, destroys the *TOleWindow* object.

Public member functions

Deactivate

```
virtual bool Deactivate();
```

If an embedded object is no longer the active embedded object, either because the user has ended an in-place editing session or because the user has clicked outside the embedded object, call *Deactivate* to unselect the object. Returns **true** if successful.

GetOcApp

```
TOcApp* GetOcApp();
```

Returns the ObjectComponents application associated with this window. Every ObjectComponents application that supports linking and embedding has an associated *TOcApp* object.

See also TOcApp

GetOcDoc

```
TOcDocument* GetOcDoc();
```

Returns the ObjectComponents document associated with this window. This document can be either a container's or a server's document. If this is a *TOcDocument* created by the container, the document is an entire compound document, which may consist of one or more embedded objects. If this is a *TOcDocument* created by the server, the document is a single OLE object's data.

See also TOcDocument

GetOcRemView

```
TOcRemView* GetOcRemView();
```

Returns the server's view associated with this window. In order to draw the OLE object in the container's window, the server creates a remote view.

See also TOcRemView

GetOcView

```
TOcView* GetOcView();
```

Points to the ObjectComponents container view associated with this window. The container view holds the compound document (that is a document containing one or more embedded objects).

See also TOcView

HasActivePart

```
bool HasActivePart();
```

Returns **true** if the container's view holds an in-place active embedded object.

See also TOcView::GetActivePart, InvalidatePart

PaintMetafile

```
virtual void PaintMetafile(TDC& dc, bool erase, TRect& rect);
```

Repaints the area of the server where the embedded object resides. The *dc* parameter points to the device context. *erase* is **true** if the background of the embedded object is to be repainted. *rect* refers to the area to be repainted.

See also InvalidatePart

Protected data members

ContainerName
string ContainerName;
Holds the name of the container. The server displays the container's name when an embedded object is being edited in the server's window (referred to as *out-of-place* editing).

DragDC
TDC* DragDC;
Points to the DC used while an object is being dragged.

See also DragPt

DragHit
TUIHandle::TWhere DragHit;
Indicates the position in the embedded object where the user points and clicks the mouse. This can be any one of the *TUIHandle::TWhere* enumerated values, for example, TopLeft, TopCenter, TopRight, MidLeft, MidCenter, MidRight, BottomLeft, BottomCenter, BottomRight, or Outside when no dragging is taking place.

See also: TUIHandle::TWhere enum

DragPart
TOcPart* DragPart;
Points to the embedded object (the part) being dragged.

DragPt
TPoint DragPt;
Indicates the point (in logical units) where the mouse is over the dragged object.

See also DragDC

DragRect
TRect DragRect;
Holds the rectangle being dragged.

See also DragDC

DragStart
TPoint DragStart;
Holds the point where the dragging of the embedded object began.

See also DragPt

Embedded
bool Embedded;
Is **true** if this *TOleWindow* is an embedded window. As a result, a *TocRemView* is created because the data is being displayed in a server's window.

OcApp
TOcApp* OcApp;
Holds the ObjectComponents application associated with this TOleWindow.

See also TOcView::OcApp, TOcApp

OcDoc

TOcDocument* OcDoc;

Holds the ObjectComponents document associated with this TOleWindow.

See also TOcDocument, TOcView::OcDocument

OcView

TOcView* OcView;

Holds the ObjectComponents view or remote view (the server's) associated with the *TOleWindow* view.

See also TOcView

Pos

TRect Pos;

Holds the current area in the window where the object is embedded. *Pos* reflects the area where the object is moved if you move the object.

Scale

TOcScaleFactor Scale;

Holds the current scaling factor. The server uses this information to determine how to scale the document.

See also TOcScaleFactor, EvOcViewGetScale, SetScale

TOleWindow::ShowObjects

bool ShowObjects;

Is **true** if the embedded object's frame (the grey or shaded brushes around the object) is displayed. The frame can be turned on or off depending on how you want the object to appear.

Protected member functions

CanClose

bool CanClose();

Returns **true** if the window can be closed. Checks all the server's child windows' *CanClose* functions, which must return **true** before the window can be closed. Terminates any open editing transactions before closing the window; otherwise, passes control to *TWindow::CanClose*.

CeEditConvert

void CeEditConvert(TCommandEnabler&);

Enables a command with an ID of CM_EDITCONVERT, which lets the user convert the selected object from one format to another. This is an OLE-specific pop-up menu option.

CeEditCopy

void CeEditCopy(TCommandEnabler&);

Enables a command with an ID of CM_EDITCOPY, which lets the user copy selected object to the Clipboard.

CeEditCut

void CeEditCut(TCommandEnabler&);

Enables a command with an ID of CM_EDITCUT, which lets a user copy and delete the selected object from the view.

CeEditDelete

void CeEditDelete(TCommandEnabler&);

Enables a command with an ID of CM_EDITDELETE, which lets the user delete the selected object from the view.

CeEditInsertObject

void CeEditInsertObject(TCommandEnabler&);

Enables a command with an ID of CM_EDITCUT, which lets the user cut a section of text from the view.

CeEditLinks

void CeEditLinks(TCommandEnabler&);

Enables a command with an ID of CM_EDITLINKS, which lets the user manually update the list of linked items in the current view.

CeEditObject

void CeEditObject(TCommandEnabler&);

Enables a command with an ID of CM_EDITOBJECT, which lets the user edit the embedded object.

CeEditPaste

void CeEditPaste(TCommandEnabler&);

Enables a command with an ID of CM_EDITPASTE, which lets the user paste the embedded object from the Clipboard.

CeEditPasteLink

void CeEditPasteLink(TCommandEnabler&);

Enables a PasteLink command with an ID of CM_EDITPASTELINK, which lets the user link to the embedded object on the Clipboard. See the ocrxxxx Clipboard constants for a description of the available Clipboard formats.

See also TOleWindow::CeEditPasteSpecial, ocrxxxx Clipboard constants

CeEditPasteSpecial

void CeEditPasteSpecial(TCommandEnabler&);

Enables the PasteSpecial command, which lets the user select a Clipboard format to be pasted or paste linked. See the ocrxxxx Clipboard constants for a description of the available Clipboard formats.

See also TOleWindow::CeEditPasteLink, ocrxxxx Clipboard Constants

CeEditVerbs

void CeEditVerbs(TCommandEnabler& ce);

Enables the Edit | Verbs command, which lets the user select one of the OLE-specific verbs from the Edit menu: for example, Edit, Open, or Play.

CeFileClose

void CeFileClose(TCommandEnabler& ce);

Enables the FileClose command, which lets the user exit from the window view.

CleanupWindow

void CleanupWindow();

Performs normal window cleanup and informs the TOcView object that the window is closed.

See also TWindow::CleanupWindow, TOcView

CmEditConvert

void CmEditConvert();

Responds to a command with an ID of CM_EDITCONVERT by converting an object from one type to another.

CmEditCopy

void CmEditCopy();

Responds to a command with an ID of CM_EDITCOPY by copying the selected text to the Clipboard.

CmEditCut

void CmEditCut();

Responds to a command with an ID of CM_EDITCUT by copying the selected text to the Clipboard before cutting the text.

CmEditDelete

void CmEditDelete();

Responds to a command with an ID of CM_EDITDELETE by deleting the selected text.

CmEditInsertObject

void CmEditInsertObject();

Responds to a command with an ID of CM_EDITINSERTOBJECT by inserting an object that a user selects from a list of object types.

CmEditLinks

void CmEditLinks();

Responds to a command with an ID of CM_EDITLINKS by updating the user-selected list of linked items in the current view.

CmEditPaste

void CmEditPaste();

Responds to a command with an ID of CM_EDITPASTE by pasting an object from the Clipboard into the document.

CmEditPasteLink

void CmEditPasteLink();

Responds to a command with an ID of CM_EDITPASTELINK by creating a link between the current document and the object on the Clipboard.

See also TOleWindow::CmEditPasteSpecial, ocrxxxx Clipboard constants

CmEditPasteSpecial

void CmEditPasteSpecial();

Responds to a command with an ID of CM_EDITPASTESPECIAL by letting the user select an object from a list of available formats for pasting from the Clipboard onto the document.

See also TOleWindow::CmEditPasteLink, ocrxxxx Clipboard constants

CmFileClose

void CmFileClose();

Responds to a command with an ID of CM_FILECLOSE by posting a WM_CLOSE message to the parent window to close the application.

CreateOcView

virtual TOcView* CreateOcView(TDocTemplate* tpl, bool isEmbedded, IUnknown* outer);

Creates an ObjectComponents view associated with the embedded object. Associates the view with the document template specified in *tpl*. If *isEmbedded* is **true**, a remote view is created (that is, a *TOcRemView* instead of a *TOcView*). The *outer* parameter refers to the *IUnknown* interface with which the view will aggregate itself.

See also TOcView, TOcRemView

CreateVerbPopup

TPopupMenu* CreateVerbPopup(const TOcVerb& ocVerb);

Creates and enables a pop-up menu option (*ocVerb*) on the Edit menu. The verb describes an action (for example, Edit, Open, Play) that is appropriate for the embedded object.

See also EvDoVerb

EvCommand

LRESULT EvCommand(uint id, HWND hWndCtl, uint notifyCode);

Overrides the usual EvCommand message to handle the OLE verbs from CM_EDITFIRSTVERB to CM_EDITLASTVERB. These commands, which are defined in oleview.rh, correspond to the OLE-specific Edit menu selections such as Edit, Open, and Play. All of the other commands are passed to *TWindow::EvCommand* for normal processing.

See also CM_xxxx edit view constants, TOleWindow::EvCommandEnable, TWindow::EvCommand

EvCommandEnable

void EvCommandEnable(TCommandEnabler& commandEnabler);

Overrides the usual EvCommandEnable message in order to enable the OLE verbs from CM_EDITFIRSTVERB to CM_EDITLASTVERB. These commands enable the OLE-specific Edit menu selections, such as Edit, Open, and Play. All of the other commands are passed to *TWindow::EvCommand* for normal processing.

See also TOleWindow::EvCommand, TWindow::EvCommand

EvDoVerb

void EvDoVerb(uint whichVerb);

Executes an OLE-related menu option from the Edit menu (for example, Edit, Copy, or Play) that is associated with the selected object.

See also CreateVerbPopup

EvLButtonDblClk

void EvLButtonDblClk(uint modKeys, TPoint& point);

Responds to a mouse button double-click message. *EvLButtonDblClk* performs hit testing to see which embedded object, if any, is being clicked on, then in-place activates the embedded object.

EvLButtonDown

void EvLButtonDown(uint modkeys, TPoint& point);

Responds to a left button down message by beginning a mouse drag transaction at the given point. Performs additional hit testing to see which embedded object, if any, is being clicked on. The *modKeys* parameter holds the values for a key combination such as a shift and double click of the mouse button.

See also EvRButtonDown

EvLButtonUp

void EvLButtonUp(uint modKeys, TPoint& point);

Responds to a left button up message by ending a mouse drag action. *point* refers to the place where the mouse is located. *modKeys* holds the values for a combined key and mouse transaction.

EvMDIActivate

void EvMDIActivate(HWND hWndActivated, HWND hWndDeactivated);

Responds to a message forwarded from the MDI child window (if one exists) and lets the *TOcView* class know that the view window child window frame has been activated or deactivated.

The *hWndActivated* parameter contains a handle to the MDI child window being activated. Both the child window being activated being activated and the child window (*hWndDeactivated*) being deactivated receive this message.

See also TOcView

EvMouseMove

void EvMouseMove(uint, TPoint& point);

Responds to a mouse move message with the appropriate transaction. If the mouse is being dragged, the embedded object is moved. If a resizing operation occurs, then the embedded object is resized. This message is handled only when a mouse dragging or resizing action involving the embedded object occurs.

EvOcEvent

LRESULT EvOcEvent(WPARAM wParam, LPARAM lParam);

Responds to a WM_OCEVENT message and subdispatches the message based on *wParam*. ObjectComponents sends WM_OCEVENT messages when it needs to communicate with an OLE-generated event; for example, if a server wants to display toolbars.

EvOcPartInvalid

bool EvOcPartInvalid(TOcPart far&);

Handles a WM_OCEVENT message concerning the embedded or linked object in the document and invalidates the part.

If the *TOleWindow* object is unable to handle the message, *EvOcPartInvalid* returns **false**.

EvOcViewAttachWindow

bool EvOcViewAttachWindow(bool attach);

Attaches the view to its ObjectWindows parent window so that the user can perform open editing on the embedded object, or if the embedded object has been deactivated while in-place editing was occurring.

If the *TOleWindow* object is unable to handle the message, *EvOcViewAttachWindow* returns **false**.

EvOcViewBorderSpaceReq

bool EvOcViewBorderSpaceReq(TRect far*);

Requests that the server create space for a tool bar in the view of an embedded object.

If the *TOleWindow* object is unable to handle the message, *EvOcViewBorderSpaceReq* returns **false**, the default value.

See also EvOcViewBorderSpaceSet

EvOcViewBorderSpaceSet

bool EvOcViewBorderSpaceSet(TRect far*);

Requests that the server's tool bar be placed in the container's view of an embedded object.

If the *TOleWindow* object is unable to handle the message, *EvOcViewBorderSpaceSet* reurns **false**, the default value.

See also EvOcViewBorderSpaceReq

EvOcViewClipData

HANDLE EvOcViewClipData(TOcFormat far&);

Requests Clipboard data in the format specified.

If the *TOleWindow* object is unable to handle the message, *EvOcViewClipData* returns **false**.

EvOcViewClose

bool EvOcViewClose();

Asks the server to close a currently open document and its associated view.

If the *TOleWindow* object is unable to handle the message, *EvOcViewClose* returns **false**.

EvOcViewDrag

bool EvOcViewDrag(TOcDragDrop far&);

Handles an OC_VIEWDRAG message asking the container to provide visual feedback while the user is dragging the embedded object.

If the *TOleWindow* object is unable to handle the message, *EvOcViewDrag* returns **false**.

EvOcViewDrop

bool EvOcViewDrop(TOcDragDrop far&);

Requests a given object be dropped at a specified place on the container's window.

If the *TOleWindow* object is unable to handle the message, *EvOcViewDrop* returns **false**.

EvOcViewGetPalette

bool EvOcViewGetPalette(LOGPALETTE far* far* palette);

Requests the color palette to draw the object.

If the *TOleWindow* object is unable to handle the message, *EvOcViewGetPalette* returns **false**.

EvOcViewGetScale

bool EvOcViewGetScale(TOcScaleFactor& scaleFactor);

Responds to an OC_VIEWGETSCALE message and gets the scaling for the server objectj, causing the embedded object to be displayed using the correct scaling value (for example, 120%). *scaleFactor* indicates the scaling factor, the ratio between the size of the embedded object and the size of the site where the object is to be displayed.

If the *TOleWindow* object is unable to handle the message, *EvOcViewGetScale* returns **false**.

See also Scale, TOcScaleFactor

EvOcViewGetSiteRect

bool EvOcViewGetSiteRect(TRect far* rect);

Gets the size of the rectangle (the site) where the embedded object is to be placed. *rect* refers to the size of the bounding rectangle that encloses the embedded object.

See also EvOcViewSetSiteRect

EvOcViewInsMenus

bool EvOcViewInsMenus(TOcMenuDescr far&);

Requests that the menus in a composite menu (a menu composed of both the server's and the container's menus).

If the *TOleWindow* object is unable to handle the message, *EvOcViewInsMenus* returns **false**.

EvOcViewLoadPart

bool EvOcViewLoadPart(TOcSaveLoad far* ocLoad);

Requests that an embedded object load itself.

If the *TOleWindow* object is unable to handle the message, *EvOcViewLoadPart* returns **false**.

EvOcViewOpenDoc

bool EvOcViewOpenDoc(const char far*);

Asks the container to open an existing document, which will be used for linking from the embedding site.

If the *TOleWindow* object is unable to handle the message, *EvOcViewOpenDoc* returns **false**.

EvOcViewPaint

bool EvOcViewPaint(TOcViewPaint far&);

Asks the server to paint an object at a given position on a specified device context.

If the *TOleWindow* object is unable to handle the message, *EvOcViewPaint* returns **false**.

EvOcViewPartInvalid

bool EvOcViewPartInvalid(TOcChangeInfo far& changeInfo);

Informs an active container that one of its embedded objects needs to be redrawn. Changes in the container's part should be reflected in any other, non-active views. Returns **true** after all views have been notified of the necessary changes.

If the *TOleWindow* object is unable to handle the message, *EvOcViewPartInvalid* returns **false**.

EvOcViewPartSize

bool EvOcViewPartSize(TRect far*);

The server asks itself the size of its current rectangle and lets the container know about the size of the server's view in pixels.

If the *TOleWindow* object is unable to handle the message, *EvOcViewPartSize* returns **false**.

EvOcViewSavePart

bool EvOcViewSavePart(TOcSaveLoad far& ocSave);

Asks the server to write an embedded object's data (the part as represented by the *ocSave* parameter) into storage.

If the *TOleWindow* object is unable to handle the message, *EvOcViewSavePart* returns **false**.

EvOcViewScroll

bool EvOcViewScroll(TOcScrollDir);

Asks the container to scroll the view window and updates any internal state as needed. *EvOcViewScroll* is called when the server is resizing or a drop interaction occurs near the edge of the window.

If the *TOleWindow* object is unable to handle the message, *EvOcViewScroll* returns **false**.

EvOcViewSetScale

bool EvOcViewSetScale(TOcScaleFactor& scaleFactor);

Responds to an OC_VIEWSETSCALE message and handles the scaling for server application, ensuring that the embedded object is displayed using the correct scaling values (for example, 120%). The server uses this value in its paint procedure when the embedded object needs to be redrawn. *scaleFactor* indicates the scaling factor, the ratio between the size of the embedded object and the size of the site where the object is to be displayed.

If the *TOleWindow* object is unable to handle the message, *EvOcViewSetScale* returns **false**.

See also EvOcViewGetScale, TOcScaleFactor

EvOcViewSetSiteRect

bool EvOcViewSetSiteRect(TRect far* rect);

Converts the *rect* to logical units. This area, referred to as the *site*, is measured in logical units that take into account any scaling factor. *rect* refers to the size of the bounding rectangle that encloses the embedded object.

See also EvOcViewGetSiteRect

EvOcViewSetTitle

void EvOcViewSetTitle(const char far* title);

Sets the window's caption to title. The new caption is the name of the in-place active server merged with the caption of the container's window.

In the case of an MDI child window, the new caption is the in-place server's name merged with the caption of the MDI child window. When the child window is maximized, the merged caption is appended to the end of the main frame window's caption.

EvOcViewShowTools

bool EvOcViewShowTools(TOcToolBarInfo far& tbi);

Asks the server to provide its tool bars for display in the container's window. Returns **true** if tool bars are supplied.

If the *TOleWindow* object is unable to handle the message, *EvOcViewShowTools* returns **false**.

EvOcViewTitle

const char far* EvOcViewTitle();

Asks the container for the caption in its frame window. Returns the frame window's caption.

EvRButtonDown

void EvRButtonDown(uint modKeys, TPoint& point);

Responds to a right button down message. Performs additional hit testing to see which embedded object, if any, is being clicked and displays a local menu with appropriate options for the embedded object.

point refers to the place where the mouse is located. *modKeys* holds the values for a combined key and transaction, such as a *Shift*+double-click of the mouse button.

See also EvLButtonDown

EvSetCursor

bool EvSetCursor(HWND, uint hitTest, uint);

Performs hit testing to tell where the cursor is located within the window and what object the cursor is moving over. If the cursor is within an embedded object, *EvSetCursor* changes the shape of the cursor.

When the cursor is over an inactive part and not on a handle, *EvSetCursor* uses an arrow cursor. If the cursor is on one of the handles of the embedded part, *EvSetCursor* changes the cursor to a resizing cursor.

EvSetFocus

void EvSetFocus(HWND hWndLostFocus);

Responds to a change in focus of the window. *hWndLostFocus* contains a handle to the window losing the focus. *EvSetFocus* checks to see if an in-place server exists and, if so, passes the focus to the in-place server.

EvSize

void EvSize(uint sizeType, TSize& size);

Passes the event to *TWindow::EvSize* for normal processing and forwards the event to *TOcView::EvResize* to let a possible in-place server adjust its size.

See also TOcView::EvSize

GetInsertPosition

virtual void GetInsertPosition(TRect& rect);

Gets the position (*rect*) where the embedded object is inserted. You need to override this function if you want to override any default position.

GetLogPerUnit

virtual void GetLogPerUnit(TSize& logPerUnit);

Gets the logical units (typically pixels) per inch for a document so that the document's embedded objects can be painted correctly on the screen DC.

Init

void Init();

Initializes the *TOleWindow* object with the appropriate window style and initializes the necessary data members (for example, sets the accelerator ID to IDA_OLEVIEW).

InvalidatePart

virtual void InvalidatePart(TOcInvalidate invalid);

Invalidates the area where the embedded object exists.The server uses this function to tell OLE that the part (the embedded object), has changed. OLE then asks the server to redraw the part into a new metafile so that OLE can redraw the object for the container application even when the server application is not active.

See also PaintMetafile, TOcView_GetActivePart

Paint

void Paint(TDC& dc, bool erase, TRect& rect);

Repaints the window's contents and tells each embedded object to repaint itself.

dc points to the paint DC. *erase* indicates whether the background should be erased. *rect* is a reference to the bounding rectangle of the area that needs repainting.

PaintParts

virtual bool PaintParts(TDC& dc, bool erase, TRect& rect, bool metafile);

Repaints the embedded objects on the given DC. The *erase* parameter is **true** if the background of the embedded object is to be repainted. *rect* indicates the area that needs repainting. *metafile* indicates whether or not the DC represents a metafile.

Select

virtual bool Select(uint modKeys, TPoint& point);

Selects the embedded object at the specified point (measured in logical units). Returns **true** if the object is captured by the mouse drag; otherwise, returns **false**.

SelectEmbedded

```
bool SelectEmbedded();
```

Selects the embedded object and returns **true** to indicate that the object has been selected.

SetScale

```
virtual void SetScale(uint16 percent);
```

Sets the ratio of the embedded object's size to the size of the site.

See also SetupDC

The area inside the container where the embedded object will be drawn

SetSelection

```
void SetSelection(TOcPart* part);
```

Selects the embedded object indicated in the *part* parameter. When the embedded object is selected, a selection box is drawn around the area. After an embedded object is selected, the user can perform operations on the embedded object: for example, moving, sizing, or copying the embedded object to the Clipboard.

Setup DC

```
virtual void SetupDC(TDC& dc, bool scale = true);
```

Determines the viewport's origin and extent (the logical coordinates and the size of the DC). Sets up the device context (DC) before painting the embedded object. *dc* refers to the DC and *scale* indicates that the scaling factor to use when painting the embedded object is a ratio between the site and the embedded object.

See also TOcScaleFactor, SetScale

SetupWindow

```
void SetupWindow();
```

Establishes a connection between the *TOcView* object and the view's HWND so the view can send notification messsages to the window.

VnInvalidate Rect

```
bool VnInvalidateRect(LPARAM p);
```

When the embedded object is modified, *VnInvalidateRect* sends a message to the View portion of the Doc/View pair. When *TOleWindow* receives this message, the view region is marked for erasing. *VnInvalidateRect* always returns **true**.

Response table entries

Response table entry	Member function
EV_WM_LBUTTONDOWN	EvLButtonDown
EV_WM_RBUTTONDOWN	EvRButtonDown
EV_WM_LBUTTONDBLCLK	EvLButtonDblClk
EV_WM_MOUSEMOVE	EvMouseMove
EV_WM_LBUTTONUP	EvLButtonUp
EV_WM_SIZE	EvSize
EV_WM_MDIACTIVATE	EvMDIActivate
EV_WM_SETFOCUS	EvSetFocus
EV_WM_SETCURSOR	EvSetCursor
EV_COMMAND(CM_EDITDELETE, CmEditDelete)	CmEditDelete
EV_COMMAND_ENABLE(CM_EDITDELETE, CeEditDelete)	CeEditDelete
EV_COMMAND(CM_EDITCUT, CmEditCut)	CmEditCut
EV_COMMAND_ENABLE(CM_EDITCUT, CeEditCut)	CeEditCut
EV_COMMAND(CM_EDITCOPY, CmEditCopy)	CmEditCopy
EV_COMMAND_ENABLE(CM_EDITCOPY, CeEditCopy)	CeEditCopy
EV_COMMAND(CM_EDITPASTE, CmEditPaste)	CmEditPaste
EV_COMMAND_ENABLE(CM_EDITPASTE, CeEditPaste)	CeEditPaste
EV_COMMAND(CM_EDITPASTESPECIAL, CmEditPasteSpecial)	CmEditPasteSpecial
EV_COMMAND_ENABLE(CM_EDITPASTESPECIAL, CeEditPasteSpecial)	CeEditPasteSpecial
EV_COMMAND(CM_EDITPASTELINK, CmEditPasteLink)	CmEditPasteLink
EV_COMMAND_ENABLE(CM_EDITPASTELINK, CeEditPasteLink)	CeEditPasteLink
EV_COMMAND(CM_EDITINSERTOBJECT, CmEditInsertObject)	CmEditInsertObject
EV_COMMAND_ENABLE(CM_EDITINSERTOBJECT, CeEditInsertObject)	CeEditInsertObject
EV_COMMAND_ENABLE(CM_EDITLINKS, CeEditLinks)	CeEditLinks
EV_COMMAND(CM_EDITLINKS, CmEditLinks)	CmEditLinks
EV_COMMAND_ENABLE(CM_EDITOBJECT, CeEditObject)	CeEditObject
EV_COMMAND_ENABLE(CM_EDITCONVERT, CeEditConvert)	CeEditConvert
EV_COMMAND(CM_EDITCONVERT, CmEditConvert)	CmEditConvert
EV_COMMAND(CM_EXIT, CmFileClose)	CmFileClose
EV_COMMAND_ENABLE(CM_EXIT, CeFileClose)	CeFileClose
EV_MESSAGE(WM_OCEVENT, EvOcEvent)	EvOcEvent
EV_OWLNOTIFY(vnInvalidate)	VnInvalidateRect
EV_OC_VIEWPARTINVALID	EvOcViewPartInvalid
EV_OC_VIEWSETTITLE	EvOcViewSetTitle
EV_OC_VIEWTITLE	EvOcViewTitle
EV_OC_VIEWBORDERSPACEREQ	EvOcViewBorderSpaceReq
EV_OC_VIEWBORDERSPACESET	EvOcViewBorderSpaceSet
EV_OC_VIEWDROP	EvOcViewDrop
EV_OC_VIEWDRAG	EvOcViewDrag
EV_OC_VIEWSCROLL	EvOcViewScroll
EV_OC_VIEWGETSCALE	EvOcViewGetScale

Response table entry	Member function
EV_OC_VIEWGETSITERECT	EvOcViewGetSiteRect
EV_OC_VIEWSETSITERECT	EvOcViewSetSiteRect
EV_OC_VIEWPAINT	EvOcViewPaint
EV_OC_VIEWSAVEPART	EvOcViewSavePart
EV_OC_VIEWLOADPART	EvOcViewLoadPart
EV_OC_VIEWINSMENUS	EvOcViewInsMenus
EV_OC_VIEWSHOWTOOLS,	EvOcViewshowTools
EV_OC_VIEWGETPALETTE	EvOcViewGetPalette
EV_OC_VIEWCLIPDATA,	EvOcViewclipData
EV_OC_VIEWCLOSE	EvOcViewClose
EV_OC_VIEWPARTSIZE	EvOcViewPartSize
EV_OC_VIEWOPENDOC	EvOcViewOpenDoc
EV_OC_VIEWATTACHWINDOW	EvOcViewAttachWindow
EV_OC_VIEWSETSCALE	EvOcViewSetScale

TOpenSaveDialog class

opensave.h

TOpenSaveDialog is the base class for modal dialogs that let you open and save a file under a specified name. *TOpenSaveDialog* constructs a *TData* structure and passes it the *TOpenSaveDialog* constructor. Then the dialog is executed (modal) or created (modeless). Upon return, the necessary fields are updated, including an error field that contains 0, or a common dialog extended error.

Public constructor

Constructor

```
TOpenSaveDialog(TWindow* parent, TData& data, TResId templateId = 0, const char far* title = 0,
  TModule* module = 0);
```

Constructs an open save dialog box object with the supplied parent window, data, resource ID, title, and current module object.

See also TData struct

Public member functions

GetFileTitle

```
static int GetFileTitle(const char far* fileName, char far* fileTitle, int fileTitleLen)
```

Stores the name of the file to be saved or opened.

GetFileTitleLen

```
static int GetFileTitleLen(const char far* fileName);
```

Stores the length of the file name to be saved or opened.

Protected data members

Data

TData& Data;

Stores the file name, its length, extension, filter, initial directory, default file name, extension, and any error messages.

ofn

OPENFILENAME ofn;

Contains the attributes of the file name such as length, extension, and directory. *ofn* is initialized using the fields in the *TOpenSaveDialog::TData class*. This member is not available under Presentation Manager.

See also TData struct

ShareViMsgId

static uint ShareViMsgId;

Contains the message ID of the registered *ShareViolation* message. This member is not available under Presentation Manager.

See also TData struct, ShareViolation

Protected constructor

Constructor

TOpenSaveDialog(TWindow* parent, TData& data, TModule* module);

Constructs a *TOpenSaveDialog* box object with the supplied parent, data, and current module object.

See also TData struct

Protected member functions

CmLbSelChanged

void CmLbSelChanged();

Indicates that the selection state of the file name list box in the *GetOpenFileName* or *GetSaveFileName* dialog boxes has changed. *CmLbSelChanged* is a default handler for command messages sent by *lst1* or *lst2* (the file and directory list boxes, respectively).

CmOk

void CmOk();

Responds to a click on the dialog box's OK button (with the identifier IDOK). Calls *CloseWindow* (passing IDOK).

See also TDialog::CloseWindow

DialogFunction

bool DialogFunction(uint message, WPARAM, LPARAM);

Returns **true** if a message is handled, returns *ShareViMsgId* if a sharing violation occurs, otherwise returns **false**.

DoExecute

int DoExecute();

Creates and executes a modal dialog box.

Init

void Init(TResId templateId);

Initializes a *TOpenSaveDialog* object with the current resource ID.

ShareViolation

virtual int ShareViolation();

If a sharing violation occurs when a file is opened or saved, *ShareViolation* is called to obtain a response. The default return value is OFN_SHAREWARN. Other sharing violation responses are listed in the following table. This member is not available under Presentation Manager.

Constant	Meaning
OFN_SHAREFALLTHROUGH	Specifies that the file name can be used and that the dialog box should return it to the application.
OFN_OFN_SHARENOWARN	Instructs the dialog box to perform no further action with the file name and not to warn the user of the situation.
OFN_SHAREWARN	This is the default response that is defined as 0. Instructs the dialog box to display a standard warning message.

See also TData struct, ShareViMsgId

Response table entries

The *TOpenSaveDialog* response table has no entries.

TOpenSaveDialog::TData struct

opensave.h

TOpenSaveDialog structure contains information about the user's file open or save selection. Specifically, this structure stores a user-specified file name filter, file extension, file name, the initial directory to use when displaying file names, any error codes, and various file attributes that determine, for example, if the file is a read-only file. The classes *TFileOpenDialog* and *TFileSaveDialog* use the information stored in this structure when a file is opened or saved.

Public constructors and destructor

Constructor

TData(uint32 flags=0, const char* filter=0, char* customFilter=0, char* initialDir=0, char* defExt=0);

Constructs a *TOpenSaveDialog::TData* structure.

Destructor

~TData();

Destructs a *TOpenSaveDialog::TData* structure.

See also TEditFile::FileData

Data members

CustomFilter

char* CustomFilter;

CustomFilter stores the user-specified file filter; for example, *.CPP.

DefExt

char* DefExt;

DefExt stores the default extension.

Error

uint32 Error;

Error contains one or more of the following error codes:

Constant	Meaning
CDERR_DIALOGFAILURE	Failed to create a dialog box.
CDERR_LOCKRESOURCEFAILURE	Failed to lock a specified resource.
CDERR_LOADRESFAILURE	Failed to load a specified resource.
CDERR_LOADSTRFAILURE	Failed to load a specified string.

FileName

char* FileName;

Holds the name of the file to be saved or opened.

Filter

char* Filter;

Filter holds the filter to use initially when displaying file names.

FilterIndex

int FilterIndex;

FilterIndex indicates which filter to use initially when displaying file names.

Flags

uint32 Flags;

Flag contains one or more of the following constants:

Constant	Meaning
OFN_HIDEREADONLY	Hides the read-only check box.
OFN_FILEMUSTEXIST	Lets the user enter only names of existing files in the File Name entry field. If an invalid file name is entered, a warning message is displayed.
OFN_PATHMUSTEXIST	Lets the user enter only valid path names. If an invalid path name is entered, a warning message is displayed.
OFN_NOVALIDATE	Performs no check of the file name and requires the owner of a derived class to perform validation.

Constant	Meaning
OFN_NOCHANGEDIR	Sets the current directory back to what it was when the dialog was initiated.
OFN_ALLOWMULTISELECT	Allows multiple selections in the File Name list box.
OFN_CREATEPROMPT	Asks if the user wants to create a file that does not currently exist.
OFN_EXTENSIONDIFFERENT	Idicates the user entered a file name different from the specified in *DefExt*. This message is returned to the caller.
OFN_NOREADONLYRETURN	The returned file does not have the Read Only attribute set and is not in a write-protected directory. This message is returned to the caller.
OFN_NOTESTFILECREATE	The file is created after the dialog box is closed. If the application sets this flag, there is no check against write protection, a full disk, an open drive door, or network protection. For certain network environments, this flag should be set.
OFN_OVERWRITEPROMPT	The Save As dialog box displays a message asking the user if it's OK to overwrite an existing file.
OFN_SHAREAWARE	If this flag is set and a call to open a file fails because of a sharing violation, the error is ignored and the dialog box returns the given file name. If this flag is not set, the virtual function *ShareViolation* is called, which returns OFN_SHAREWARN (by default) or one of the following values:
OFN_SHAREFALLTHROUGH	File name is returned from the dialog box. OFN_SHARENOWARN No further action is taken.
OFN_SHAREWARN	User receives the standard warning message for this type of error.
OFN_SHOWHELP	Shows the Help button in the dialog box.

InitialDir

```
char* InitialDir;
```

InitialDir holds the directory to use initially when displaying file names.

Public member functions

SetFilter

```
void SetFilter(const char*filter = 0);
```

Makes a copy of the filter list used to display the file names.

TOutStream class

docview.h

Derived from *TStream* and *ostream*, *TOutStream* is a base class used to create output storage streams for a document.

Public constructor

Constructor

```
TOutStream(TDocument& doc, const char far* name, int mode);
```

Constructs a *TOutStream* object. doc refers to the document object, *name* is the user-defined name of the stream, and *mode* is the mode of opening the stream.

See also TInStream, of XXXX document open enum, shxxx document sharing enum

TPaintDC class dc.h

A DC class derived from *TWindowDC* that wraps begin and end paint calls for use in a WM_PAINT response function.

Public constructor and destructor

Constructor
TPaintDC(HWND wnd);
Creates a *TPaintDC* object with the given owned window. The data member *Wnd* is set to *wnd*.

See also TWindowDC::Wnd, TDC::TDC

Destructor
~TPaintDC();
Destroys this object.

Public data member

Ps
PAINTSTRUCT Ps;
The paint structure associated with this *TPaintDC* object.

See also PAINTSTRUCT struct

Protected data member

Wnd
HWND Wnd
The associated window handle.

TPalette class gdiobjec.h

TPalette is the GDI Palette class derived from *TGdiObect*. The *TPalette* constructors can create palettes from explicit information or indirectly from various color table types that are used by DIBs.

Public constructors

Constructors
Form 1 TPalette(HPALETTE handle, TAutoDelete autoDelete = NoAutoDelete);
Creates a *TPalette* object and sets the *Handle* data member to the given borrowed *handle*. The *ShouldDelete* data member defaults to false, ensuring that the borrowed handle will not be deleted when the C++ object is destroyed.

Form 2 **TPalette(const TClipboard&);**
Creates a *TPalette* object with values taken from the given clipboard.

Form 3 **TPalette(const TPalette& palette);**
This public copy constructor creates a complete copy of the given *palette* object as in

```
TPalette myPalette = yourPalette;
```

Form 4 **TPalette(const LOGPALETTE far* logPalette);**
Creates a *TPalette* object from the given *logPalette* array.

Form 5 **TPalette(const PALETTEENTRY far* entries, int count);**
Creates a *TPalette* object with *count* entries from the given *entries* arrray.

Form 6 **TPalette(const BITMAPINFO far* info, uint flags = 0);**
Creates a *TPalette* object from the color table following the given BITMAPINFO structure. This constructor works only for 2-, 16-, and 256-color bitmaps. A 0 handle is returned for other bitmaps including 24-bit DIBs.

Form 7 **TPalette(const BITMAPCOREINFO far* core, uint flags = 0);**
Presentation Manager (PM) 1.x DIBs only. Creates a *TPalette* object from the color table following the given BITMAPCOREINFO structure. This constructor works only for 2-, 16-, and 256-color bitmaps. A 0 handle is returned for other bitmaps including 24-bit DIBs. Note that every color in a PM 1.x table must be present because there is no *ClrUsed* field in the DIB header.

Form 8 **TPalette(const TDib& dib, uint flags = 0);**
Creates a *TPalette* object from the given DIB object. The *flags* argument represents the values of the data structure used to create the palette.

Form 9 **TPalette();**
(Presentation Manager only) Creates a *TPalette* object that uses the default system palette .

See also *TClipboard::GetClipboardData*, TGdiObject::Handle, TGdiObject::ShouldDelete, TPalette::GetPaletteEntries, BITMAPCOREINFO struct, BITMAPINFO struct, LOGPALETTE struct, PALETTEENTRY struct

Public member functions

AnimatePalette

void AnimatePalette(uint start, uint count, const PALETTEENTRY far* entries);
Replaces entries in this logical palette from the *entries* array of PALETTEENTRY structures. *start* specifies the first entry to be animated, and *count* gives the number of entries to be animated. The new entries are mapped into the system palette immediately.

See also PALETTEENTRY struct

GetNearestPaletteIndex

uint GetNearestPaletteIndex(TColor color) const;
Returns the index of the color entry that represents the best color in this palette to the given *color*.

See also TColor

GetNumEntries

uint GetNumEntries() const;

Returns the number of entries in this palette or 0 if the call fails.

See also TGdiObject::GetObject

GetObject

bool GetObject(uint16 far& numEntries) const;

Finds the number of entries in this logical palette and sets the value in the *numEntries* argument. To find the entire LOGPALETTE structure, use *GetPaletteEntries*. Returns true if the call is successful; otherwise returns false.

See also TGdiObject::GetObject, TPalette::GetPaletteEntries, LOGPALETTE struct

GetPaletteEntries

uint GetPaletteEntries(uint16 start, uint16 count, PALETTEENTRY far* entries) const;

Retrieves a range of entries in this logical palette, and places them in the *entries* array. *start* specifies the first entry to be retrieved, and *count* gives the number of entries to be retrieved. Returns the number of entries actually retrieved, or 0 if the call fails.

See also PALETTEENTRY struct

GetPaletteEntry

uint GetPaletteEntry(uint16 index, PALETTEENTRY far& entry) const;

Retrieves the entry in this logical palette at *index*, and places it in the *entries* array. Returns the number of entries actually retrieved: 1 if successful or 0 if the call fails.

See also TPalette::SetPaletteEntry, PALETTEENTRY struct

operator <<

TClipboard& operator << (TClipboard& clipboard, TPalette& palette);

Copies the given *palette* to the given *clipboard* argument. Returns a reference to the resulting clipboard, which allows normal chaining of <<.

See also TClipboard

operator HPALETTE()

operator HPALETTE() const;

Typecasting operator. Converts this palette's *Handle* to type HPALETTE, which is the data type representing the handle to a logical palette.

ResizePalette

bool ResizePalette(uint numEntries);

Changes the size of this logical palette to the number given by *numEntries*. Returns true if the call is successful; otherwise returns false.

See also TPalette::AnimatePalette

SetPaletteEntries

uint SetPaletteEntries(uint16 start, uint16 count, const PALETTEENTRY far* entries);

Sets the RGB color values in this palette from the *entries* array of PALETTEENTRY structures. *start* specifies the first entry to be animated, and *count* gives the number of entries to be animated. Returns the number of entries actually set, or 0 if the call fails.

See also PALETTEENTRY struct

SetPaletteEntry

uint SetPaletteEntry(uint16 index, const PALETTEENTRY far& entry);

Sets the RGB color value at *index* in this palette from the *entry* argument. *start* specifies the first entry to be animated, and *count* gives the number of entries to be animated. Returns 1 (the number of entries actually set if successful) or 0 if the call fails.

See also PALETTEENTRY struct

ToClipboard

void ToClipboard(TClipboard& clipboard);

Moves this palette to the target clipboard argument. If a copy is to be put on the Clipboard, use *TPalette(myPalette).ToClipboard;* to make a copy first. The handle in the temporary copy of the object will be moved to the clipboard. *ToClipboard* sets *ShouldDelete* to false so that the object on the clipboard is not deleted. The handle will still be available for examination.

See also TClipBoard::SetClipBoardData

UnrealizeObject

bool UnrealizeObject();

Directs the GDI to completely remap the logical palette to the system palette on the next *RealizePalette(HDC)* or *TDC::RealizePalette* call. Returns true if the call is successful; otherwise false.

See also TDC::RealizePalette

Protected member functions

Create

void Create(const BITMAPINFO far* info, uint flags);
void Create(const BITMAPCOREINFO far* core, uint flags);

Sets values in this palette from the given bitmap structure. These functions are usually called by the constructor rather than directly.

See also TBITMAPCOREINFO struct, BITMAPINFO struct

TPaletteEntry class

color.h

TPaletteEntry is a support class derived from the structure *tagPALETTEENTRY*. The latter is defined as follows:

```
typedef struct tagPALETTEENTRY {
   uint8  peRed;
   uint8  peGreen;
   uint8  peBlue;
   uint8  peFlags;
} PALETTEENTRY;
```

where *peRed*, *peGreen*, and *peBlue* specify the red, green, and blue intensity-values for a palette entry.

The *peFlags* member can be set to NULL or one of the following values:

Value	Meaning
PC_EXPLICIT	Specifies that the low-order word of the logical palette entry designates a hardware palette index. This flag allows the application to show the contents of the display device palette.
PC_NOCOLLAPSE	Specifies that the color be placed in an unused entry in the system palette instead of being matched to an existing color in the system palette. If there are no unused entries in the system palette, the color is matched normally. Once this color is in the system palette, colors in other logical palettes can be matched to this color.
PC_RESERVED	Specifies that the logical palette entry be used for palette animation; this prevents other windows from matching colors to this palette entry since the color frequently changes. If an unused system-palette entry is available, this color is placed in that entry. Otherwise, the color is available for animation.

TPaletteEntry is used in conjunction with the classes *TPalette* and *TColor* to simplify logical color-palette operations. Constructors are provided to create *TPaletteEntry* objects from explicit COLORREF and RGB values, or from *TColor* objects.

Public constructors

Constructors

Form 1 `TPaletteEntry(int r, int g, int b, int f = 0);`

Creates a palette entry object with *peRed*, *peGreen*, *peBlue*, and *peFlags* set to *r*, *g*, *b*, and *f*, respectively.

Form 2 `TPaletteEntry(TColor c);`

Creates a palette entry object with *peRed*, *peGreen*, *peBlue*, and *peFlags* set to *r*, *g*, *b*, and *f*, respectively.

See also tagPALETTEENTRY struct, TPaletteEntry(TColorc), TColor::Red, TColor::Green, TColor::Blue

TPen class

gdiobjec.h

TPen is derived from *TGdiObject*. It encapsulates the GDI pen tool. Pens can be constructed from explicit information or indirectly. *TPen* relies on the base class's destructor, *~TGdiObject*.

Public constructors

Constructors

Form 1 `TPen(HPEN handle, TAutoDelete autoDelete = NoAutoDelete);`

Creates a *TPen* object and sets the *Handle* data member to the given borrowed *handle*. The *ShouldDelete* data member defaults to false, ensuring that the borrowed handle will not be deleted when the C++ object is destroyed.

Form 2 **TPen(TColor color, int width=1, int style=PS_SOLID);**
Creates a *TPen* object with the given values. The *width* argument is in device units, but if set to 0, a 1-pixel width is assumed. Sets *Handle* with the given default values. If *color* is black or white, *width* is one, *style* is solid, a stock pen handle is returned. The values for *style* are listed in the following table.

Value	Meaning
PS_SOLID	Creates a solid pen.
PS_DASH	Creates a dashed pen. Valid only when the pen width is one or less in device units.
PS_DOT	Creates a dotted pen. Valid only when the pen width is one or less in device units.
PS_DASHDOT	Creates a pen with alternating dashes dots. Valid only when the pen width is one or less in device units.
PS_DASHDOTDOT	Creates a pen with alternating dashes double-dots. Valid only when the pen width is one or less in device units.
PS_NULL	Creates a null pen.
PS_INSIDEFRAME	Creates a solid pen. When this pen is used in any GDI drawing function that takes a bounding rectangle, the dimensions of the figure will be shrunk so that it fits entirely in the bounding rectangle, taking into account the width of the pen.

Form 3 **TPen(const LOGPEN far* logPen);**
Creates a *TPen* object from the given *logPen* values.

Form 4 **TPen(const Tpen& pen);**
The *TPen* copy constructor.

Form 5 **TPen(uint32 penStyle, uint32 width, const TBrush& brush, uint32 styleCount, uint32* style);**
(32-bit) Creates a *TPen* object with the given values.

Form 6 **TPen(uint32 penStyle, uint32 width, const LOGBRUSH& logBrush, uint32 styleCount, uint32* style);**
(32-bit) Creates a *TPen* object with the given values.

See also TColor, TGdiObject::Handle, TGdiObject::ShouldDelete, LOGBRUSH struct, LOGPEN struct

Public member functions

GetObject
bool GetObject(LOGPEN far& logPen) const;
Retrieves information about this pen object and places it in the given *LOGPEN* structure. Returns true if the call is successful, otherwise false.

See also TGdiObject::GetObject, LOGPEN struct

operator HPEN()
operator HPEN() const;
Typecasting operator. Converts this pen's *Handle* to type *HPEN* (the data type representing the handle to a logical pen).

TPicResult enum — validate.h

enum TPicResult {prComplete, prIncomplete, prEmpty, prError, prSyntax, prAmbiguous, prIncompNoFill};
TPicResult is the result type returned by the *Picture* member function of *TPXPictureValidator*. The result type indicates whether the data entered into the edit control matches a specified format. For example, *prIncomplete* indicates that the data entered is missing some information that was specified in the format picture of the data.

See also
TPXPictureValidator::Picture

TPlacement enum — gadgetwi.h

enum TPlacement {Before, After};
Enumerates the placement of a gadget. The new gadget is inserted either before or after another gadget.

You can control the placement of the new gadget by specifying a *sibling* gadget that the new gadget is inserted before or after. If the sibling argument in *TGadgetWindow::Insert* is 0 then the new gadget is inserted at the beginning or the end of the existing gadgets. By default, the new gadget is inserted at the end of the existing gadgets.

See also TGadgetWindow::Insert

TPopupMenu class — menu.h

TPopupMenu creates an empty pop-up menu to add to an existing window or pop-up menu.

Public constructors

Constructors

Form 1 TPopupMenu(TAutoDelete autoDelete = AutoDelete);
Constructs an empty pop-up menu.

Form 2 TPopupMenu(HMENU handle, TAutoDelete autoDelete = NoAutoDelete);
Alias constructor for a pop-up menu.

Public member functions

TrackPopupMenu

Form 1 bool TrackPopupMenu(uint flags, int x, int y, int rsvd, HWND wnd, TRect* rect = 0);
Allows the application to create a pop-up menu at the specified location in the specified window. *flags* specifies a screen position and can be one of the TPM_*xxxx* values(TPM_CENTERALIGN, TPM_LEFTALIGN, TPM_RIGHTALIGN, TPM_LEFTBUTTON, or TPM_RIGHTBUTTON). *wnd* is the handle to the window that

receives messages about the menu. *x* specifies the horizontal position in screen coordinates of the left side of the menu. *y* species the vertical position in screen coordinates of the top of the menu (for example, 0,0 specifies that a menu's left corner is in the top left corner of the screen). *rect* defines the area that the user can click without dismissing the menu.

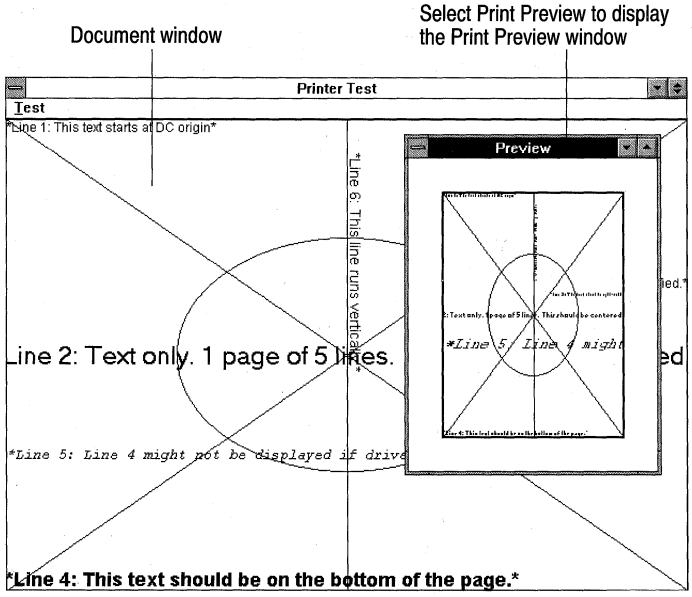
Form 2 **bool TrackPopupMenu(uint flags, TPoint& point, int rsvd, HWND wnd, TRect* rect = 0);**
This function is the same as the previous *TrackPopupMenu* except that the *x* and *y* positions are specified in *point*.

See also TPM_*xxxx* (Windows API)

TPreviewPage class

preview.h

TPreviewPage displays a page of a document in a print preview window. To obtain the information needed to display the page, *TPreviewPage* interacts with *TPrintPreviewDC* and *TPrintout*. Specifically, it creates a *TPrintPreviewDC* from the window DC provided in *Paint* and passes that *TPrintPreviewDC* object to *TPrintout*'s *SetPrintParams* member function. The sample program PRINT.CPP displays the following sample print preview window:



Public constructor

Constructor

TPreviewPage(TWindow* parent, TPrintout& printout, TPrintDC& prndc, TSize& printExtent, int pagenum = 1);
Constructs a *TPreviewPage* object where *parent* is the parent window, printout is a reference to the corresponding *TPrintout* object, *prndc* is a reference to the

TPrintPreviewDC object, *printExtent* is the extent (width and height) in logical units of the printed page, and *pagenum* is the number of the preview page. *TPreviewPage* has the attributes of a visible child window with a thin border. Sets the background color of the preview page window to white.

Public member functions

Paint

void Paint(TDC& dc, bool, TRect& clip);

Displays the page in the preview window. To determine the preview page's attributes (line width, and so on), *Paint* calls several of *TPrintout*'s member functions. Then, to adjust the printer object for previewing, *Paint* determines if the page fits in the preview window or if clipping is necessary. Finally, *Paint* passes clipping and banding information to *TPrintout*'s *PrintPage* function, which is called to display the page in the preview window.

See also TPrintout::BeginPrinting, TPrintout::EndPrinting, TPrintout::PrintPage

SetPageNumber

void SetPageNumber(int newNum);

Sets *newNum* to the number of the page currently displayed in the preview window.

Protected data members

PageNum

void SetPageNumber(int newNum);

Sets *newNum* to the number of the page currently displayed in the preview window.

PrintDC

TPrintDC& PrintDC;

PrintDC& is a handle to the device context to use for printing.

PrintExtent

TSize PrintExtent;

Contains the extent (width and height) in logical units of the page.

Printout

TPrintout& Printout;

Holds a reference to the *TPrintout* object.

Protected member functions

EvSize

void EvSize(uint sizeType, TSize& size);

Invalidates the entire window when the size of the page displayed in the preview window changes.

Response table entries

Response table entry	Member function
EV_WM_SIZE	EvSize

TPrintDC class dc.h

Derived from *TDC*, *TPrintDC* provides access to a printer.

Public constructors

Constructors

Form 1 `TPrintDC(HDC handle, TAutoDelete autoDelete = NoAutoDelete);`
Creates a *TPrint* object for the DC given by *handle*.

Form 2 `TPrintDC(const char far* driver, const char far* device, const char far* output, const DEVMODE far* initData);`
Creates a *TPrint* object given print driver, device, output, and data from the DEVMODE structure.

See also TDC, DEVMODE struct

Public member functions

AbortDoc

`int AbortDoc();`
Aborts the current print job on this printer and erases everything drawn since the last call to *StartDoc*. *AbortDoc* calls the user-defined function set with *TPrintDC::SetAbortProc* to abort a print job because of error or user intervention. *TPrintDC::EndDoc* should be used to terminate a successfully completed print job.

If successful, *AbortDoc* returns a positive or zero value; otherwise a negative value is returned.

See also TPrintDC::EndDoc, TPrintDC::SetAbortProc, TPrintDC::Escape

BandInfo

`int BandInfo(TBandInfo& bandInfo);`
Retrieves information about the banding capabilities of this device, and copies it to the given *bandInfo* structure. Returns 1 if the call is successful; returns 0 if the call fails or if this device does not support banding.

See also TBandInfo, TPrintDC::Escape

DeviceCapabilities

`static uint32 DeviceCapabilities(const char far* driver, const char far* device, const char far* port, int capability, char far* output=0, LPDEVMODE devmode=0);`
Retrieves data about the specified *capability* of the named printer *driver*, *device*, and *port*, and places the results in the *output* **char** array. The driver, device, and port names must be zero-terminated strings. The *devmode* argument points to a DEVMODE **struct**. If

devmode is 0 (the default), *DeviceCapabilities* retrieves the current default initialization values for the specified printer driver; otherwise, it retrieves the values contained in the DEVMODE structure. The format of the *output* array depends on the capability being queried. If *output* is 0 (the default), *DeviceCapabilities* returns the number of bytes required in the *output* array. Possible values for *capability* are as follows:

Value	Meaning

DC_BINNAMES	The function enumerates the paper bins on the given device. If a device driver supports this constant, the output array is a data structure that contains two members. The first member is an array identifying valid paper bins:
	If a device driver supports this constant, the output array is a data structure that contains two members. The first member is an array identifying valid paper bins:
	`short BinList[cBinMax]`
	The second member is an array of character strings specifying the bin names:
	`char PaperNames[cBinMax][cchBinName]`
	If a device driver does not support this value, the *output* array is empty and the return value is NULL.
	If *output* is NULL, the return value specifies the number of bins supported.

DC_BINS	The function retrieves a list of constants that identify the available bins and copies the list to the *output* array. If this array is NULL, the function returns the number of supported bins. The following bin identifiers can be returned: DMBIN_AUTO DMBIN_CASSETTE DMBIN_ENVELOPE DMBIN_ENVMANUAL DMBIN_FIRST DMBIN_LARGECAPACITY DMBIN_LARGEFMT DMBIN_LAST DMBIN_LOWER DMBIN_MANUAL DMBIN_MIDDLE DMBIN_ONLYONE DMBIN_SMALLFMT DMBIN_TRACTOR DMBIN_UPPER
DC_DRIVER	The function returns the driver version number.
DC_DUPLEX	The function returns the level of duplex support. The return value is 1 if the function supports duplex output; otherwise it is 0.
DC_ENUMRESOLUTIONS	The function copies a list of available printer resolutions to the *output* array. The resolutions are copied as pairs of int32 integers; the first value of the pair specifies the horizontal resolution and the second value specifies the vertical resolution. If *output* is 0, the function returns the number of supported resolutions.
DC_EXTRA	The function returns the number of bytes required for the device-specific data that is appended to the DEVMODE structure.

Value	Meaning
DC_FIELDS	The function returns a value indicating which members of the DEVMODE structure are set by the device driver. This value can be one or more of the following constants: DM_ORIENTATION DM_PAPERSIZE DM_PAPERLENGTH DM_PAPERWIDTH DM_SCALE DM_COPIES DM_DEFAULTSOURCE DM_PRINTQUALITY DM_COLOR DM_DUPLEX DM_YRESOLUTION DM_TTOPTION
DC_FILEDEPENDENCIES	The function returns a list of files that must be loaded when the device driver is installed. If *output* is 0 and this value is specified, the function returns the number of file names that must be loaded. If *output* is nonzero, the function returns the specified number of 64-character file names.
DC_MAXEXTENT	The function returns the maximum supported paper-size. These dimensions are returned in a POINT structure; the *x* member gives the maximum paper width and the *y* member gives the maximum paper length.
DC_MINEXTENT	The function returns the minimum supported paper-size. These dimensions are returned in a POINT structure; the *x* member gives the minimum paper width and the *y* member gives the minimum paper length.
DC_ORIENTATION	This function returns the number of degrees that a portrait-oriented paper is rotated counterclockwise to produce landscape orientation. if the return value is 0, there is no landscape orientation. If the return value is 90, the portrait-oriented paper is rotated 90 degrees (as is the case when HP laser printers are used). if the return value is 270, the portrait-oriented paper is rotated 270 degrees (as is the case when dot-matrix printers are used).
DC_PAPERNAMES	This function returns a list of supported paper names such as Letter size or Legal size. The *output* array points to an array containing the paper names. If the *output* array is 0, the function returns the number of available paper sizes.
DC_PAPERS	The function retrieves a list of supported paper sizes and copies it to the *output* array. The function returns the number of sizes identified in the array. If *output* is 0, the function returns the number of supported paper sizes.
DC_PAPERSIZE	The function retrieves the supported paper sizes (specified in .1 millimeter units) and copies them to the *output* array.
DC_SIZE	The function returns the size of the DEVMODE structure required by the given device driver.

Value	Meaning
DC_TRUETYPE	This function returns the printer driver's TrueType font capabilities. The values returned can be one or more of the following constants: DCTT_BITMAP — Device supports printing TrueType fonts as graphics. (Dot-matrix and PCL printers) DCTT_DOWNLOAD — Device supports downloading TryeType fonts. (PostScript and PCL printers) DCTT_SUBDEV — Device supports substituting device fonts for TrueType fonts. (PostScript printers)
DC_VERSION	The function returns the device driver version number.

If *DeviceCapabilities* succeeds, the return value depends on the value of *capability*, as noted above. Otherwise, the return value is GDI_ERROR.

See also TDC::GetDeviceCaps, DEVMODE struct

EndDoc

int EndDoc();

Ends the current print job on this printer. *EndDoc* should be called immediately after a successfully completed print job. *TPrintDC::AbortDoc* should be used to terminate a print job because of error or user intervention.

If successful, *EndDoc* returns a positive or zero value; otherwise a negative value is returned.

See also TPrintDC::StartDoc, TPrintDC::AbortDoc, TPrintDC::Escape

EndPage

int EndPage();

Tells this printer's device driver that the application has finished writing to a page. If successful, *EndPage* returns a positive or zero value; otherwise a negative value is returned. Possible failure values are listed below:

Value	Meaning
SP_ERROR	General error.
SP_APPABORT	Job terminated because the application's print-canceling function returned 0.
SP_USERABORT	User terminated the job.
SP_OUTOFDISK	Insufficient disk space for spooling.
SP_OUTOFMEMORY	Insufficient memory for spooling.

See also TPrintDC::StartPage, TPrintDC::Escape

Escape

int Escape(int escape, int count=0, const void* inData=0, void* outData=0);

Allows applications to access the capabilities of a particular device that are not directly available through the GDI of this DC. The *Escape* call is specified by setting a mnemonic value in the *escape* argument. In Win32 the use of *Escape* with certain *escape* values has

been replaced by specific functions. The names of these new functions are based on the corresponding *escape* mnemonic, as shown in the following table:

Value	Action
ABORTDOC	Superseded by *TPrintDC_AbortDoc* in Win32.
BANDINFO	Obsolete in Win32. Because all printer drivers for Windows version 3.1 and later set the text flag in every band, this escape is useful only for older printer drivers.
BEGIN_PATH	No changes for Win32. This escape is specific to PostScript printers.
CLIP_TO_PATH	No changes for Win32. This escape is specific to PostScript printers.
DEVICEDATA	Superseded in Win32. Applications should use the PASSTHROUGH escape to achieve the same functionality.
DRAFTMODE	Superseded in Win32. Applications can achieve the same functionality by setting the dmPrintQuality member of the DEVMODE structure to DMRES_DRAFT and passing this structure to the *CreateDC* function.
DRAWPATTERNRECT	No changes for Win32.
ENABLEDUPLEX	Superseded in Win32. Applications can achieve the same functionality by setting the *dmDuplex* member of the DEVMODE structure and passing this structure to the *CreateDC* function.
ENABLEPAIRKERNING	No changes for Win32.
ENABLERELATIVEWIDTHS	No changes for Win32.
ENDDOC	Superseded by *TPrintDC_EndDoc* in Win32.
END_PATH	No changes for Win32. This escape is specific to PostScript printers.
ENUMPAPERBINS	Superseded in Win32. Applications can use *TPrintDC::DeviceCapabilities* to achieve the same functionality.
ENUMPAPERMETRICS	Superseded in Win32. Applications can use *TPrintDC::DeviceCapabilities* to achieve the same functionality.
EPSPRINTING	No changes for Win32. This escape is specific to PostScript printers.
EXT_DEVICE_CAPS	Superseded in Win32. Applications can use *TDC::GetDeviceCaps* to achieve the same functionality. This escape is specific to PostScript printers.
EXTTEXTOUT	Superseded in Win32. Applications can use *TDC::ExtTextOut* to achieve the same functionality. This escape is not supported by the version 3.1 PCL driver.
FLUSHOUTPUT	Removed for Win32.
GETCOLORTABLE	Removed for Win32.
GETEXTENDEDTEXTMETRICS	No changes for Win32. Support for this escape might change in future versions of Windows.
GETEXTENTTABLE	Superseded in Win32. Applications can use *::GetCharWidth* to achieve the same functionality. This escape is not supported by the version 3.1 PCL or PostScript drivers.
GETFACENAME	No changes for Win32. This escape is specific to PostScript printers.
GETPAIRKERNTABLE	No changes for Win32.
GETPHYSPAGESIZE	No changes for Win32. Support for this escape might change in future versions of Windows.
GETPRINTINGOFFSET	No changes for Win32. Support for this escape might change in future versions of Windows.
GETSCALINGFACTOR	No changes for Win32. Support for this escape might change in future versions of Windows.

Value	Action
GETSETPAPERBINS	Superseded in Win32. Applications can achieve the same functionality by calling *TPrintDC::DeviceCapabilities* to find the number of paper bins, calling *::ExtDeviceMode* to find the current bin, and then setting the *dmDefaultSource* member of the DEVMODE structure and passing this structure to the *CreateDC* function. GETSETPAPERBINS changes the paper bin only for the current device context. A new device context will use the system-default paper bin until the bin is explicitly changed for that device context.
GETSETPAPERMETRICS	Obsolete in Win32. Applications can use *TPrintDC::DeviceCapabilities* and *::ExtDeviceMode* to achieve the same functionality.
GETSETPAPERORIENT	Obsolete in Win32. Applications can achieve the same functionality by setting the *dmOrientation* member of the DEVMODE structure and passing this structure to the *CreateDC* function. This escape is not supported by the Windows 3.1 PCL driver.
GETSETSCREENPARAMS	No changes for Win32.
GETTECHNOLOGY	No changes for Win32. Support for this escape might change in future versions of Windows. This escape is not supported by the Windows 3.1 PCL driver.
GETTRACKKERNTABLE	No changes for Win32.
GETVECTORBRUSHSIZE	No changes for Win32. Support for this escape might change in future versions of Windows.
GETVECTORPENSIZE	No changes for Win32. Support for this escape might change in future versions of Windows.
MFCOMMENT	No changes for Win32.
NEWFRAME	No changes for Win32. Applications should use *::StartPage* and *::EndPage* instead of this escape. Support for this escape might change in future versions of Windows.
NEXTBAND	No changes for Win32. Support for this escape might change in future versions of Windows.
PASSTHROUGH	No changes for Win32.
QUERYESCAPESUPPORT	No changes for Win32.
RESTORE_CTM	No changes for Win32. This escape is specific to PostScript printers.
SAVE_CTM	No changes for Win32. This escape is specific to PostScript printers.
SELECTPAPERSOURCE	Obsolete in Win32. Applications can achieve the same functionality by using *TPrintDC::DeviceCapabilities*.
SETABORTPROC	Superseded in Win32 by *::SetAbortProc*. See *TPrintDC::SetAbortProc*.
SETALLJUSTVALUES	No changes for Win32. Support for this escape might change in future versions of Windows. This escape is not supported by the Windows 3.1 PCL driver.
SET_ARC_DIRECTION	No changes for Win32. This escape is specific to PostScript printers.
SET_BACKGROUND_COLOR	No changes for Win32. Applications should use *::SetBkColor* instead of this escape. Support for this escape might change in future versions of Windows.
SET_BOUNDS	No changes for Win32. This escape is specific to PostScript printers.
SETCOLORTABLE	No changes for Win32. Support for this escape might change in future versions of Windows.
SETCOPYCOUNT	Superseded in Win32. An application should call *TPrintDC::DeviceCapabilities*, specifying DC_COPIES for the *capability* parameter, to find the maximum number of copies the device can make. Then the application can set the number of copies by passing to the *CreateDC* function a pointer to the DEVMODE structure.

Value	Action
SETKERNTRACK	No changes for Win32.
SETLINECAP	No changes for Win32. This escape is specific to PostScript printers.
SETLINEJOIN	No changes for Win32. This escape is specific to PostScript printers.
SETMITERLIMIT	No changes for Win32. This escape is specific to PostScript printers.
SET_POLY_MODE	No changes for Win32. This escape is specific to PostScript printers.
SET_SCREEN_ANGLE	No changes for Win32.
SET_SPREAD	No changes for Win32.
STARTDOC	Superseded in Win32. Applications should call *::StartDoc* instead of this escape.
TRANSFORM_CTM	No changes for Win32. This escape is specific to PostScript printers.

Escape calls are translated and sent to the printer device driver. The *inData* buffer lets you supply any data needed for the escape. You must set *count* to the size (in bytes) of the *inData* buffer. If no input data is required, *inData* and *count* should be set to the default value of 0. Similarly, you must supply an *outData* buffer for those *Escape* calls that retrieve data. If the escape does not supply output, set *outData* to the default value of 0.

NextBand

```
int NextBand(TRect& rect);
```

Tells this printer's device driver that the application has finished writing to a band. The device driver sends the completed band to the Print Manager and copies the coordinates of the next band in the rectangle specified by *rect*.

If successful, *NextBand* returns a positive or zero value; otherwise a negative value is returned. Possible failure values are listed below:

Value	Meaning
SP_ERROR	General error.
SP_APPABORT	Job terminated because the application's print-canceling function returned 0.
SP_USERABORT	User terminated the job.
SP_OUTOFDISK	Insufficient disk space for spooling.
SP_OUTOFMEMORY	Insufficient memory for spooling.

See also TPrintDC::Escape, TPrintDC::BandInfo

QueryAbort

```
bool QueryAbort(int rsvd=0);
```

16-bit applications only. Tries to call the *AbortProc* callback function for this printer to determine if a print job should be aborted or not. *QueryAbort* returns the value returned by *AbortProc* or true if no such callback function exists. true indicates that printing should continue; false indicates that the print job should be terminated. The *rsvd* argument is a reserved value that should be set to 0.

See also TPrintDC::SetAbortProc, TPrintDC::AbortDoc

QueryEscSupport

```
uint QueryEscSupport(int escapeNum);
```

Returns true if the escape specified by *escapeNum* is implemented on this device; otherwise false.

See also TPrintDC::Escape

SetAbortProc

int SetAbortProc(ABORTPROC proc);

Establishes the user-defined *proc* as the printer-abort function for this printer. This function is called by *TPrintDC::AbortDoc* to cancel a print job during spooling.

SetAbortProc returns a positive (nonzero) value if successful; otherwise it returns a negative (nonzero) value.

See also TPrintDC::Escape

SetCopyCount

int SetCopyCount(int reqestCount, int& actualCount);

Sets *requestCount* to the number of uncollated copies of each page that this printer should print. The actual number of copies to be printed is copied to *actualCount*. The actual count will be less than the requested count if the latter exceeds the maximum allowed for this device. *SetCopyCount* returns 1 if successful; otherwise, it returns 0.

See also TPrintDC::DeviceCapabilities, TPrintDC::Escape

StartDoc

int StartDoc(const char far* docName, const char far* output);

Starts a print job for the named document on this printer DC. If successful, *StartDoc* returns a positive value, the job ID for the document. If the call fails, the value SP_ERROR is returned. Detailed error information can be obtained by calling *GetLastError*.

This function replaces the earlier *::Escape* call with value STARTDOC.

See also TPrintDC::EndDoc, TPrintDC::Escape

StartPage

int StartPage();

Prepares this device to accept data. The system disables *::ResetDC* between calls to *StartPage* and *EndPage*, so that applications cannot change the device mode except at page boundaries. If successful, *StartPage* returns a positive value; otherwise, a negative or zero value is returned.

See also TPrintDC::EndPage

Protected data member

DocInfo

DOCINFO DocInfo;

Holds the input and output file names used by *TPrintDC::StartDoc*. The DOCINFO structure is defined as follows:

```
typedef struct {
   int cbSize;          // size of the structure, bytes
```

```
    DocInfo lpszDocName;   // document name <= 32 chars inc. final 0
    DocInfo lpszOutput;    // output file name
} DOCINFO;
```

The *lpszOutput* field allows a print job to be redirected to a file. If this field is NULL, the output will go to the device for the specified DC.

See also TPrintDC::StartDoc

TPrintDialog::TData struct

printdia.h

TPrintDialog::TData contains information required to initialize the printer dialog box with the user's print selections. This information consists of the number of copies to be printed, the first and last pages to print, the maximum and minimum number of pages that can be printed and various flag values that indicate whether the Pages radio button is displayed, the Print to File check box is enabled, and so on. *TPrintDialog* uses this struct to initialize the print dialog box. Whenever the user changes the print requirements, this struct is updated.

If an error occurs, *TPrintDialog::TData* returns one of the common dialog extended error codes. *TPrintDialog::TData* also takes care of locking and unlocking memory associated with the DEVMODE and DEVNAMES structures, which contain information about the printer driver, the printer, and the output printer port.

TPrinter has access to this information through its data member, *Data.*

See also TPrintDialog, TPrinter

Public data members

Copies

int Copies;

Copies indicates the actual number of pages to be printed.

Error

uint32 Error;

If the dialog box is successfully executed, *Error* returns 0. Otherwise, it contains one of the following error codes.

Constant	Meaning
CDERR_DIALOGFAILURE	Failed to create a dialog box.
CDERR_FINDRESFAILURE	Failed to find a specified resource.
CDERR_INITIALIZATION	Failed to initialize the common dialog box function. A lack of sufficient memory can generate this error.
CDERR_LOCKRESOURCEFAILURE	Failed to lock a specified resource.
CDERR_LOADRESFAILURE	Failed to load a specified resource.
CDERR_LOADSTRFAILURE	Failed to load a specified string.
CDERR_MEMALLOCFAILURE	Unable to allocate memory for internal data structures.

Constant	Meaning
CDERR_MEMLOCKFAILURE	Unable to lock the memory associated with a handle.
CDERR_REGISTERMSGFAIL	A message, designed for the purpose of communicating between two applications, could not be registered.
PDERR_CREATEICFAILURE	*TPrintDialog* failed to create an information context.
PDERR_DEFAULTDIFFERENT	The printer described by structure members doesn't match the default printer. This error message can occur if the user changes the printer specified in the control panel.
PDERR_DNDMMISMATCH	The printer specified in *DevMode* and in *DevNames* is different.
PDERR_GETDEVMODEFAIL	The printer device-driver failed to initialize the *DevMode* structure.
PDERR_INITFAILURE	The *TPrintDialog* structure could not be initialized.
PDERR_LOADDRVFAILURE	The specified printer's device driver could not be loaded.
PDERR_NODEFAULTPRN	A default printer could not be identified.
PDERR_NODEVICES	No printer drivers exist.
PDERR_PARSEFAILURE	The string in the [devices] section of the WIN.INI file could not be parsed.
PDERR_PRINTERNOTFOUND	The [devices] section of the WIN.INI file doesn't contain the specified printer.
PDERR_RETDEFFAILURE	Either *DevMode* or *DevNames* contain zero.
PDERR_SETUPFAILURE	*TPrintDialog* failed to load the required resources.

Flags

uint32 Flags;

Flags, which are used to initialize the printer dialog box, can be one or more of the following values that control the appearance and functionality of the dialog box:

Constant	Meaning
PD_ALLPAGES	Indicates that the All radio button was selected when the user closed the dialog box.
PD_COLLATE	Causes the Collate checkbox to be checked when the dialog box is created.
PD_DISABLEPRINTTOFILE	Disables the Print to File check box.
PD_HIDEPRINTTOFILE	Hides and disables the Print to File check box.
PD_NOPAGENUMS	Disables the Pages radio button and the associated edit control.
PD_NOSELECTION	Disables the Selection radio button.
PD_NOWARNING	Prevents the warning message from being displayed when there is no default printer.
PD_PAGENUMS	Selects the Pages radio button when the dialog box is created.
PD_PRINTSETUP	Displays the Print Setup dialog box rather than the Print dialog box.
PD_PRINTTOFILE	Checks the Print to File check box when the dialog box is created.

Constant	Meaning
PD_RETURNDC	Returns a device context matching the selections that the user made in the dialog box.
PD_RETURNDEFAULT	Returns *DevNames* structures that are initialized for the default printer without displaying a dialog box.
PD_RETURNIC	Returns an information context matching the selections that the user made in the dialog box.
PD_SELECTION	Selects the Selection radio button when the dialog box is created.
PD_SHOWHELP	Shows the Help button in the dialog box.
PD_USEDEVMODECOPIES	If a printer driver supports multiple copies, setting this flag causes the requested number of copies to be stored in the *dmCopies* member of the *DevMode* structure and 1 in *Copies*. If a printer driver does not support multiple copies, setting this flag disables the *Copies* edit control. If this flag is not set, the number 1 is stored in *DevMode* and the requested number of copies in *Copies*.

FromPage

```
int FromPage;
```

FromPage indicates the beginning page to print.

See also TPrintDialog::TData::ToPage

MaxPage

```
int MaxPage;
```

MaxPage indicates the maximum number of pages that can be printed.

MinPage

```
int MinPage;
```

MinPage indicates the minimum number of pages that can be printed.

ToPage

```
int ToPage;
```

ToPage indicates the ending page to print.

See also TPrintDialog::TData::FromPage

Public member functions

ClearDevMode

```
void ClearDevMode();
```

Clears device mode information (information necessary to initialize the dialog controls).

ClearDevNames

```
void ClearDevNames();
```

Clears the device name information (information that contains three strings used to specify the driver name, the printer name, and the output port name).

GetDeviceName

```
const char far* GetDeviceName() const;
```

Gets the name of the output device.

GetDevMode
const DEVMODE far* GetDevMode() const;
Gets a pointer to a DEVMODE structure (a structure containing information necessary to initialize the dialog controls).

GetDevNames
const DEVNAMES far* GetDevNames() const;
Gets a pointer to a DEVNAMES structure (a structure containing three strings used to specify the driver name, the printer name, and the output port name).

GetDriverName
const char far* GetDriverName() const;
Gets the name of the printer device driver.

GetOutputName
const char far* GetOutputName() const;
Gets the name of the physical output medium.

Lock
void Lock();
Locks memory associated with the DEVMODE and DEVNAMES structures.

SetDevMode
void SetDevMode(const DEVMODE far* devMode);
Sets the values for the DEVMODE structure.

SetDevNames
void SetDevNames(const char far* driver, const char far* device, const char far* output);
Sets the values for the DEVNAMES structure.

TransferDC
TPrintDC* TransferDC();
Creates and returns a *TPrintDC* with the current settings.

Unlock
void Unlock();
Unlocks memory associated with the DEVMODE and DEVNAMES structures.

TPrintDialog class

printdia.h

TPrintDialog displays either a modal print or a print setup dialog box. The print dialog box lets you specify the characteristics of a particular print job. The setup dialog box lets you configure the printer and specify additional print job characteristics. You can also use *TPrinter* and *TPrintout* to provide support for printer dialog boxes. *TPrintDialog* uses the *TPrintDialog::TData* struct to initialize the dialog box with the user's printer options, such as the number of pages to print, the output device, and so on.

See also
TPrintDialog::TData struct, TPrinter, TPrintout

Public constructor

Constructor
```
TPrintDialog(TWindow* parent, TData& data, const char far* printTemplateName=0,
  const char far* setupTemplateName=0, const char far* title=0, TModule* module=0);
```
Constructs a print or print setup dialog box with specified data from the *TPrintDialog::TData* structure, parent window, window caption, print and setup templates, and module.

See also TPrintDialog::TData struct

Public member functions

DoExecute
```
int DoExecute();
```
If no error occurs, *DoExecute* copies flags and print specifications into the *data* argument in the constructor. If an error occurs, *DoExecute* sets the error number of *data* to an error code from *TPrintDialog::TData::Error* .

See also TPrintDialog::TData::Error

GetDefaultPrinter
```
bool GetDefaultPrinter();
```
Without displaying a dialog box, *GetDefaultPrinter* gets the device mode and name that are initialized for the system default printer.

Protected data members

Data
```
TData& Data;
```
Data is a reference to the *TData* object passed in the constructor. The *TData* object contains print specifications such as the number of copies to be printed, the number of pages, the ouput device name, and so on.

See also TPrintDialog::TData struct

pd
```
PRINTDLG pd;
```
Specifies the dialog box print job characteristics such as page range, number of copies, device context, and so on necessary to initialize the print or print setup dialog box.

See also TPrintDialog::TData struct

Protected member functions

CmSetup
void CmSetup();
Responds to the click of the setup button with an EV_COMMAND message.

DialogFunction
bool DialogFunction(uint message, WPARAM, LPARAM);
Returns **true** if a message is handled.

See also TDialog::DialogFunction

Response table entries

The *TPrintDialog* response table has no entries.

TPrinter class

printer.h

TPrinter represents the physical printer device. To print or configure a printer, initialize an instance of *TPrinter*.

Public constructor and destructor

Constructor
TPrinter();
Constructs an instance of *TPrinter* associated with the default printer. To change the printer, call *SetDevice* after the object has been initialized or call *Setup* to let the user select the new device through a dialog box.

Destructor
virtual ~TPrinter();
Frees the resources allocated to *TPrinter*.

Public member functions

ClearDevice
virtual void ClearDevice();
Called by *SetPrinter* and the Destructor, *ClearDevice* disassociates the device with the current printer. *ClearDevice* changes the current status of the printer to PF_UNASSOCIATED, which causes the object to ignore all calls to *Print* until the object is reassociated with a printer.

GetSetup
TPrintDialog::TData& GetSetup();
Returns a reference to the *TPrintDialog* data structure.

GetUserAbort

```
static bool GetUserAbort();
```

Returns **true** if the user has chosen to stop printing through the printing dialog. Returns **false** otherwise.

Print

```
virtual bool Print(TWindow* parent, TPrintout& printout, bool prompt);
```

Print renders the given printout object on the associated printer device and displays an Abort dialog box while printing. It displays any errors encountered during printing. Prompt allows you to show the user a window.

See also TPrinter::Error

ReportError

```
virtual void ReportError(TWindow* parent, TPrintout& printout);
```

Print calls *ReportError* if it encounters an error. By default, it brings up the system message box with an error string created from the default string table. This function can be overridden to show a custom error dialog box.

Setup

```
virtual void Setup(TWindow* parent);
```

Setup lets the user select and/or configure the currently associated printer. *Setup* opens a dialog box as a child of the given window. The user then selects one of the buttons in the dialog box to select or configure the printer. The form of the dialog box is based on *TPrintDialog*, the common dialog printer class.

SetUserAbort

```
static void SetUserAbort(bool abort=true);
```

Sets the printing abort flag.

Protected data members

BandRect

```
TRect BandRect;
```

BandRect specifies the size of the banding rectangle.

Data

```
TPrintDialog::TData* Data;
```

Data is a pointer to the *TPrintDialog* data structure that contains information about the user's print selection.

See also TPrintDialog::TData struct

Error

```
int Error;
```

Error is the error code returned by GDI during printing. This value is initialized during a call to *Print*.

FirstBand

```
bool FirstBand;
```

FirstBand is set to **true** if the first band of the print job is being printed, otherwise **false**.

Flags
unsigned Flags;
The *Flags* data member specifies whether the printout bands contain graphics bands, text bands, or both. The valid flag values are enumerated by *TPrintoutFlags*:

```
enum TPrintoutFlags {
   pfGraphics // Current band accepts graphics
   pfText     // Current band accepts text
   pfBoth     // Current band accepts either graphics or text
};
```

PageSize
TSize PageSize;
PageSize specifies the size of the printed page, as specifed in the device context.

UseBandInfo
bool UseBandInfo;
UseBandInfo is set to true if the printer supports banding, otherwise it's set to false.

Protected member functions

CalcBandingFlags
void CalcBandingFlags(TPrintDC& prnDC);
CalcBandingFlags determines if there are either text and graphics bands, and sets data member *Flags* accordingly.

CreateAbortWindow
virtual TWindow* CreateAbortWindow(TWindow* parent, TPrintout& printout);
Creates a printer abort dialog message box.

ExecPrintDialog
virtual bool ExecPrintDialog(TWindow* parent);
Executes a *TPrintDialog*.

GetDefaultPrinter
virtual void GetDefaultPrinter();
Updates the *printer* structure with information about the user's default printer.

SetPrinter
virtual void SetPrinter(const char* driver, const char* device, const char* output);
SetPrinter changes the printer device association. *Setup* calls *SetPrinter* to change the association interactively. The valid parameters to this method can be found in the [devices] section of the WIN.INI file.

Entries in the [devices] section have the following format:

```
<device name>=<driver>, <port> {, <port>}
```

TPrinter::TXPrinter class

printer.h

A nested class, *TXPrinter* describes an exception that results from an invalid printer object. This type of error can occur when printing to the physical printer.

Public constructors

Constructors
TXPrinter(uint resId = IDS_PRINTERERROR);
Constructs a *TXPrinter* object with a default IDS_PRINTERERROR message.

TPrinterAbortDlg class

printer.h

TPrinterAbortDlg is the object type of the default printer-abort dialog box. This dialog box is initialized to display the title of the current printout, as well as the device and port currently used for printing.

TPrinterAbortDlg expects to have three static text controls, with control IDs of 101 for the title, 102 for the device, and 103 for the port. These controls must have "%s" somewhere in the text strings so that they can be replaced by the title, device, and port. The dialog-box controls can be in any position and tab order.

Public constructor

Constructor
TPrinterAbortDlg(TWindow* parent, TResId resId, const char far* title,
  const char far* device, const char far* port);
Constructs an Abort dialog box that contains a Cancel button and displays the given title, device, and port.

Protected member functions

EvCommand
virtual LRESULT EvCommand(uint id, HWND hWndCtl, uint notifyCode);
Handles the Cancel button on the Printer-abort dialog box.

SetupWindow
void SetupWindow();
Associates objects with the dialog resource template so that the title, device, and port can be determined for printing. See the description of *TPrintoutFlags* for information about printing flags and printer status information.

See also TPrintoutFlags enum

TPrintout class

printer.h

TPrintout represents the physical printed document that is to sent to a printer to be printed. *TPrintout* does the rendering of the document onto the printer. Because this object type is abstract, it cannot be used to print anything by itself. For every document, or document type, a class derived from *TPrintout* must be created and its *PrintPage* function must be overridden.

Public constructor and destructor

Constructor

TPrintout(const char far* title);

Constructs an instance of *TPrintOut* with the given title.

Destructor

virtual ~TPrintout();

Destroys the resources allocated by the constructor.

Public member functions

BeginDocument

virtual void BeginDocument(int startPage, int endPage, unsigned flags);

The printer object's *Print* function calls *BeginDocument* once before printing each copy of a document. The *flags* field indicates if the current print band accepts graphics, text, or both.

The default *BeginDocument* does nothing. Derived objects can override *BeginDocument* to perform any initialization needed at the beginning of each copy of the document.

See also TPrintoutFlags enum

BeginPrinting

virtual void BeginPrinting();

The printer object's *Print* function calls *BeginPrinting* once at the beginning of a print job, regardless of how many copies of the document are to be printed. Derived objects can override *BeginPrinting* to perform any initialization needed before printing.

EndDocument

virtual void EndDocument();

The printer object's *Print* function calls *EndDocument* after each copy of the document finishes printing. Derived objects can override *EndDocument* to perform any needed actions at the end of each document.

EndPrinting

virtual void EndPrinting();

The printer object's *Print* function calls *EndPrinting* after all copies of the document finish printing. Derived objects can override *EndPrinting* to perform any needed actions at the end of each document.

GetDialogInfo

```
virtual void GetDialogInfo(int& minPage, int& maxPage, int& selFromPage, int& selToPage);
```

Retrieves information needed to allow the printing of selected pages of the document and returns **true** if page selection is possible. Use of page ranges is optional, but if the page count is easy to determine, *GetDialogInfo* sets the number of pages in the document. Otherwise, it sets the number of pages to 0 and printing will continue until *HasPage* returns **false**.

GetTitle

```
const char far* GetTitle() const;
```

Returns the title of the current printout.

HasPage

```
virtual bool HasPage(int pageNumber);
```

HasPage is called after every page is printed. By default, it returns **false**, indicating that only one page is to be printed. If the document contains more than one page, this function must be overridden to return **true** while there are more pages to print.

PrintPage

```
virtual void PrintPage(int page, TRect& rect, unsigned flags);
```

PrintPage is called for every page (or band, if *Banding* is **true**) and must be overridden to print the contents of the given page. The *rect* and *flags* parameters are used during banding to indicate the extent and type of band currently requested from the driver (and should be ignored if *Banding* is false). *page* is the number of the current page.

SetPrintParams

```
virtual void SetPrintParams(TPrintDC* dc, TSize pageSize);
```

SetPrintParams sets DC to *dc* and *PageSize* to *pageSize*. The printer object's *Print* function calls *SetPrintParams* to obtain the information it needs to determine pagination and page count. Derived objects that override *SetPrintParams* must call the inherited function.

See also TPreviewPage::Paint

WantBanding

```
bool WantBanding() const;
```

Returns the value of data member *Banding*.

WantForceAllBands

```
bool WantForceAllBands() const;
```

Returns the value of data member *ForceAllBands*.

Type definitions — printer.h

TPrintoutFlags enum

```
enum{pfGraphics, pfText, pfBoth};
```

ObjectWindows defines the following banding constants used to set flags for printout objects.

Constant	Meaning
pfGraphics	Current band accepts only graphics.
pfText	Current band accepts only text.
pfBoth	Current band accepts both text and graphics.

See also TPrinter, TPrintOut

Protected data members

Banding
bool Banding;
If *Banding* is **true**, the printout is banded and the *PrintPage* function is called once for every band. Otherwise, *PrintPage* is called only once for every page. Banding a printout is more memory- and time-efficient than not banding. By default, *Banding* is set to **false**.

DC
TPrintDC* DC;
DC is the handle to the device context to use for printing.

ForceAllBands
bool ForceAllBands;
Many device drivers do not provide all printer bands if both text and graphics are not performed on the first band (which is typically a text-only band). Leaving *ForceAllBands* **true** forces the printer driver to provide all bands regardless of what calls are made in the *PrintPage* function. If *PrintPage* does nothing but display text, it is more efficient for *ForceAllBands* to be **false**. By default, it is true. *ForceAllBands* takes effect only if *Banding* is **true**.

PageSize
TSize PageSize;
PageSize is the size of the print area on the printout page.

Title
const char far* GetTitle() const;
Returns the title of the current printout.

TPrintPreviewDC class

preview.h

Derived from *TPrintDC*, *TPrintPreviewDC* maps printer device coordinates to logical screen coordinates. It sets the extent of the view window and determines the screen and printer font attributes. Many of *TPrintPreviewDC's* functions override *TDC's* virtual functions.

Public constructor and destructor

Constructor

TPrintPreviewDC(TDC& screen, TPrintDC& printdc, const TRect& client, const TRect& clip);

TPrintPreviewDC's constructor takes a screen DC as well as a printer DC. The screen DC is passed to the inherited constructor while the printer DC is copied to the member, *PrnDC*.

Destructor

~TPrintPreviewDC();

Destroys a *TPrintPreviewDC* object.

Public member functions

GetDeviceCaps

int GetDeviceCaps(int index) const;

GetDeviceCaps returns capability information, such as font and pitch attributes, about the printer DC. The *index* argument specifies the type of information required.

See also TDC::GetDeviceCaps

LPtoSDP

Form 1 bool LPtoSDP(TPoint* points, int count = 1) const;

Converts each of the *count* points in the *points* array from logical points of the printer DC to screen points. Returns a nonzero value if the call is successful; otherwise, returns 0.

Form 2 bool LPtoSDP(TRect& rect) const;

Converts each of the points in the *rect* from logical points of the printer DC to screen device points. Returns a nonzero value if the call is successful; otherwise, returns 0.

See also TPrintPreviewDC::SDPtoLP, TDC::LPtoDP

OffsetViewportOrg

bool OffsetViewportOrg(const TPoint& delta, TPoint far* oldOrg = 0);

Modifies this DC's viewport origin relative to the current values. The *delta* x- and y-components are added to the previous origin and the resulting point becomes the new viewport origin. The previous origin is saved in *oldOrg*. Returns nonzero if the call is successful; otherwise, returns 0.

See also TPrintPreviewDC::SetViewportOrg, TDC::OffsetViewportOrg

ReOrg

virtual void ReOrg();

Gets the x- and y- extents of the viewport, equalizes the logical and screen points, and resets the x- and y- extents of the viewport.

ReScale

virtual void ReScale();

Maps the points of the printer DC to the screen DC. Sets the screen window extent equal to the maximum logical pointer of the printer DC.

RestoreFont

void RestoreFont();

Restores the original GDI font object to this DC.

See also TPrintPreviewDC::SelectObject, TDC::OrgFont

ScaleViewportExt

bool ScaleViewportExt(int xNum, int xDenom, int yNum, int yDenom, TSize far* oldExtent = 0);

Modifies this DC's viewport extents relative to the current values. The new extents are derived as follows:

```
xNewVE = (xOldVE * xNum) / xDenom
yNewVE = (yOldVE * yNum) / yDenom
```

The previous extents are saved in *oldExtent*. Returns nonzero if the call is successful; otherwise returns 0.

See also TDC::ScaleViewportExt, TPrintPreviewDC::SetViewportExt

ScaleWindowExt

bool ScaleWindowExt(int xNum, int xDenom, int yNum, int yDenom, TSize far* oldExtent = 0);

Modifies this DC's window extents relative to the current values. The new extents are derived as follows:

```
xNewWE = (xOldWE * xNum) / xDenom
yNewWE = (yOldWE * yNum) / yDenom
```

The previous extents are saved in *oldExtent*. Returns nonzero if the call is successful; otherwise returns 0.

See also TDC::SetWindowExt, TPrintPreviewDC::ScaleWindowExt

SDPtoLP

Form 1 bool SDPtoLP(TPoint* points, int count = 1) const;

Converts each of the *count* points in the *points* array from screen device points to logical points of the printer DC. Returns a nonzero value if the call is successful; otherwise, returns 0.

Form 2 bool SDPtoLP(TRect& rect) const;

Converts each of the points in the *rect* from screen device points to logical points of the printer DC. Returns a nonzero value if the call is successful; otherwise, returns 0.

See also TPrintPreviewDC::LPtoSDP, TDC::DPtoLP

SelectObject

void SelectObject(const TFont& newFont);

Selects the given font object into this DC.

See also TPrintPreviewDC::SelectStockObject, TDC::SelectObject

SelectStockObject

void SelectStockObject(int index);

Retrieves a handle to a predefined stock font.

See also TDC::SelectStockObject

SetBkColor

TColor SetBkColor(TColor color);

Sets the current background color of this DC to the given *color* value or the nearest available. Returns 0x80000000 if the call fails.

See also TDC::SetBkColor

SetMapMode

int SetMapMode(int mode);

Sets the current window mapping mode of this DC to *mode*. Returns the previous mapping mode value. The mapping mode defines how logical coordinates are mapped to device coordinates. It also controls the orientation of the device's x- and y-axes.

See also TDC::GetMapMode, TDC::SetMapMode

SetTextColor

TColor SetTextColor(TColor color);

Sets the current text color of this DC to the given *color* value. The text color determines the color displayed by *TDC::TextOut* and *TDC::ExtTextOut*.

See also TDC::GetTextColor, TDC::SetTextColor

SetViewportExt

bool SetViewportExt(const TSize& extent, TSize far* oldExtent = 0);

Sets the screen's viewport x- and y-extents to the given *extent* values. The previous extents are saved in *oldExtent*. Returns nonzero if the call is successful; otherwise, returns 0. The *extent* value determines the amount of stretching or compression needed in the logical coordinate system to fit the device coordinate system. *extent* also determines the relative orientation of the two coordinate systems.

See also TDC::GetViewportExt, TDC::SetViewportExt

SetViewportOrg

bool OffsetViewportOrg(const TPoint& delta, TPoint far* oldOrg = 0);

Modifies this DC's viewport origin relative to the current values. The *delta* x- and y-components are added to the previous origin and the resulting point becomes the new viewport origin. The previous origin is saved in *oldOrg*. Returns nonzero if the call is successful; otherwise, returns 0.

See also TPrintPreviewDC::SetViewportOrg, TDC::OffsetViewportOrg

SetWindowExt

bool SetWindowExt(const TSize& extent, TSize far* oldExtent=0);

Sets the DC's window x- and y-extents to the given *extent* values. The previous extents are saved in *oldExtent*. Returns nonzero if the call is successful; otherwise, returns 0. The *extent* value determines the amount of stretching or compression needed in the logical coordinate system to fit the device coordinate system. *extent* also determines the relative orientation of the two coordinate systems.

See also TDC::GetWindowExt, TDC::SetWindowExt, TPrintPreviewDC::ScaleWindowExt

SyncFont
virtual void SyncFont();
Sets the screen font equal to the current printer font.

Protected data members

CurrentPreviewFont
TFont* CurrentPreviewFont;
The current view font.

PrnDC
TPrintDC& PrnDC;
Holds a reference to the printer DC.

PrnFont
HFONT PrnFont;
The current printer font.

Protected member functions

GetAttributeHDC
HDC GetAttributeHDC() const;
Returns the attributes of the printer DC (*PrnDC*).

See also TDC::GetAttributeHDC

TProfile class

profile.h

An instance of *TProfile* encapsulates a setting within a system file, often referred to as a *profile* or *initialization* file. Examples of this type of file include the Windows initialization files SYSTEM.INI and WIN.INI. Within the system file itself, the individual settings are grouped within sections. For example,

```
[Diagnostics]   ; section name
Enabled=0        ; setting
```

For a setting, the value to the left of the equal sign is called the *key*. The value to the right of the equal sign, the *value*, can be either an integer or a string data type.

Public constructor and destructor

Constructor
TProfile(const char* section, const char* filename=0);
Constructs a *TProfile* object for the indicated *section* within the profile file specified by *filename*. If the file name is not provided, the file defaults to the system profile file; for example, WIN.INI under Windows.

Destructor

~TProfile();

Destroys the *TProfile* object.

Public member functions

GetInt

int GetInt(const char* key, int defaultInt = 0);

Looks up and returns the integer value associated with the given string, *key*. If *key* is not found, the default value, *defaultInt*, is returned.

GetString

bool GetString(const char* key, char buff[], unsigned buffSize, const char* defaultString = "");

Looks up and returns the string value associated with the given *key* string. The string value is copied into *buff*, up to *buffSize* bytes. If the key is not found, *defaultString* provides the default value. If a 0 key is passed, all section values are returned in *buff*.

WriteInt

bool WriteInt(const char* key, const char* int value);

Looks up the key and replaces its value with the integer value passed (*int*). If the key is not found, *WriteInt* makes a new entry. Returns **true** if successful.

WriteString

bool WriteString(const char* key, const char* str);

Looks up the key and replaces its value with the string value passed (*str*). If the key is not found, *WriteString* makes a new entry. Returns **true** if successful.

TPXPictureValidator class

validate.h

TPXPictureValidator objects compare user input with a picture of a data format to determine the validity of entered data. The pictures are compatible with the pictures Borland's Paradox relational database uses to control data entry. For a complete description of picture specifiers, see the *Picture* member function.

Public constructor

Constructor

TPXPictureValidator(const char far* pic, bool autoFill=false);

Constructs a picture validator object by first calling the constructor inherited from *TValidator* and setting *pic* to point to it. Then sets the *voFill* bit in *Options* if *AutoFill* is true and sets *Options* to *voOnAppend*. Throws a *TXValidator* exception if the picture is invalid.

Public member functions

Error

void Error();

Overrides *TValidator's* virtual function and displays a message box that indicates an error in the picture format and displays the string pointed to by *Pic*.

See also TValidator::Error

IsValid

```
bool IsValid(const char far* str);
```

IsValid overrides *TValidator's* virtual function and compares the string passed in *str* with the format picture specified in *Pic*. *IsValid* returns true if *Pic* is NULL or if *Picture* returns *Complete* for *str*, indicating that *str* needs no further input to meet the specified format; otherwise, it returns false.

See also TPXPictureValidator::Picture

IsValidInput

```
bool IsValidInput(char far* str, bool suppressFill);
```

IsValidInput overrides *TValidator's* virtual function and checks the string passed in *str* against the format picture specified in *Pic*. *IsValid* returns **true** if *Pic* is NULL or *Picture* does not return *Error* for *str*; otherwise, it returns **false**. The *suppressFill* parameter overrides the value in *voFill* for the duration of the call to *IsValidInput*.

If *suppressFill* is **false** and *voFill* is set, the call to *Picture* returns a filled string based on *str*, so the image in the edit control automatically reflects the format specified in *Pic*.

See also TPXPictureValidator::Picture

Picture

```
virtual TPicResult Picture(char far* input, bool autoFill=false);
```

Formats the string passed in *input* according to the format specified by the picture string pointed to by *Pic*. *Picture* returns *prError* if there is an error in the picture string or if *input* contains data that cannot fit the specified picture. Returns *prComplete* if *input* can fully satisfy the specified picture. Returns *prIncomplete* if *input* contains data that incompletely fits the specified picture.

The following characters are used in creating format pictures:

Type of character	Character	Description
Special	#	Accept only a digit
	?	Accept only a letter (case_insensitive)
	&	Accept only a letter, force to uppercase
	@	Accept any character
	!	Accept any character, force to uppercase
Match	;	Take next character literally
	*	Repetition count
	[]	Option
	{}	Grouping operators

Type of character	Character	Description
	,	Set of alternatives
	All others	Taken literally

See also TPicResultenum

Protected data member

Pic
string Pic;
Points to a string containing the picture that specifies the format for data in the associated edit control. The constructor sets *Pic* to a string that is passed as one of the parameters.

TRadioButton class

radiobut.h

A *TRadioButton* is an interface object that represents a corresponding radio button element in Windows. Use *TRadioButton* to create a radio button control in a parent *TWindow*. A *TRadioButton* can also be used to facilitate communication between your application and the radio button controls of a *TDialog*.

Radio buttons have two states: checked and unchecked. *TRadioButton* inherits its state management member functions from its base class, *TCheckBox*. Optionally, a radio button can be part of a group (*TGroupBox*) that visually and logically groups its controls. *TRadioButton* is a streamable class.

Public constructors

Constructors

Form 1 `TRadioButton(TWindow* parent, int id, const char far* title, int x, int y, int w, int h, TGroupBox *group = 0, TModule* module = 0);`
Constructs a radio button object with the supplied parent window (*parent*), control ID (*id*), associated text (*title*), position (*x*, *y*) relative to the origin of the parent window's client area, width (*w*), height (*h*), and associated group box (*group*). Invokes the *TCheckBox* constructor with similar parameters. The style is set to WS_CHILD | WS_VISIBLE | BS_AUTORADIOBUTTON.

Form 2 `TRadioButton(TWindow* parent,int resourceId,TGroupBox *group, TModule* module = 0);`
Constructs a *TRadioButton* object to be associated with a radio button control of a *TDialog*. Invokes the *TCheckBox* constructor with identical parameters.The *resourceId* parameter must correspond to a radio button resource that you define.

See also TControl::TControl

Protected member functions

BNClicked

void BNClicked();

Responds to an incoming BN_CLICKED message.

See also BN_xxxx Button Message Constants

GetClassName

char far* GetClassName();

Returns "*BUTTON*," the name of the predefined radio button class.

Response table entries

Response table entry	Member function
EV_MESSAGE (BM_SETSTYLE, BMSetStyle)	*BMSetStyle*
EV_WM_GETDLGCODE	*EvGetDlgCode*

TRangeValidator class

validate.h

A *TRangeValidator* object determines whether the data typed by a user falls within a designated range of integers. *TRangeValidator* is a streamable class.

Public constructor

Constructor

TRangeValidator(long min, long max);

Constructs a range validator object by first calling the constructor inherited from *TFilterValidator*, passing a set of characters containing the digits '0'..'9' and the characters '+' and '–'. Sets *Min* to *min* and *Max* to *max*, establishing the range of acceptable long integer values.

See also TFilterValidator::TFilterValidator

Public member functions

Error

void Error();

Error overrides *TValidator's* virtual function and displays a message box indicating that the entered value does not fall within the specified range.

IsValid

bool IsValid(const char far* str);

Converts the string *str* into an integer number and returns **true** if the result meets all three of these conditions:

- It is a valid integer number.

- Its value is greater than or equal to *min.*
- Its value is less than or equal to *max.*

If any of those tests fails, *IsValid* returns **false**.

Transfer

uint Transfer(char far* str, void* buffer, TTransferDirection direction);

Incorporates the three types, *tdSizeData*, *tdGetData*, and *tdSetData*, that a range validator can handle for its associated edit control. *str* is the edit control's string value, and *buffer* is the data passed to the edit control. Depending on the value of *direction*, *Transfer* either sets *str* from the number in *buffer* or sets the number at *buffer* to the value of the string *str*. If *direction* is *tdSetData*, *Transfer* sets *str* from *buffer*. If *direction* is *tdGetData*, *Transfer* sets *buffer* from *str*. If *direction* is *tdSizeData*, *Transfer* neither sets nor reads data.

Transfer always returns the size of the data transferred.

See also TWindow::Transfer

Protected data members

Max

long Max;

Max is the highest valid **long** integer value for the edit control.

Min

long Min;

Min is the lowest valid **long** integer value for the edit control.

TRegion class

gdiobjec.h

TRegion, derived from *TGdiobject*, represents GDI abstract shapes or regions. *TRegion* can construct region objects with various shapes. Several operators are provided for combining and comparing regions.

Type definitions

TEllipse

enum TEllipse{Ellipse};

Defines the class-specific constant, *Ellipse*, used to distinguish the ellipse constructor from the rectangle copy constructor.

See also TRegion::TRegion (constTRect&rect), TRegion::TRegion (constTRect&ETEllipse)

Public constructors

Constructors

Form 1 TRegion();

The default constructor creates an empty *TRegion* object. *Handle* is set to 0 and *ShouldDelete* is set to **true**.

Form 2 **TRegion(HRGN handle,TAutoDelete autoDelete = NoAutoDelete);**
Creates a *TRegion* object sets the *Handle* data member to the given borrowed *handle*. The *ShouldDelete* data member defaults to false ensuring that the borrowed handle is not deleted when the C++ object is destroyed. *HRGN* is the data type representing the handle to an abstract shape.

Form 3 **TRegion(const TRegion& region);**
This public copy constructor creates a copy of the given *TRegion* object as in

```
TRegion myRegion = yourRegion;
```

Form 4 **TRegion(const TRect& rect);**
Creates a region object from the given *TRectangle* object as in

```
TRegion myRegion(rect1);
TRegion* pRegion;
pRegion = new TRegion(rect2);
```

Form 5 **TRegion(const TRect& E, TEllipse);**
Creates the elliptical *TRegion* object that inscribes the given rectangle *E*. The *TEllipse* argument distinguishes this constructor from the *TRegion(const TRect& rect)* constructor.

Form 6 **TRegion(const TRect& rect, const TSize& corner);**
Creates a *TRegion* object from the given *rect* corner.

Form 7 **TRegion(const TPoint* points,int count,int fillMode);**
Creates a filled *TRegion* object from the polygons given by *points* and *fillMode*.

Form 8 **TRegion(const TPoint* points, const int* polyCounts, int count,int fillMode);**
Creates a filled *TRegion* object from the polygons given by *points* and *fillMode*.

See also TGdiObject::Handle, TGdiObject::ShouldDelete, TPoint class, TRect class, TSize class

Public member functions

Contains

bool Contains(const TPoint& point) const;
Returns **true** if this region contains the given point.

See also TPoint

GetRgnBox

int GetRgnBox(TRect& box) const;
TRect GetRgnBox() const;
Finds the bounding rectangle (the minimum rectangle containing this region). In Form 1, the resulting rectangle is placed in *box* and the returned values are as follows:

Value	Meaning
COMPLEXREGION	Region has overlapping borders.

Value	Meaning
NULLREGION	Region is empty.
SIMPLEREGION	Region has no overlapping borders.

In Form 2, the resulting rectangle is returned.

See also TRect

operator ==

```
bool operator ==(const TRegion& other) const;
```

Returns **true** if this region is equal to the *other* region.

See also TRegion::operator!=

operator !=

```
bool operator !=(const TRegion& other) const;
```

Returns **true** if this region is not equal to the *other* region.

See also TRegion::operator ==

operator =

```
TRegion& operator =(const TRegion& source);
```

Assigns the *source* region to this region. A reference to the result is returned, allowing chained assignments.

operator +=

```
TRegion& operator +=(const TSize& delta);
```

Adds the given *delta* to each point of this region to displace (translate) it by *delta.x* and *delta.y*. Returns a reference to the resulting region.

See also TSize, TRegion::operator –=

operator –=

```
TRegion& operator –=(const TSize& delta);
TRegion& operator –=(const TRegion& source);
```

The first form subtracts the given *delta* from each point of this region to displace (translate) it by *–delta.x* and *–delta.y*. The second form creates a "difference" region consisting of all parts of this region that are not parts of the *source* region. Both forms return a reference to the resulting region.

See also TSize class, TRegion::operator +=

operator &=

```
TRegion& operator &=(const TRegion& source);
TRegion& operator &=(const TRect& source);
```

Creates the intersection of this region with the given *source* region or rectangle, and returns a reference to the result.

See also TRect class

operator |=

```
TRegion& operator |=(const TRegion& source);
TRegion& operator |=(const TRect& source);
```

Creates the union of this region and the given *source* region or rectangle, and returns a reference to the result.

See also TRect

operator ^=
TRegion& operator ^=(const TRegion& source);
TRegion& operator ^=(const TRect& source);
Creates the union of this region and the given *source* region or rectangle, but excludes any overlapping areas. Returns a reference to the resulting region object.

See also TRect class

operator HRGN()
operator HRGN() const;
Typecast operator. *HRGN* is the data type representing the handle to a physical region.

SetRectRgn
void SetRectRgn(const TRect& rect);
Creates a rectangle of the size given by *rect*.

See also TRect class

Touches
bool Touches(const TRect& rect) const;
Returns **true** if this region touches the given rectangle.

See also TRect class

TRelationship enum — layoutco.h

enum TRelationship {lmAsIs, lmPercentOf, lmAbove, lmLeftOf = lmAbove, lmBelow, lmRightOf = lmBelow, lmSameAs, lmAbsolute};

Used by the *TLayoutConstraint* struct, *TRelationship* specifies the relationship between the edges and sizes of one window and the edges and sizes of another window (which can be a parent or sibling). These relationships can be specified as either the same value as the sibling or parent window (*lmAsIs*), an absolute value (*lmAbsolute*), a percent of one of the windows (*lmPercentOf*), a value that is either added above (*lmAbove*) or left (*lmLeftOf*) of one of the windows, or a value that is subtracted from below (*lmBelow*) or right (*lmRightOf*) of one of the windows.

See also TLayoutConstraint struct

TReplaceDialog class — findrepl.h

TReplaceDialog creates a modeless dialog box that lets the user enter a selection of text to replace. Because these are model dialog boxes, you can search for text, edit the text in the window, and return to the dialog box to enter another selection. *TReplaceDialog* uses the

TFindReplaceDialog::TData class to set the user-defined values for the dialog box, such as the text strings to search for and replace.

Public constructor

Constructor
TReplaceDialog(TWindow* parent, TData& data, TResID templateName=0, const char far* title=0, TModule* module=0);
Constructs a *TReplaceDialog* object with a parent window, resource ID, template name, caption, and module instance. The *data* parameter is a reference to the *TFindReplaceDialog::TData* class that contains information about the appearance and functionality of the dialog box, such as the user-entered text strings to search for and replace.

See also TFindReplaceDialog::TData class

Protected member function

DoCreate
HWND DoCreate();
Creates a modeless find and replace dialog box.

See also TDialog::DoCreate

TResponseTableEntry class

eventhan.h

A template class, *TResponseTableEntry* lets you define a pattern for entries into a response table. Entries consist of a message, a notification code, a resource ID, a dispatcher type, and a pointer to a member function.

See DECLARE_RESPONSE_TABLE and DEFINE_RESPONSE_TABLE for additional information about the macros in the response tables.

Public data members

Dispatcher
TAnyDispatcher Dispatcher;
An abstract dispatcher type that points to one of the dispatcher functions.

Id
uint Id;
Contains the menu or accelerator resource ID (CM_xxxx) for the message response member function.

Msg
uint Msg;

Contains the ID of the message sent. These can be command messages, child id messages, notify-based messages such as LBN_SELCHANGE, or messages such as LBUTTONDOWN.

NotifyCode

uint NotifyCode;

Stores the control notification code (for example, ID_LISTBOX) for the response table entry. These can be button, combo box, edit control, or list box notification codes.

Pmf

PMF Pmf;

Points to the actual handler or member function.

Type definitions

T

typedef void(T_*PMF)();

Type for a generic member function that responds to notification messages. *T* is the template for the response table.

TRgbQuad class

color.h

TRgbQuad is a support class derived from the structure *tagRGBQUAD*, which is defined as follows:

```
typedef struct tagRGBQUAD {
   uint8  rgbBlue;
   uint8  rgbGreen;
   uint8  rgbRed;
   uint8  rgbReserved;
} RGBQUAD;
```

where *rgbBlue*, *rgbGreen*, and *rgbRed* specify the relative blue, green, and red intensities of a color. *rgbReserved* is not used and must be set to 0.

TRgbQuad is used in conjunction with the classes *TPalette* and *TColor* to simplify RGBQUAD-based color operations. Constructors are provided to create *TRgbQuad* objects from explicit RGB values, from *TColor* objects, or from other *TRgbQuad* objects.

Public constructors

Constructors

Form 1 TRgbQuad(int r, int g, int b);

Creates a *TRgbQuad* object with *rgbRed*, *rgbGreen*, and *rgbBlue* set to *r*, *g*, and *b* respectively. *rgbReserved* is set to 0.

Form 2 TRgbQuad(TColor c);

Creates a *TRgbQuad* object with *rgbRed*, *rgbGreen*, *rgbBlue* set to *c.Red*, *c.Green*, *c.Blue* respectively. *rgbReserved* is set to 0.

Form 3 **TRgbQuad(const RGBQUAD far& q);**
Creates a *TRgbQuad* object with the same values as the referenced RGBQUAD object.

See also TColor::Red, TColor::Green, TColor::Blue

TRgbTriple class — color.h

TRgbTriple is a support class derived from the structure *tagRgbTriple*, which is defined as follows:

```
typedef struct tagRGBTRIPLE {
    uint8  rgbBlue;
    uint8  rgbGreen;
    uint8  rgbRed;
} RGBTRIPLE;
```

rgbBlue, *rgbGreen*, and *rgbRed* specify the relative blue, green, and red intensities for a color.

TRgbTriple is used in conjunction with the classes *TPalette* and *TColor* to simplify bmci-color-based operations. Constructors are provided to create *TRgbTriple* objects from explicit RGB values, from *TColor* objects, or from other *TRgbTriple* objects.

Public constructors

Constructors

Form 1 **TRgbTriple(int r, int g, int b);**
Creates a *TRgbTriple* object with *rgbRed*, *rgbGreen*, and *rgbBlue* set to *r*, *g*, and *b* respectively.

Form 2 **TRgbTriple(TColor c);**
Creates a *TRgbTriple* object with *rgbRed*, *rgbGreen*, *rgbBlue* set to *c.Red*, *c.Green*, and *c.Blue* respectivelyrgbReserved is set to 0.

Form 3 **TRgbTriple(const RGBTRIPLE far& t);**
Creates a *TRgbTriple* object with the same values as the referenced RGBTRIPLE object.

See also tag RGBTRIPLE struct, TColor::Red, TColor::Green, TColor::Blue

TScreenDC class — dc.h

Derived from *TWindowDC*, *TScreenDC* is a DC class that provides direct access to the screen bitmap. *TScreenDC* gets a DC for handle 0, which is for the whole screen with no clipping. Handle 0 paints on top of other windows.

Public constructor

Constructor
TScreenDC();

Default constructor for *TScreenDC* objects.

TScrollBar class

scrollba.h

TScrollBar objects represent standalone vertical and horizontal scroll bar controls. Most of *TScrollBar*'s member functions manage the scroll bar's sliding box (thumb) position and range.

One special feature of the type *TScrollBar* is the notify-based set of member functions that automatically adjust the scroll bar's thumb position in response to scroll bar messages.

Never place *TScrollBar* objects in windows that have either the WS_HSCROLL or WS_VSCROLL styles in their attributes.

TScrollBar is a streamable class.

Public data members

LineMagnitude

int LineMagnitude;

LineMagnitude is the number of range units to scroll the scroll bar when the user requests a small movement by clicking on the scroll bar's arrows. *TScrollBar*'s constructor sets *LineMagnitude* to 1 by default. (The scroll range is 0–100 by default.)

See also TScrollBar::SetupWindow

PageMagnitude

int PageMagnitude;

PageMagnitude is the number of range units to scroll the scroll bar when the user requests a large movement by clicking in the scroll bar's scrolling area. *TScrollBar*'s constructor sets *PageMagnitude* to 10 by default. (The scroll range is 0–100 by default.)

Public constructors

Constructors

Form 1 TScrollBar(TWindow* parent, int id, int x, int y, int w, int h, bool isHScrollBar, TModule* module = 0);

Constructs and initializes a *TScrollBar* object with the given parent window (*parent*), a control ID (*id*), a position (*x, y*), and a size of (*w, h*). Invokes the *TControl* constructor with similar parameters. If *isHScrollBar* is **true**, adds SBS_HORZ to the window style. If not true, adds SBS_VERT. If the supplied height for a horizontal scroll bar or the supplied width for a vertical scroll bar is 0, a standard value is used. *LineMagnitude* is initialized to 1 and *PageMagnitude* is set to 10.

Form 2 TScrollBar(TWindow* parent,int resourceId,TModule* module = 0);

Constructs a *TScrollBar* object to be associated with a scroll bar control of a *TDialog*. Invokes the *TControl* constructor with identical parameters.

The *resourceId* parameter must correspond to a scroll bar resource that you define.

See also TControl::TControl

Public member functions

DeltaPos

virtual int DeltaPos(int delta);

Calls *SetPosition* to change the scroll bar's thumb position by the value supplied in *delta*. A positive *delta* moves the thumb down or right. A negative *delta* value moves the thumb up or *left*. The new thumb position is returned.

See also TScrollBar::SetPosition

EvHScroll

void EvHScroll(uint scrollCode, uint thumbPos, HWND hWndCtl);

Response table handler that calls the virtual function (*SBBottom, SBLineDown* and so on) in response to messages sent by *TWindow::DispatchScroll*.

EvVScroll

void EvVScroll(uint scrollCode, uint thumbPos, HWND hWndCtl);

Response table handler that calls the virtual function (*SBBottom, SBLineDown* and so on) in response to messages sent by *TWindow::DispatchScroll*.

GetPosition

virtual int GetPosition() const;

Returns the scroll bar's current thumb position.

See also TScrollBar::SetPosition, TScrollBarData struct

GetRange

virtual void GetRange(int& min, int& max) const;

Returns the end values of the present range of scroll bar thumb positions in *min* and *max*.

See also TScrollBar::SetPosition, TScrollBar::SetRange, TScrollBarData struct

SBBottom

virtual void SBBottom();

Calls *SetPosition* to move the thumb to the bottom or right of the scroll bar. *SB_BOTTOM* is called to respond to the thumb being dragged to the bottom or rightmost position of the scroll bar.

See also TScrollBar::SetPosition

SBLineDown

virtual void SBLineDown();

Calls *SetPosition* to move the thumb down or right (by *LineMagnitude* units). *SBLineDown* is called to respond to a click on the bottom or right arrow of the scroll bar.

See also TScrollBar::SetPosition

SBLineUp

virtual void SBLineUp();

Calls *SetPosition* to move the thumb up or left (by *LineMagnitude* units). *SBLineUp* is called to respond to a click on the top or left arrow of the scroll bar.

See also TScrollBar::SetPosition

SBPageDown

virtual void SBPageDown();

Calls *SetPosition* to move the thumb down or right (by *PageMagnitude* units). *SBPageDown* is called to respond to a click in the bottom or right scrolling area of the scroll bar.

See also TScrollBar::SetPosition

SBPageUp

virtual void SBPageUp();

Calls *SetPosition* to move the thumb up or left (by *PageMagnitude* units). *SBPageUp* is called to respond to a click in the top or left scrolling area of the scroll bar.

See also TScrollBar::SetPosition

SBThumbPosition

virtual void SBThumbPosition(int thumbPos);

Calls *SetPosition* to move the thumb. *SBThumbPosition* is called to respond when the thumb is set to a new position.

See also TScrollBar::SetPosition

SBThumbTrack

virtual void SBThumbTrack(int thumbPos);

Calls *SetPosition* to move the thumb as it is being dragged to a new position.

See also TScrollBar::SetPosition

SBTop

virtual void SBTop();

Calls *SetPosition* to move the thumb to the top or right of the scroll bar. *SBTop* is called to respond to the thumb being dragged to the top or rightmost position on the scroll bar.

See also TScrollBar::SetPosition

SetPosition

virtual void SetPosition(int thumbPos);

Moves the thumb to the position specified in *ThumbPos*. If *ThumbPos* is outside the present range of the scroll bar, the thumb is moved to the closest position within range.

See also TScrollBar::GetPosition

SetRange

virtual void SetRange(int min, int max);

Sets the scroll bar to the range between *min* and *max*.

See also TScrollBar::GetRange

Transfer

uint Transfer(void* buffer, TTransferDirection direction);

Transfers scroll bar data to or from the transfer buffer pointed to by *buffer*. *buffer* is expected to point to a *TScrollBarData* structure.

Data is transferred to or from the transfer buffer if *tdGetData* or *tdSetData* is supplied as the direction.

Transfer always returns the size of the transfer data (the size of the *TScrollBarData* structure). To retrieve the size of this data without transferring data, pass *tdSizeData* as the *direction*.

See also TScrollBarData struct

Protected member functions

GetClassName

char far* GetClassName();

Returns the name of *TScrollBar's* registration class, "SCROLLBAR".

SetupWindow

void SetupWindow();

Sets the scroll bar's range to 0, 100. To redefine this range, call *SetRange*.

See also TScrollBar::SetRange

Response table entries

Response table entry	Member function
EV_WM_HSCROLL	*EvHScroll*
EV_WM_VSCROLL	*EvVScroll*

TScrollBarData struct

scrollba.h

The *TScrollBarData* structure contains integer values that represent a range of thumb positions on the scroll bar. *TScrollBar*'s function *GetRange* calls *TScrollBarData* to obtain the highest and lowest thumb positions on the scroll bar. *GetPosition* calls *TScrollBarData* to obtain the current thumb position on the scroll bar.

See also

TScrollBar::Transfer

Public data members

HighValue

int HighValue;

Contains the highest value of the thumb position in the scroll bar's range.

See also TScrollBar::GetRange

LowValue
int LowValue;
Contains the lowest value of thumb position in the scroll bar's range.

See also TScrollBar::GetRange

Position
int Position;
Contains the scroll bar's thumb position.

See also TScrollBar::GetPosition

TScroller class scroller.h

TScroller supports an automatic window-scrolling mechanism (referred to as autoscrolling) that works in conjunction with horizontal or vertical window scroll bars (it also works if there are no scroll bars). When autoscrolling is activated, the window automatically scrolls when the mouse is dragged from inside the client area of the window to outside that area. If the *AutoMode* data member is true, *TScroller* performs autoscrolling.

To use *TScroller*, set the *Scroller* member of your *TWindow* descendant to a *TScroller* object instantiated in the constructor of your *TWindow* descendant. *TScroller* is a streamable class.

Public data members

AutoMode
bool AutoMode;
Is **true** if automatic scrolling is activated.

AutoOrg
bool AutoOrg;
Is **true** if scroller offsets original.

HasHScrollBar, HasVScrollBar
bool HasHScrollBar, HasVScrollBar;
Is true if scroller has horizontal or vertical scroll.

TrackMode
bool TrackMode;
Is **true** if track scrolling is activated.

Window
TWindow* Window;
Points to the window whose client area scroller is to be managed.

XLine, YLine
int XLine, YLine;

Specifies the number of logical device units per line to scroll the rectangle in the horizontal (*X*) and vertical (*Y*) directions.

XPage, YPage
int XPage, YPage;
Specifies the number of logical device units per page to scroll the rectangle in the horizontal (*X*) and vertical (*Y*) directions.

XPos, YPos
long XPos, YPos;
Specifies the current position of the rectangle in horizontal (*XPos*) and vertical (*YPos*) scroll units.

XRange, YRange
long XRange, YRange;
Specifies the number of horizontal and vertical scroll units.

XUnit, YUnit
int XUnit, YUnit;
Specifies the amount (in logical device units) to scroll the rectangle in the horizontal (*X*) and vertical (*Y*) directions. The rectangle is scrolled right if *XUnit* is positive and left if *XUnit* is negative. The rectangle is scrolled down if *YUnit* is positive and up if *YUnit* is negative.

Public constructor and destructor

Constructor
TScroller(TWindow* window, int xUnit, int yUnit, long xRange, long yRange);
Constructs a *TScroller* object with *window* as the owner window, and *xUnit*, *yUnit*, *xRange*, and *yRange* as *xUnit*, *yUnit*, *xRange* and *yRange*, respectively. Initializes data members to default values. *HasHScrollBar* and *HasVScrollBar* are set according to the scroll bar attributes of the owner window.

Destructor
virtual ~TScroller();
Destructs a *TScroller* object. Sets owning window's *Scroller* number variable to 0.

Public member functions

AutoScroll
virtual void AutoScroll();
Scrolls the owner window's display in response to the mouse being dragged from inside to outside the window. The direction and the amount by which the display is scrolled depend on the current position of the mouse.

BeginView
virtual void BeginView(TDC& dc, TRect& rect);
If *TScroller_AutoOrg* is **true** (default condition), *BeginView* automatically offsets the origin of the logical coordinates of the client area by *XPos*, *YPos* during a paint

operation. If *AutoOrg* is **false** (for example, when the scroller is larger than 32,767 units), you must set the offset manually.

EndView

virtual void EndView();

Updates the position of the owner window's scroll bars to be coordinated with the position of the *TScroller*.

HScroll

virtual void HScroll(uint scrollEvent, int thumbPos);

Responds to the specified horizontal *scrollEvent* by calling *ScrollBy* or *ScrollTo*. The type of scroll event is identified by the corresponding SB_ constants. *thumbPos* contains the current thumb position when the scroller is notified of SB_THUMBTRACK and SB_THUMBPOSITION scroll events.

IsAutoMode

virtual bool IsAutoMode();

IsAutoMode is **true** if automatic scrolling is activated.

See also TScroller::AutoMode

IsVisibleRect

bool IsVisibleRect(long x, long y, int xExt, int yExt);

Is **true** if the rectangle (*x*, *y*, *xExt*, and *yExt*) is visible.

SetPageSize

virtual void SetPageSize();

Sets the *XPage* and *YPage* data members to the width and height (in *XUnits* and *YUnits*) of the owner window's client area.

See also TScroller::XPage, YPage, TScroller::XUnit, YUnit

SetRange

virtual void SetRange(long xRange, long yRange);

Sets the *xRange* and *yRange* of the *TScroller* to the parameters specified. Then calls *SetSBarRange* to synchronize the range of the owner window's scroll bars.

See also TScroller::SetSBarRange

SetSBarRange

virtual void SetSBarRange();

Sets the range of the owner window's scroll bars to match the range of the *TScroller*.

SetUnits

virtual void SetUnits(int xUnit, int yUnit);

Sets the *XUnit* and *YUnit* data members to *TheXUnit* and *TheYUnit*, respectively. Updates *XPage* and *YPage* by calling *SetPageSize*.

See also TScroller::XPage, YPage, TScroller::XUnit, YUnit

ScrollBy

void ScrollBy(long dx, long dy);

Scrolls to a position calculated using the passed delta values (*dx* and *dy*). A positive delta position moves the thumb position down or right. A negative delta position moves the thumb up or left.

ScrollTo
virtual void ScrollTo(long x, long y);
Scrolls the rectangle to the position specified in *x* and *y*.

SetWindow
void SetWindow(TWindow* win);
Sets the owning window to *win*.

VScroll
virtual void VScroll(uint scrollEvent, int thumbPos);

Responds to the specified vertical *scrollEvent* by calling *ScrollBy* or *ScrollTo*. The type of scroll event is identified by the corresponding SB_ constants. *thumbPos* contains the current thumb position when the scroller is notified of SB_THUMBTRACK and SB_THUMBPOSITION scroll events.

See also TScroller::ScrollTo

XScrollValue
int XScrollValue(long rangeUnit);
XScrollValue converts a horizontal range value from the scroll bar to a horizontal scroll value.

See also TScroller::YScrollValue

XRangeValue
int XRangeValue(int scrollUnit);
XRangeValue converts a horizontal scroll value from the scroll bar to a horizontal range value.

See also TScroller::YRangeValue

YScrollValue
int YScrollValue(long rangeUnit);
YScrollValue converts a vertical range value from the scroll bar to a vertical scroll value.

See also TScroller::XScrollValue

YRangeValue
int YRangeValue(int scrollUnit);
YRangeValue converts a vertical scroll value from the scroll bar to a vertical range value.

See also TScroller::XRangeValue

TSeparatorGadget class

gadget.h

TSeparatorGadget is a simple class you can use to create a separator between gadgets. To do so, you must specify the size of the separator in units of SM_CXBORDER (width of

the window frame) and SM_CYBORDER (height of the window frame). The right and bottom boundaries of the separator are set after calling *GetsystemMetrics*. By default, the separator disables itself and turns off shrink-wrapping. Note that the default border style is none.

See also

TGadget::TBorderStyle enum

Public member function

TSeparatorGadget

TSeparatorGadget(int size = 6);

Used for both the width and the height of the separator, *size* is initialized at 6 border units (the width or height of a thin window border).

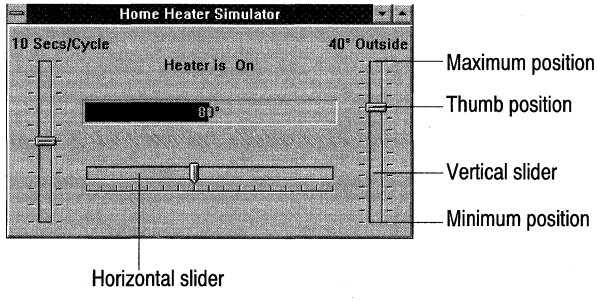
TSlider class

slider.h

An abstract base class derived from *TScrollBar*, *TSlider* defines the basic behavior of sliders (controls that are used for providing nonscrolling, position information). Like scroll bars, sliders have minimum and maximum positions as well as line and page magnitude. Sliders can be moved using either the mouse or the keyboard. If you use a mouse to move the slider, you can drag the thumb position, click on the slot on either side of the thumb position to move the thumb by a specified amount (*PageMagnitude*), or click on the ruler to position the thumb at a specific spot on the slider. The keyboard's Home and End keys move the thumb position to the minimum (*Min*) and maximum (*Max*) positions on the slider.

You can use *TSlider*'s member functions to cause the thumb positions to automatically align with the nearest tick positions. (This is referred to as snapping.) You can also specify the tick gaps (the space between the lines that separate the major divisions of the X- or Y-axis).

The sample program SLIDER.CPP on your distribution disk displays the following thermostat that uses sliders.



See the two derived classes, *THSlider* and *TVSlider*, for specific details about horizontal and vertical sliders.

Public constructor and destructor

Constructor

TSlider(TWindow* parent, int id, int X, int Y, int W, int H, TResId thumbResId, TModule* module = 0);

Constructs a slider object setting *Pos* and *ThumbRgn* to 0, *TicGap* to *Range* divided by 10, *SlotThick* to 17, *Snap* to true, and *Sliding* to false. Sets *Attr.W* and *Attr.H* to the values in *X* and *Y*. *ThumbResId* is set to *thumbResId*.

Destructor

~TSlider();

Destructs a *TSlider* object and deletes *ThumbRgn*.

Public member functions

GetPosition

int GetPosition() const;

Returns the slider's current thumb position. Overloads *TScrollBar's* virtual function.

See also TSlider::SetPosition

GetRange

void GetRange(int &min, int &max) const;

Returns the end values of the present range of slider thumb positions in *min* and *max*. Overloads *TScrollBar's* virtual function.

See also TSlider::SetRange

SetPosition

void SetPosition(int thumbPos);

Moves the thumb to the position specified in *thumbPos*. If *thumbPos* is outside the present range of the slider, the thumb is moved to the closest position within the specified range. Overloads *TScrollBar's* virtual function.

See also TSlider::GetPosition

SetRange

void SetRange(int min, int max);

Sets the slider to the range between *min* and *max*. Overloads *TScrollBar's* virtual function.

See also TSlider::GetRange

SetRuler

void SetRuler(int ticGap, bool snap = false);

Sets the slider's ruler. Each slider has a built-in ruler that is drawn with the slider. The ruler, which can be blank or have tick marks on it, can be created so that it forces the thumb to snap to the tick positions automatically.

Protected member functions

EvEraseBkgnd

bool EvEraseBkgnd(HDC hDC);

Responds to a WM_ERASEBKGND message and erases the background of the slider when the slider is changed. Calls the virtual functions *PaintRuler*, *PaintSlot*, and *PaintThumb* to paint the components of the slider. To avoid flickering, *EvEraseBkgnd* is called to erase the background as the painting occurs.

See also TSlider::EvPaint

EvGetDlgCode

uint EvGetDlgCode(MSG far* msg);

Responds to a WM_GETDLGCODE message and let the dialog manater control the response to a DIRECTION key or TAB key input. Captures cursor-movement keys to move the thumb by returning a DLGC_WANTARROWS message, which indicates that direction keys are desired.The *msg* parameter indicates the kind of message, for example a a control or a command message, sent to the dialog box manager.

EvGetDlgCode returns a code that indicates how the control message is to be treated.

See also TButton::EvGetDlgCode, TWindow::DefaultProcessing, DLGC_xxxx dialog control message constants

EvKeyDown

void EvKeyDown(uint key, uint repeatCount, uint flags);

EvKeyDown translates the virtual key code into a movement and then moves the thumb. *key* indicates the virtual key code of the pressed key, *repeatCount* holds the number of times the same key is pressed, and *flags* contains one of the following messages, which translate to virtual key (VK) codes:

Value	Virtual key code
SB_PAGEUP	VK_PRIOR
SB_PAGEDOWN	VK_NEXT
SB_BOTTOM	VK_END
SB_TOP	VK_HOME
SB_LINEUP	VK_LEFT(same as SB_LINELEFT)
SB_LINEUP	VK_UP
SB_LINEDOWN	VK_RIGHT(same as SB_LINERIGHT)
SB_LINEDOWN	VK_DOWN

EvKillFocus

void EvKillFocus(HWND hWndGetFocus);

In response to a WM_KILLFOCUS message sent to a window that is losing the keyboard, *EvKillFocus* hides and then destroys the caret.

EvLButtonDblClk

void EvLButtonDblClk(uint modKeys, TPoint& point);

Responds to a WM_LBUTTONDBLCLK message (which indicates the user double-clicked the left mouse button), then throws away the messages so the base class doesn't receive them.

EvLButtonDown

```
void EvLButtonDown(uint modKeys, TPoint& point);
```

Responds to a mouse press by positioning the thumb or beginning a mouse drag. If the mouse is pressed down while it is over the thumb, *EvLButtonDown* enters sliding state. If the mouse is in the slot, *EvLButtonDown* pages up or down. If the mouse is on the ruler, *EvLButtonDown* jumps to that position. *EvLButtonDown* generates a scroll code of SB_THUMBPOSITION, SB_LINEUP, SB_LINEDOWN, SB_PAGEUP, SB_PAGEDOWN, SB_THUMBTRACK.

See also TSlider::EvLButtonUp

EvLButtonUp

```
void EvLButtonUp(uint modKeys, TPoint& point);
```

If the mouse button is released, *EvLButtonUp* ends sliding, paging, or jumping to a position on the ruler.

See also TSlider::EvLButtonDown

EvMouseMove

```
void EvMouseMove(uint modKeys, TPoint& point);
```

Moves the mouse to the indicated position. If the mouse is being dragged, *EvMouseMove* positions the thumb and sends the appropriate message to the parent window.

EvPaint

```
void EvPaint();
```

Paints the entire slider—ruler, slot, and thumb. Calls the virtual functions *PaintRuler*, *PaintSlot*, and *PaintThumb* to paint the components of the slider.

See also TSlider::EvEraseBkgnd

EvSetFocus

```
void EvSetFocus(HWND hWndLostFocus);
```

Creates a blinking caret to show the focus in the current window.

EvSize

```
void EvSize(uint sizeType, TSize& size);
```

Recalculates the size of the slider when the window size is changed.

GetBkColor

```
void GetBkColor(TDC& dc);
```

Sends a WM_CTLCOLOR message to the parent and calls *dc::GetBkColor* to extract the background color for the slider.

HitTest

```
virtual int HitTest(TPoint& point) = 0;
```

Gets information about where a given *X*, *Y* location falls on the slider. The return value is in *scrollCodes*. Each of the derived classes performs comparisons to return a scroll code.

See also TSlider::NotifyParent

NotifyParent

virtual void NotifyParent(int scrollCode, int pos=0) = 0;

Sends a WS_HSCROLL or WS_VSCROLL message to the parent window.

See also TVSlider::HitTest

PaintRuler

virtual void PaintRuler(TDC& dc) = 0;

Paints the ruler. It is assumed that the slot or thumb do not overlap the ruler.

PaintSlot

virtual void PaintSlot(TDC& dc) = 0;

Paints the slot in which the thumb slides.

PaintThumb

virtual void PaintThumb(TDC& dc);

Paints the thumb itself using a resource DIB translated to the current system button colors and which overlaps the slot.

PointToPos

virtual int PointToPos(TPoint& point) = 0;

Translates an *X,Y* point to a position in slider units.

See also TSlider::PosToPoint

PosToPoint

virtual TPoint PosToPoint(int pos) = 0;

Translates a position in slider units to an *X,Y* point.

See also TSlider::PointToPos

SetupThumbRgn

virtual void SetupThumbRgn();

Creates the region that defines the thumb shape for this slider class. Although the default region is a simple bounding rectangle, it can be any shape. While the slider thumb is being moved, this region is used for testing the mouse position and updating the thumb position.

See also TSlider::ThumbRgn

SetupWindow

void SetupWindow();

Calls *TScrollBar::SetupWindow* and *SetupThumbRgn* to set up the window.

See also TScrollBar::SetupWindow

SlideThumb

virtual void SlideThumb(TDC& dc, int thumbPos);

Slides the thumb to a given position and performs the necessary blitting and painting.

SnapPos

int SnapPos(int pos);

Constrains *pos* so it is in the range from *Min* to *Max* and (if snapping is enabled) performs snapping by rounding *pos* to the nearest *TicGap*.

See also TSlider::TicGap

Protected data members

BkColor

TColor BkColor;

Stores the background color of the slider.

CaretRect

TRect CaretRect;

Refers to the position of the caret's rectangle.

Max

int Max;

Contains the maximum value of the slider position.

Min

int Min;

Contains the minimum value of the slider position.

MouseOffset

static TSize MouseOffset;

MouseOffset shows the offset from the rectangle's original top left position to the position where the mouse is clicked. Thus, when the rectangle is moved, it can maintain the same relationship to the position of the mouse click.

Pos

int Pos;

Indicates where the thumb is positioned on the slider.

See also TSlider::GetPosition

Range

uint Range;

Contains the difference between the maximum and minimum range of the slider.

SlideDC

static TDC* SlideDC;

SlideDC is used while the mouse is down and the thumb is sliding. It reflects the movement of the mouse on the DC.

Sliding

bool Sliding;

Sliding is **true** if the thumb is sliding. Otherwise, **false**.

SlotThick

int SlotThick;

Indicates the thickness of the slot. Set to 17 by default.

Snap
bool Snap;
Snap is **true** if snapping is activated; otherwise **false**.

ThumbRect
TRect ThumbRect;
Holds the thumb's bounding rectangle.

ThumbResId
TResId ThumbResId;
ThumbResId is the bitmap resource ID for the thumb knob.

ThumbRgn
virtual void SetupThumbRgn();
Creates the region that defines the thumb shape for this slider class. Although the default region is a simple bounding rectangle, it can be any shape. While the slider thumb is being moved, this region is used for testing the mouse position and updating the thumb position.

See also TSlider::ThumbRgn

TicGap
int TicGap;
Specifies the amount of space in pixels between ticks.

Response table entries

Response table entry	Member function
EV_WM_ERASEBKGND	EvEraseBkgnd
EV_WM_GETDLGCODE	EvGetDlgCode
EV_WM_KEYDOWN	EvKeyDown
EV_WM_KILLFOCUS	EvKillFocus
EV_WM_LBUTTONDBLCLK	EvLButtonDblClk
EV_WM_LBUTTONDOWN	EvLButtonDown
EV_WM_LBUTTONUP	EvLButtonUp
EV_WM_MOUSEMOVE	EvMouseMove
EV_WM_PAINT	EvPaint
EV_WM_SETFOCUS	EvSetFocus
EV_WM_SIZE	EvSize

THSlider class

slider.h

Derived from *TSlider*, *THSlider* provides implementation details for horizontal sliders. See *TSlider* for an illustration of a horizontal slider.

Public constructors

Constructors

```
THSlider(TWindow* parent, int id, int X, int Y, int W, int H, TResId thumbResId = IDB_HSLIDERTHUMB,
  TModule* module = 0);
```

Constructs a slider object with a default bitmap resource ID of IDB_HSLIDERTHUMB for the thumb knob.

Protected member functions

HitTest

```
int HitTest(TPoint& point);
```

Overrides *TSlider*'s virtual function and gets information about where a given X, Y location falls on the slider. The return value is in *scrollCodes*.

See also TSlider::HitTest

NotifyParent

```
void NotifyParent(int scrollCode, int pos=0);
```

Overrides *TSlider*'s virtual function and sends a WS_HSCROLL message to the parent window.

See also TSlider::NotifyParent

PaintRuler

```
void PaintRuler(TDC&);
```

Overrides *TSlider*'s virtual function and paints the horizontal ruler.

See also TSlider::PaintRuler

PaintSlot

```
void PaintSlot(TDC&);
```

Overrides *TSlider*'s virtual function and paints the slot in which the thumb slides.

See also TSlider::PaintSlot

PointToPos

```
int PointToPos(TPoint& point);
```

Overrides *TSlider*'s virtual function and translates an *X,Y* point to a position in slider units.

See also TSlider::PointToPos

PosToPoint

```
TPoint PosToPoint(int pos);
```

Overrides *TSlider*'s virtual function and translates a position in slider units to an *X,Y* point.

See also TSlider::PosToPoint

TVSlider class

slider.h

Derived from *TSlider*, *TVSlider* provides implementation details for vertical sliders. See *TSlider* for an illustration of a vertical slider.

Public constructor

Constructor

TVSlider(TWindow* parent, int id, int X, int Y, int W, int H, TResId thumbResId = IDB_VSLIDERTHUMB, TModule* module = 0);

Constructs a vertical slider object with a default bitmap resource ID of IDB_VSLIDERTHUMB for the thumb knob.

Protected member functions

HitTest

int HitTest(TPoint& point);

Overrides *TSlider*'s virtual function and gets information about where a given X, Y location falls on the slider. The return value is in *scrollCodes*.

See also TSlider::HitTest

NotifyParent

void NotifyParent(int scrollCode, int pos=0);

Overrides *TSlider's* virtual function and sends a WS_VSCROLL message to the parent window.

See also TSlider::NotifyParent

PaintRuler

void PaintRuler(TDC& dc);

Overrides *TSlider*'s virtual function and paints the vertical ruler.

See also TSlider::PaintRuler

PaintSlot

void PaintSlot(TDC& dc);

Overrides *TSlider*'s virtual function and paints the slot in which the thumb slides.

See also TSlider::PaintSlot

PointToPos

int PointToPos(TPoint& point);

Overrides *TSlider*'s virtual function and translates an *X,Y* point to a position in slider units.

See also TSlider::PointToPos

PosToPoint

TPoint PosToPoint(int pos);

Overrides *TSlider*'s virtual function and translates a position in slider units to an *X,Y* point.

See also TSlider::PosToPoint

TSortedStringArray class

validate.h

Implements a list of ASCII characters stored as a sorted array of elements that are string objects. *TSortedStringArray* can perform many of the string manipulation functions, such as adding and removing elements from the array, and provides many of the common C++ functions implemented by container classes.

See also

TValidator class, TLookupValidator class

Public constructor

Constructor

```
TSortedStringArray(int upper, int lower, int delta);
```

Constructs a *TSortedStringArray* object with an upper boundary of *upper*, a lower boundary of *lower*, and a growth delta of *delta*.

Type definitions

void

```
typedef void (*IterFunc)(string&, void*);
```

Function type used as a parameter to the *ForEach* member function.

int

```
typedef int (*CondFunc)(const string&, void*);
```

Function type used as a parameter to the *FirstThat* and *LastThat* member functions.

Public member functions

Add

```
int Add(const string& t);
```

Adds an element to the array at the next available index position. Adding an element beyond the upper boundary leads to an overflow condition. If this condition occurs and the growth delta, *delta*, (from the constructor) is nonzero, the array is expanded (by sufficient multiples of *delta* bytes) to accommodate the addition. If *delta* is zero, *Add* fails. *Add* returns 0 if the object could not be added.

ArraySize

```
unsigned ArraySize() const;
```

Returns the size of the array.

Destroy

Form 1 `int Destroy(const string& t);`
Removes the element specified by *t* and deletes it.

Form 2 `int Destroy(int loc);`
Removes an element from the array at the specified index location, *loc*, and deletes it.

Detach

Form 1 `int Detach(const string& t);`
Removes the specified element from the array. Returns 1 if successful; otherwise, returns 0.

Form 2 `int Detach(int loc);`
Removes an element from the array at the specified index location, *loc*. Returns 1 if successful; otherwise, returns 0.

Find

`int Find(const string& t) const;`
Finds the first element represented by *t* and returns the index where the element is located.

FirstThat

`string* FirstThat(CondFunc cond, void* args) const;`
Returns a pointer to the first element in the array that satisfies a specified condition. *cond* is the test function pointer that returns true for a specified condition. *args* contains the various arguments passed. Returns 0 if no element in the array satisfies the given condition. Because *FirstThat* creates its own internal iterator, you can use it as a search function also.

Flush

`void Flush();`
Removes all elements from the array without destroying the array itself.

ForEach

`void ForEach(IterFunc iter, void* args);`
Creates an internal iterator to execute the specified function, *iter*, for each element in the array. Use the *args* argument to pass various kinds of data to this function.

GetItemsInContainer

`unsigned GetItemsInContainer() const;`
Returns the number of elements in the array.

HasMember

`int HasMember(const string& t) const;`
Returns 1 if the element specified by *t* exists in the array; otherwise returns 0.

IsEmpty

`int IsEmpty() const;`
Returns 1 if the array is empty; otherwise, returns 0.

IsFull

`int IsFull() const`

Returns 1 if the array is full; otherwise, returns 0.

LastThat
string* LastThat(CondFunc cond, void* args) const;
Returns a pointer to the last element in the array that satisfies a specified condition. *cond* is the test function pointer that returns true for a specified condition. *args* contains the various arguments passed. Returns 0 if no element in the array satisfies the given condition. Because *LastThat* creates its own internal iterator, you can use it as a search function also.

LowerBound
int LowerBound() const;
Returns the array's lower boundary.

operator []
Form 1 string& operator [](int loc);
Returns a reference to the element at the location specified by *loc*. This version resizes the array if it is necessary to make *loc* a valid index.

Form 2 string& operator [](int loc) const;
Returns a reference to the element at the location specified by *loc*. This version throws an exception in the debugging version on an attempt to index out of bounds.

UpperBound
int UpperBound() const;
Returns the array's upper boundary.

TStatic class

static.h

A *TStatic* is an interface object that represents a static text interface element. Static elements consist of text or graphics that the user does not change. An application, however, can modify the static control. You must use a *TStatic* object, for example, to create a static control that's used to display a text label such as a copyright notice in a parent *TWindow*. *TStatic* can also be used to make it easier to modify the text of static controls in *TDialogs*. See the sample program in the EXAMPLES\OWL\OWLAPI\STATIC directory for an example of a static control.

Public data members

TextLen
uint TextLen;
TextLen holds the size of the text buffer for static controls. The number of characters that can actually be stored in the static control is one less than *TextLen* because of the null terminator on the string. *TextLen* is also the number of bytes transferred by the *Transfer* member function.

Public constructors

Constructors

Form 1 `TStatic(TWindow* parent, int id, const char far* title, int x, int y, int w, int h, uint textLen = 0, TModule* module = 0);`

Constructs a static control object with the supplied parent window (*parent*), control ID (*Id*), text (*title*), position (*x, y*) relative to the origin of the parent window's client area, width (*w*), height (*h*), and default text length (*textLen*) of zero. By default, the static control is visible upon creation and has left-justified text. (set to WS_CHILD | WS_VISIBLE | WS_GROUP | SS_LEFT.) Invokes a *TControl* constructor.You can change the default style of the static control by via the control object's constructor.

Form 2 `TStatic(TWindow* parent, int resourceId, uint textLen = 0, TModule* module = 0);`

Constructs a *TStatic* object to be associated with a static control interface control of a *TDialog*. Invokes the *TControl* constructor with similar parameters then sets *TextLen* to *textLen*. Disables the data transfer mechanism by calling *DisableTransfer*. The *resourceId* parameter must correspond to a static control resource that you define.

See also TControl::TControl

Public member functions

Clear

`virtual void Clear();`

Clears the text of the associated static control.

GetText

`int GetText(char far* str, int maxChars);`

Retrieves the static control's text, stores it in the *str* argument of *maxChars* size, and returns the number of characters copied.

GetTextLen

`int GetTextLen();`

Returns the length of the static control's text.

SetText

`void SetText(const char far* str);`

Sets the static control's text to the string supplied in *str*.

Transfer

`uint Transfer(void* buffer, TTransferDirection direction);`

Transfers *TextLen* characters of text to or from a transfer buffer pointed to by *buffer*. If *direction* is *tdGetData*, the text is transferred to the buffer from the static control. If *direction* is *tdSetData*, the static control's text is set to the text contained in the transfer buffer. *Transfer* returns *TextLen*, the number of bytes stored in or retrieved from the buffer. If *direction* is *tdSizeData*, *Transfer* returns *TextLen* without transferring data.

Protected member functions

EvSize
void EvSize (uint sizeType, TSize& size);
Overrides *TWindow*'s virtual function. When the static control is resized, *EvSize* ensures it's also repainted.

GetClassName
virtual char far* GetClassName();
Returns the name of *TStatic*'s registration class (STATIC), or returns STATIC_CLASS if BWCC is enabled.

Response table entries

Response table entry	Member function
EV_WM_SIZE	*EvSize*

TStatus class

except.h

Used primarily for backward compatibility with previous versions of ObjectWindows, *TStatus* is used by *TModule* and *TWindow* to indicate an error in the initialization of an interface object. If *Status* is set to a nonzero value, a *TXCompatibility* exception is thrown.

See also
TModule::Status, TWindow::Status

Public constructor

Constructor
TStatus();
Constructs a *TStatus* object and initializes the status code to 0.

Public data members

operator =
TStatus& operator =(int statusCode);
Sets the status code and throws a *TXCompatibility* exception.

operator int()
operator int() const;
Returns the status code.

TStatusBar class

statusba.h

TStatusBar provides support for status bars, inserted gadgets, hint text, and keyboard modes. In contrast to plain message bars, status bars provide several display options. ObjectWindows status bars let you include multiple text gadgets (the text on the left of the status bar) and different border styles. You can also reserve space for mode indicators (the text that displays the program's current state, such as extended selection (of keys and other modes), ***CapsLock***, ***NumLock***, ***ScrollLock***, ***Overtype***, and macro recording). By default, ObjectWindows uses these virtual key codes and displays the following strings for the indicated enabled modes:

Mode	Virtual key code	Text displayed
Extended		EXT
CapsLock	VK_CAPITAL	CAPS
NumLock	VK_NUMLOCK	NUM
ScrollLock	VK_SCROLL	SCRL
Overtype	VK_INSERT	OVR
Macro recording		REC

TStatusBar creates text gadgets for the mode indicators you request and adjusts the spacing between mode indicators. The *TSpacing* struct stores spacing and layout unit constraints.

Like other control bars, the status bar is constructed and destroyed at the same time as its parent's window, but this is not a required procedure.

The following program statements show how to construct a status bar and insert it at the bottom of the window.

```
TStatusBar* sb = new TStatusBar(0, TGadget_Recessed,
   TStatusBar_CapsLock | TStatusBar_NumLock | TStatusBar_Overtype);
frame->Insert(*sb, TDecoratedFrame::Bottom);
MainWindow = frame;
```

See the MDIFILE.CPP sample program on your distribution disk for an example of how to create a window with a status bar.

Type definitions

TModeIndicator

enum TModeIndicator {ExtendSelection = 1, CapsLock = 1 << 1, NumLock = 1 << 2, ScrollLock = 1 << 3, Overtype = 1 << 4, RecordingMacro = 1 << 5};

Enumerates the keyboard modes. By default, these are arranged horizontally on the status bar from left to right. Sets the extended selection, ***CapsLock***, ***NumLock***, ***ScrollLock***, ***Overtype***, and recording macro indicators.

Public constructor

Constructor

```
TStatusBar(TWindow* parent = 0, TGadget_TBorderStyle borderStyle = TGadget_Recessed,
  uint modeIndicators = 0, TFont* font = new TGadgetWindowFont, TModule* module = 0);
```

Constructs a *TStatusBar* object in the *parent* window and creates any new gadgets and mode indicator gadgets. Sets *BorderStyle* to *borderStyle*, *ModeIndicators* to *modeIndicators*, and *NumModeIndicators* to 0. *borderStyle* can be any one of the value of *theBorderStyle* enum (for example, plain, raised, recessed, or embossed). The parameter mode indicators can be one of the values of the *TModeIndicator* enum, such as CapsLock, NumLock, ScrollLock, or Overtype. The parameter *font* points to a font object that contains the type of fonts used for the gadget window. The parameter, *module*, which defaults to 0, is passed to the base *TWindow's* constructor in the *module* parameter. Sets the values of the margins and borders depending on whether the gadget is raised, recessed, or plain.

Public member functions

GetModeIndicator

```
bool GetModeIndicator(TModeIndicator i) const;
```

Returns the current status bar mode indicator.

Insert

```
void Insert(TGadget& gadget, TPlacement = After, TGadget* sibling = 0);
```

Inserts the gadget (objects derived from class *TGadget*) in the status bar. By default, the new gadget is placed just after any existing gadgets and to the left of the status mode indicators. For example, you can insert a painting tool or a happy face that activates a recorded macro.

operator[]

```
TGadget* operator[](uint index);
```

Returns a gadget at a given index, but cannot access mode indicator gadgets.

SetModeIndicator

```
void SetModeIndicator(TModeIndicator, bool state);
```

Sets *TModeIndicator* to a given text gadget and set the status (on, by default) of the mode indicator. For the mode indicator to appear on the status bar, you must specify the mode when the window is constructed.

See also TStatusBar::TModeIndicator

SetSpacing

```
void SetSpacing(TSpacing& spacing);
```

Uses the *TSpacing* values to set the spacing to be used between mode indicator gadgets. *TSpacing* sets the status-bar margins in layout units. Typically, the message indicator (the leftmost text gadget) is left-aligned on the status bar and the other indicators are right-aligned. See *TLayoutMetrics* for an detailed explanation of layout units and constraints.

```
struct TSpacing {
```

```
    TMargins::TUnits  Units;
    int Value;
    TSpacing() {Units = TMargins::LayoutUnits; Value = 0;}
};
```

See also TStatusBar::TModeIndicator

ToggleModeIndicator

void ToggleModeIndicator(TModeIndicator);

Toggles the *ModeIndicator*.

Protected data members

BorderStyle

TGadget::TBorderStyle BorderStyle;

One of the enumerated border styles—none, plain, raised, recessed, or embossed—used by the mode indicators on the status bar.

ModeIndicators

uint ModeIndicators;

The *ModeIndicators* bit field indicates which mode indicators have been created for the status bar.

ModeIndicatorState

uint ModeIndicatorState;

Specifies the mode of the status bar. This can be any one of the values of *TModeIndicator* enum such as CapsLock, NumLock, ScrollLock, Overtype, *RecordingMacro*, or *ExtendSelection*.

NumModeIndicators

uint NumModeIndicators;

Specifies the number of mode indicators, which can range from 1 to 5.

Spacing

TSpacing Spacing;

Specifies the spacing between mode indicators on the status bar.

Protected member functions

IdleAction

bool IdleAction(long);

If more than one application is running, *TStatusBar* calls *IdleAction* instead of *PreProcessMsg* to check the state of the NumLock, CapsLock, or ScrollLock keys and to update the mode indicators if they do not reflect the current status of these keys.

See also TStatusBar::PreProcessMsg

PositionGadget

void PositionGadget(TGadget* previous, TGadget* next, TPoint& point);

Determines the position of the new gadget in relation to any previously existing gadgets and uses the *Pixels*, *LayoutUnits*, and *BorderUnits* fields of *TMargins* to determine the amount of spacing to leave between the mode indicators.

PreProcessMsg

bool PreProcessMsg(MSG& msg);

Overrides *TWindow::PreProcessMsg* to process keyboard messages and update the mode indicators when the NumLock, CapsLock, or ScrollLock keys are pressed.

See also TStatusBar::IdleAction

TStorageDocument class

stgdoc.h

Derived from *TDocument*, *TStorageDocument* supplies functionality that supports OLE's compound file structure. A compound file structure is a file-management system that stores files in a hierarchical structure within a root file. This storage structure is analagous to the directory or folder and file scheme used on a disk, except that a directory is called a storage and a file is called a stream.

In addition, *TStorageDocument* provides support for OLE's compound document mechanism. A compound document can store many different kinds of embedded objects, for example, spreadsheets as well as bitmaps.

Basically, *TStorageDocument* supports having a document read and write its own storage. In order to provide this functionality, *TStorageDocument* overrides several virtual methods from *TDocument* and, following *TDocument*'s strategy, also applies property lists both to documents and to their views. In this way, documents can use these attributes to read files in from storage and write files out to storage

Messages are sent to the application, which queries the properties in order to determine how to process the document or view. Each derived class must implement its own property attribute types-either string or binary data.

See also

TDocument, TOleDocument

Type definitions

TStgDocProp

enum TStgDocProp {PrevProperty, CreateTime, ModifyTime, AccessTime, StorageSize, IStorageInstance, NextProperty};

Enumerates the following constants that define the document's properties:

Constant	Meaning
PrevProperty	Should always have a value of *TDocument::NextProperty* – 1. This is the initial value in any document's property list.
CreateTime	The time the file is created.
ModifyTime	The time the file is modified.

Constant	Meaning
AccessTime	The last time the file was accessed.
StorageSize	The size of the storage buffer.
IStorageInstance	The storage instance for the document.
NextProperty	The final value for any document's list of properties.

Public constructor and destructor

Constructor

Constructor TStorageDocument(TDocument* parent = 0): TDocument(parent), StorageI(0), OpenCount(0), CanRelease(false);

Constructs a *TStorageDocument* object with the specified *TDocument* parent window. Sets the data members *StorageI* and *OpenCount* to 0. Sets *CanRelease* to **false**. Later, the member function *ReleaseDoc* sets this flag to **true** so that the document can be closed.

Destructor

~TStorageDocument();

Destroys a *TStorageDocument* object.

Public member functions

Close

bool Close();

Releases the *IStorage* if *CanRelease* is **true**. (*CanRelease* is set to **true** when *ReleaseDoc* is called.) Before closing the document, *Close* checks any child documents and tries to close them.

See also TStorageDocument::Open, TStorageDocument::ReleaseDoc, TOutStream, ofxxxx document open enum

Commit

bool Commit(bool force = false);

Saves the current data to storage. When a file is closed, the document manager calls either *Commit* or *Revert*. If *force* is **true**, all data is written to storage. *TDocument*'s *Commit* checks any child documents and commits their changes to storage also. Before the current data is saved, all child documents must return **true**. If all child documents return **true**, *Commit* flushes the views for any operations that occurred since the last time the view was checked. Once all data for the document object is updated and saved, *Commit* returns **true**.

CommitTransactedStorage

bool CommitTransactedStorage();

If a file is opened or created in transacted mode, call *CommitTransactedStorage* to commit a document to permanent storage. By default, a document uses transacted instead of direct storage. With transacted storage, a document written to *IStorage* is only temporary until it is committed permanently. If a compound file is opened or created in direct mode, then *CommitTransactedStorage* does not need to be called.

FindProperty

```
int FindProperty(const char far* name);
```

Gets the property index, given the property name. Returns the integer index number that corresponds to the name. If the name isn't found in the list of properties, returns 0.

See also pfxxxx property attribute constants

GetProperty

```
int GetProperty(int index, void far* dest, int textlen=0);
```

Returns the total number of properties for this storage document, where *index* is the property index, *dest* contains the property data, and *textlen* is the size of the property array. If *textlen* is 0, the property data is returned as binary data; otherwise, the property data is returned as text data.

See also pfxxxx property attribute constants, TStorageDocument::SetProperty

GetStorage

```
IStorage* GetStorage();
```

Returns the document's root *IStorage*.

See also TStorageDocument::SetStorage

InStream

```
TInStream* InStream(int omode, const char far* strmId=0);
```

Provides an input stream for the *TStorageDocument* object. Returns a pointer to a *TInStream* object. The parameter *omode* contains a combination of the document open and sharing modes (for example, *ofReadWrite*) defined in docview.h. The *strmId* parameter is used for documents that support named streams.

See also TStorageDocument::OutStream, TInStream, ofxxxx document open enum

IsOpen

```
bool IsOpen();
```

Checks to see if the storage document has any *IStorage* created. If there is no storage, *IsOpen* returns **false**; otherwise, it returns **true**.

Open

```
bool Open(int omode, const char far* name);
```

Opens or creates a document based on IStorage. The *name* parameter specifies the name of the document, if any, to open. The *omode* parameter contains a combination of the document open and sharing modes (for example, *ofReadWrite*) defined in docview.h.

See also TStorageDocument::Close, TOutStream, ofxxxx document open enum

OpenHandle

```
virtual bool OpenHandle(int omode, HANDLE hGlobal);
```

OpenHandle writes data to a memory block. *OpenHandle* first creates an *ILockBytes* (an OLE2 interface that implements reading, writing, and locking a series of bytes in a file) interface on the global handle and then creates an *IStorage* based on the *ILockBytes* interface. The parameter *omode* contains a combination of the document open and sharing modes (for example, *ofReadWrite*) defined in docview.h.

See also ofxxxx document open enum

OutStream

TOutStream* OutStream(int omode, const char far* strmId=0);

Provides an output stream for a particular storage medium. Returns a pointer to a *TOutStream*. The parameter *omode* contains a combination of the document open and sharing modes (for example, *ofReadWrite*) defined in docview.h. The *strmId* parameter is used for documents that support named streams.

See also TStorageDocument::InStream, TOutStream, ofxxxx document open enum

PropertyCount

int PropertyCount();

Gets the total number of properties for the *TStorageDocument* object. Returns *NextProperty* –1.

See also pfxxxx property attribute constants

PropertyFlags

int PropertyFlags(int index);

Returns the attributes of a specified property given the index of the property whose attributes you want to retrieve.

See also pfxxxx property attribute constants

PropertyName

const char* PropertyName(int index);

Returns the name of the property given the index value.

See also pfxxxx property attribute constants

ReleaseDoc

virtual bool ReleaseDoc();

Releases the storage for the document and closes the document.

Revert

bool Revert(bool clear = false);

Performs the reverse of *Commit* and cancels any changes made to the storage document since the last commit. If *clear* is **true**, data is not reloaded for views. Revert also checks all child documents and cancels any changes if all children return **true**. When a file is closed, the document manager calls either *Commit* or *Revert*. *Revert* returns **true** if the revert operation is successful.

SetDocPath

bool SetDocPath(const char far* path);

Sets the document path for the Open and Save file operations.

SetProperty

bool SetProperty(int index, const void far* src);

Sets the value of the property, given the index of the property, and *src*, the data type (either binary or text) to which the property must be set.

See also pfxxxx property attribute constants, TStorageDocument::GetProperty

SetStorage

IStorage* SetStorage(IStorage* stg);

Attaches an *IStorage* pointer (*stg*) to this document.

See also TStorageDocument::GetStorage, TStorageDocument::StorageI

Protected data members

ThisOpen

int ThisOpen;

Holds the actual mode bits used for opening the storage. The mode bits determine how the file is opened: for example, read only, read and write, and so on.

See also ofxxxx document open enum

StorageI

IStorage* StorageI;

Holds the current *IStorage* instance. (If no storage is open, *StorageI* is 0.)

See also TStorageDocument::GetStorage, TStorageDocument::SetStorage

TStream class

docview.h

An abstract base class, *TStream* provides links between streams and documents, views, and document files.

Public destructor

Destructor

~TStream();

Closes the stream. Derived classes generally close the document if it was opened especially for this stream.

Public member functions

GetDocument

TDocument& GetDocument();

Returns the current document open for streaming.

GetOpenMode

int GetOpenMode();

Gets mode flags used when opening document streams. For example, the stream can be opened in *ofRead* mode to allow reading, but not changing (writing to) the file.

See also ofxxxx document open enum

GetStreamName

const char far* GetStreamName();

Gets the name of the stream used for opening the document.

Protected data members

Doc
TDocument& Doc;
Stores the document that owns this stream.

NextStream
TStream* NextStream;
Points to the next stream in the list of active streams.

Protected constructor

Constructor
TStream(TDocument& doc, const char far* name,int mode);
Constructs a *TStream* object. *doc* refers to the document object, *name* is the user-defined name of the stream, and *mode* is the mode used for opening the stream.

See also TInStream, TOutStream, ofXXXX document open enum, shXXXX document sharing enum

TStringLookupValidator class

validate.h

Derived from *TLookupValidator*, *TStringLookupValidator* is a streamable class. A *TStringLookupValidator* object verifies the data in its associated edit control by searching through a collection of valid strings. You can use string-lookup validators when your edit control needs to accept only members of a certain set of strings.

Public constructor and destructor

Constructor
TStringLookupValidator(TSortedStringArray* strings);
Constructs a string-lookup object by first calling the constructor inherited from *TLookupValidator* and then setting *Strings* to *strings*.

Destructor
~TStringLookupValidator();
This destructor disposes of a list of valid strings by calling *NewStringList* and then disposes of the string-lookup validator object by calling the Destructor inherited from *TLookupValidator*.

Public member functions

Error
void Error();

Overrides *TValidator's* virtual function and displays a message box indicating that the typed string does not match an entry in the string list.

See also TValidator::Error

Lookup

bool Lookup(const char far* str);

Overrides *TLookupValidator's* virtual function. Returns true if the string passed in *str* matches any of the *strings*. Uses the search method of the string collection to determine if str is present.

See also TLookupValidator::Lookup

NewStringList

void NewStringList(TSortedStringArray* strings);

Sets the list of valid input string for the string-lookup validator. Disposes of any existing string list and then sets *Strings* to *strings*.

Protected data member

Strings

TSortedStringArray* Strings;

Points to a string collection containing all the valid strings the user can type. If *Strings* is NULL, all input is validated.

TSystemMenu class — menu.h

TSystemMenu creates a system menu object that then becomes the existing system menu.

Public constructor

Constructor

TSystemMenu(HWND wnd, bool revert = false);

Constructs a system menu object. If *revert* is true, then the menu created is a default system menu. Otherwise, it is the menu currently in the window.

See also TPopupMenu::TPopupMenu

TTextGadget class — textgadg.h

Derived from *TGadget*, *TTextGadget* is a text gadget object. When you construct a text gadget, you must specify how many characters you want to reserve space for and how the text should be aligned horizontally. The inner boundaries of the text gadget are computed by multiplying the number of characters by the maximum character width.

Public constructor and destructor

Constructor

TTextGadget(int id = 0, TBorderStyle = Recessed, TAlign = Left, uint numChars = 10, const char* text = 0);

Constructs a *TTextGadget* object with the specified ID, border style, and alignment. Sets *Margins.Left* and *Margins.Right* to 2. Sets *Text* and *TextLen* to 0.

Destructor

~TTextGadget();

Destroys a *TTextGadget* object.

Public member functions

GetText

char* GetText();

Returns the text for the gadget.

SetText

void SetText(const char* text);

If the text stored in *Text* is not the same as the new text, *SetText* deletes the text stored in *Text*. Then, it sets *TextLen* to the length of the new string. If no text exists, it sets both *Text* and *TextLen* to 0 and then calls *Invalidate* to invalidate the rectangle.

Protected data members

Align

TAlign Align;

Text alignment attribute—left-, center-, or right-aligned.

NumChars

uint NumChars;

Holds the number of text characters.

Text

char* Text;

Points to the text for the gadget.

TextLen

uint TextLen;

Stores the length of the text.

Protected member functions

GetDesiredSize

void GetDesiredSize(TSize &size);

If shrink-wrapping is requested, *GetDesiredSize* returns the size needed to accommodate the borders, margins, and text; otherwise, if shrink-wrapping is not requested, it returns the gadget's current width and height.

See also TGadget::GetDesiredSize

Invalidate

void Invalidate();

Calls *TGadget::GetInnerRect* to compute the area of the text for the gadget and then *TGadget::InvalidateRect* to invalidate the rectangle in the parent window.

See also TGadget::GetInnerRect, TGadget::Invalidate

Paint

void Paint(TDC& dc);

Calls *TGadget::PaintBorder* to paint the border. Calls *TGadget::GetInnerRect* to calculate the area of the text gadget's rectangle. If the text is left-aligned, *Paint* calls *dc.GetTextExtent* to compute the width and height of a line of the text. To set the background color, *Paint* calls *dc.GetSysColor* and sets the default background color to face shading (COLOR_BTNFACE). To set the button text color, *Paint* calls *dc.SetTextColor* and sets the default button text color to COLOR_BTNTEXT. To draw the text, *Paint* calls *dc.ExtTextOut* and passes the parameters ETO_CLIPPED (so the text is clipped to fit the rectangle) and ETO_OPAQUE (so the rectangle is filled with the current background color).

See also TGadget::Paint

Type definitions — gadgetwi.h

TTileDirection

enum TTileDirection{Horizontal, Vertical};

Enumerates the two directions the gadget can be tiled. *TGadgetWindow::TileGadgets* actually tiles the gadgets in the direction requested.

See also TGadgetWindow::Direction, TGadgetWindow::TileGadgets

TAlign

enum TAlign {Left, Center, Right};

Enumerates the text-alignment attributes. *Left* aligns the text at the left edge of the bounding rectangle. *Right* aligns the text at the right edge of the bounding rectangle. *Center* aligns the text horizontally at the center of the bounding rectangle.

TTinyCaption class — tinycapt.h

Derived from *TWindow*, *TTinyCaption* is a mix-in class that handles a set of non-client events to produce a smaller caption bar for a window. Whenever it displays the caption bar, *TTinyCaption* checks the window style and handles the WS_SYSMENU, WS_MINIMIZEBOX, WS_MAXIMIZEBOX display attributes. Thus, you can use *TTinyCaption* to set the attributes of the tiny caption bar before enabling the caption. For example,

```
Attr.Style = WS_POPUP | WS_BORDER | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX;
```

TTinyCaption provides functions that let you manipulate frame types, border styles, and menus. You can adjust the height of the caption bar or accept the default height, which is about one-half the height of a standard caption bar. If you set *CloseBox* to **true**, then the window will close when you click the close box instead of displaying the system menu.

The sample program OWLCMD.CPP on your distribution disk displays the following tiny caption bar:

If you are using *TTinyCaption* as a mix-in class that does partial event handling, call the *DoXxxx* function in the mix-in class (instead of the *EvXxxx* function) to avoid duplicating default processing. The following example from OWLCMD.CPP (a sample program on your distribution disk) illustrates this process:

```
void TMyFrame::EvSysCommand(uint cmdType,TPoint& p)
{
   if (TTinyCaption::DoSysCommand(cmdType, p) == esPartial)
   FrameWindow::EvSysCommand(cmdType, p);
```

The *TFLoatingFrame* class can be used with *TTinyCaption* to produce a close box. See the sample programs OWLCMD.CPP and MDIFILE.CPP on your distribution disk for examples of how to use *TTinyCaption*.

Protected data members

Border
TSize Border;
Thin frame border size for dividers.

CaptionFont
TFont* CaptionFont;
Font used for the text in the tiny caption bar.

CaptionHeight
int CaptionHeight;
Height of the caption bar.

CloseBox
bool CloseBox;
If **true**, the window will close when the close box is clicked.

DownHit
uint DownHit;
Location of mouse-button press or cursor move.

Frame
TSize Frame;

Actual left and right, top and bottom dimensions of the caption bar.

IsPressed

bool IsPressed;

Is **true** if a mouse button is pressed.

TCEnabled

bool TCEnabled;

Is **true** if the tiny caption bar is displayed.

WaitingForSysCmd

bool WaitingForSysCmd;

Is **true** if *TTinyCaption* is ready to receive system messages.

Protected constructor and destructor

Constructor

TTinyCaption();

Constructs a *TTinyCaption* object attached to the given parent window. Initializes the caption font to 0 and TCEnabled to false so that the tiny caption bar is not displayed automatically.

Destructor

~TTinyCaption();

Destroys a *TTinyCaption* object and deletes the caption font.

Protected member functions

DoCommand

TEventStatus DoCommand(uint id, HWND hWndCtl, uint notifyCode, LRESULT& evRes);

Displays the system menu using *::TrackPopup* so that *TTinyCaption* sends WM_COMMAND instead of WM_SYSCOMMAND messages. If a system menu command is received, it's then transformed into a WM_SYSCOMMAND message. If the tiny caption bar is **false**, *DoCommand* returns *esPartial*.

See also TTinyCaption::EvCommand, TEventStatus enum

DoLButtonUp

TEventStatus DoLButtonUp(uint hitTest, TPoint& screenPt);

Releases the mouse capture if the caption bar is enabled and a mouse button is pressed. Sets *hitTest*, indicating the mouse button has been pressed. Captures the mouse message and repaints the smaller buttons before returning *esComplete*.

See also TTinyCaption::EvLButtonUp

DoMouseMove

TEventStatus DoMouseMove(uint hitTest, TPoint& screenPt);

Returns *TEventStatus*.

DoNCActivate

TEventStatus DoNCActivate(bool active, bool& evRes);

If the tiny caption is not enabled or is iconic, returns *esPartial*. Otherwise, repaints the caption as an active caption and returns *esComplete*.

See also TTinyCaption::EvNCActivate

DoNCCalcSize

TEventStatus DoNCCalcSize(bool calcValidRects, NCCALCSIZE_PARAMS far&calcSize, uint& evRes);

If the caption bar is not enabled or is iconic, returns *esPartial*. Otherwise, calculates the dimensions of the caption and returns *esComplete*.

See also TTinyCaption::EvNCCalcSize

DoNCHitTest

TEventStatus DoNCHitTest(TPoint& screenPt, uint& evRes);

If the caption bar is not enabled, returns *esPartial*. Otherwise, sends a message to the caption bar that the mouse or the cursor has moved, and returns *esComplete*.

See also TTinyCaption::EvNCHitTest

DoNCLButtonDown

TEventStatus DoNCLButtonDown(uint hitTest, TPoint& screenPt);

If the caption bar isn't enabled, returns *esPartial*. Otherwise, determines if the user released the button outside or inside a menu, and returns *esComplete*.

See also TTinyCaption::EvNCLButtonDown

DoNCPaint

TEventStatus DoNCPaint();

If the caption bar isn't enabled or is iconized, returns *esPartial*. Otherwise, gets the focus, paints the caption, and returns *esPartial*, thus indicating that a separate paint function must be called to paint the borders of the caption.

See also TTinyCaption::EvNCPaint

DoSysCommand

TEventStatus DoSysCommand(uint cmdType, TPoint& p);

If the caption bar isn't enabled, returns *esPartial*. If the caption bar is iconized and the user clicks the icon, calls *DoSysMenu* to display the menu in its normal mode and returns *esComplete*.

See also TTinyCaption::EvSysCommand

DoSysMenu

void DoSysMenu;

Gets the system menu and sets up menu items. *DoSysMenu* is also responsible for displaying and tracking the status of the menu.

EnableTinyCaption

void EnableTinyCaption(int ch=58, bool closeBox=false);

Activates the tiny caption bar. By default, *EnableTinyCaption* replaces the system window with a tiny caption window that doesn't close when the system window is clicked. If the *closeBox* argument is **true**, clicking on the system menu will close the

window instead of bringing up the menu. You can use *EnableTinyCaption* to hide the window if you are using a tiny caption in a derived class. To diminish the tiny caption bar, try the following values:

```
EnableTinyCaption(30, true);
```

Or, to maximize the tiny caption bar, use these values:

```
EnableTinyCaption(48, true);
```

EvCommand

LRESULT EvCommand(uint id, HWND hWndCtl, uint notifyCode);

EvCommand provides extra processing for commands, but lets the focus window and its parent windows handle the command first.

See also TTinyCaption::DoCommand

EvLButtonUp

void EvLButtonUp(uint hitTest, TPoint& screenPt);

Responds to a mouse button-up message by calling *DoLButtonUp*. If *DoLButtonUp* doesn't return *IsComplete*, *EvLButtonUp* calls *TWindow::EvLButtonUp*.

See also TTinyCaption::DoLButtonUp

EvMouseMove

void EvMouseMove(uint hitTest, TPoint& screenPt);

Responds to a mouse-move message by calling *DoMouseMove*. If *DoMouseMove* doesn't return *IsComplete*, *EvMouseMove* calls *TWindow::EvMouseMove*.

See also TTinyCaption::DoMouseMove

EvNCActivate

bool EvNCActivate(bool active);

Responds to a request to change a title bar or icon by calling *DoNCActivate*. If *DoNCActivate* doesn't return *esComplete*, *EvNCActivate* calls *TWindow::EvNCActivate*.

See also TTinyCaption::DoNCActivate

EvNCCalcSize

uint EvNCCalcSize(bool calcValidRects, NCCALCSIZE_PARAMS far& calcSize);

Responds to a request to change a title bar or icon by calling *DoNCActivate*. If *DoNCActivate* doesn't return *esComplete*, *EvNCActivate* calls *TWindow::EvNCActivate*.

Calculates the size of the command window including the caption and border so that it can fit within the window.

See also TTinyCaption::DoNCActivate

EvNCHitTest

uint EvNCHitTest(TPoint& screenPt);

Responds to a cursor move or press of a mouse button by calling *DoNCHitTest*. If *DoNCHitTest* doesn't return *esComplete*, *EvNCHitTest* calls *TWindow::EvNCHitTest*.

See also TTinyCaption::DoNCHitTest

EvNCLButtonDown

void EvNCLButtonDown(uint hitTest, TPoint& screenPt);

Responds to a press of the left mouse button while the cursor is within the nonclient area of the caption bar by calling *DoNCLButtonDown*. If *DoNCLButtonDown* doesn't return *esComplete, EvNCLButtonDown* calls *TWindow::EvNCLButtonDown*.

See also TTinyCaption::DoNCLButtonDown

EvNCPaint

void EvNCPaint();

Responds to a request to change a title bar or icon. Paints the indicated device context or display screen and does any special painting required for the caption.

See also TTinyCaption::DoNCActivate

EvSysCommand

void EvSysCommand(uint cmdType, TPoint& p);

Responds to a WM_SYSCOMMAND message by calling *DoSysCommand*. If *DoSysCommand* returns *esPartial, EvSysCommand* calls *TWindow::EvSysCommand*.

See also TTinyCaption::DoSysCommand

GetCaptionRect

TRect GetCaptionRect();

Gets the area of the caption for changing or repainting.

See also TTinyCaption::PaintCaption

GetMaxBoxRect

TRect GetMaxBoxRect();

Returns the size of the maximize box rectangle.

See also TTinyCaption::PaintMaxBoxRect

GetMinBoxRect

TRect GetMinBoxRect();

Returns the size of the minimize box rectangle.

See also TTinyCaption::PaintMinBoxRect

GetSysBoxRect

TRect GetSysBoxRect();

Returns the size of the system box rectangle.

See also TTinyCaption::PaintSysBoxRect

PaintButton

void PaintButton(TDC& dc, TRect& boxRect, bool pressed);

Paints a blank button.

PaintCaption

void PaintCaption(bool active);

Calls *dc.SelectObject* to select the given rectangle and *dc.PatBlt* to paint the tiny caption bar using the currently selected brush for this device context.

See also TDC::SelectObject, TDC::PatBlt

PaintCloseBox

void PaintCloseBox(TDC& dc, TRect& boxRect, bool pressed);

Paints a close box on the tiny caption bar. You can override the default box if you want to design your own close box.

See also TTinyCaption::GetSysBoxRect

PaintMaxBox

void PaintMaxBox(TDC& dc, TRect& boxRect, bool pressed);

Paints a maximize box on the tiny caption bar.

See also TTinyCaption::GetMaxBoxRect

PaintMinBox

void PaintMinBox(TDC& dc, TRect& boxRect, bool pressed);

Paints a minimize box on the tiny caption bar.

See also TTinyCaption::GetMinBoxRect

PaintSysBox

void PaintSysBox(TDC& dc, TRect& boxRect, bool pressed);

Paints the system box.

See also TTinyCaption::GetSysBoxRect

Response table entries

Response table entry	Member function
EV_WM_NCACTIVATE	EvNCActivate
EV_WM_NCCALCSIZE	EvNcCalcSize
EV_WM_NCHITTEST	EvNcHitTest
EV_WM_NCPAINT	EvNcPaint
EV_WM_NCLBUTTONDOWN	EvNclButtonDown
EV_WM_LBUTTONUP	EvLButtonUp
EV_WM_MOUSEMOVE	EvMouseMove
EV_WM_SYSCOMMAND	EvSysCommand

TToolBox class

toolbox.h

Derived from *TGadgetWindow*, *TToolBox* arranges gadgets in a matrix in which all columns are the same width (as wide as the widest gadget) and all rows are the same height (as high as the highest gadget).

You can specify exactly how many rows and columns you want for your toolbox, or you can let *TToolbox* calculate the number of columns and rows you need. If you specify AS_MANY_AS_NEEDED, the *TToolBox* calculates how many rows or columns are

needed based on the opposite dimension. For example, if there are twenty gadgets, and you requested four columns, your matrix would have five rows.

Public constructor

Constructor

```
TToolBox(TWindow* parent, int numColumns = 2, int numRows = AS_MANY_AS_NEEDED,
  TTileDirection direction = Horizontal, TModule* module = 0);
```

Constructs a *TToolBox* object with the specified number of columns and rows and tiling direction. Overlaps the borders of the toolbox with those of the gadget and sets *ShrinkWrapWidth* to true.

Public member functions

GetDesiredSize

```
void GetDesiredSize(TSize& size);
```

Overrides *TGadget*'s *GetDesiredSize* function and computes the size of the cell by calling *GetMargins* to get the margins.

See also TGadgetWindow::GetDesiredSize

Insert

```
void Insert(TGadget& gadget, TPlacement = After, TGadget* sibling = 0);
```

Overrides *TGadget*'s *Insert* function and tells the button not to notch its corners.

See also TGadgetWindow::Insert

LayoutSession

```
void LayoutSession();
```

Called when a change occurs in the size of the margins of the toolbox or size of the gadgets, *LayoutSession* gets the desired size and moves the window to adjust to the desired change in size.

SetDirection

```
virtual void SetDirection(TTileDirection direction);
```

Sets the direction of the tiling—either horizontal or vertical.

Protected data members

NumColumns

```
int NumColumns;
```

Contains the number of columns for the toolbox.

NumRows

```
int NumRows;
```

Contains the number of rows for the toolbox.

Protected member function

TileGadgets
TRect TileGadgets;
Tiles the gadgets in the direction requested (horizontal or vertical). Derived classes can adjust the spacing between gadgets.

See also TGadgetWindow::TileGadget

TTransferDirection enum — window.h

enum TTransferDirection {tdGetData, tdSetData, tdSizeData};
TTransferDirection enum describes the following constants, which the transfer function uses to determine how to transfer data to and from the transfer buffer.

Constant	Meaning
tdGetData	Retrieve data from the class.
tdSetData	Send data to the class.
tdSizeData	Return the size of data transferred by the class.

See also TWindow::Transfer, TWindow::TransferData

TUIHandle class — uihandle .h

TUIHandle manages and draws various kinds of UI handles, including hatched border handles, and resizing grapples (small squares that appear along the edges) on a rectangle. You can use this class to create a hatched border that encloses various kinds of drawing objects you want to manipulate.

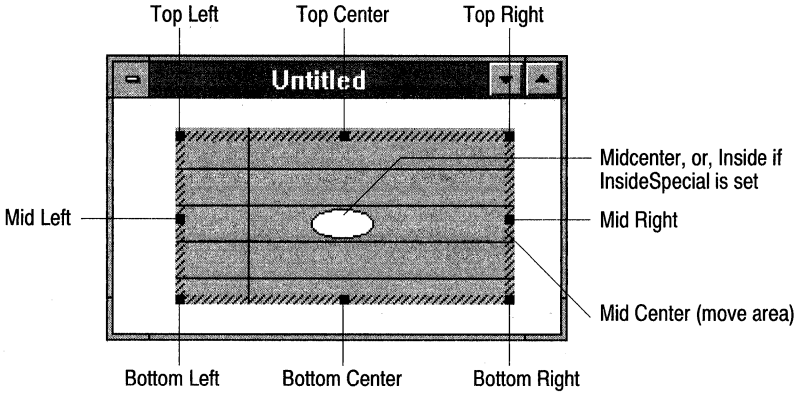
With the help of this class, you can create an application that lets you:

- Resize the shape of the rectangle by pointing to and grabbing one of the grapples on the border.
- Move the entire rectangle by clicking in the middle of the rectangle.

Although, by default, a hatched border with eight grapples is created, you can vary the existence of the grapples as well as the pattern of the border. You can also draw a dashed frame enclosing a rectangle or draw a rectangle filled with hatch marks. *THatch8x8Brush* contains brushes you can use to draw the hatched border.

TUIHandle uses the **enum** *TWhere* to return the area where the user points and clicks the mouse (referred to as a *hit area*).

The following diagram displays a UI handle and identifies several small square grapples where hit testing occurs.



The following code from SAMPCONT.CPP sets up a hatched border and UI grapples for an OLE2 container application.

```
// Do the default rectangle painting.
     TRect r // Insert your rectangle drawing code here.
// Draw 8 grapples with a border on top of the object.
      TUIHandle handle(r, TUIHandle::HandlesIn|TUIHandle::Grapples
      |TUIHandle::HatchBorder, 5);
     handle.Paint(dc);
// Insert your code here....
// Draw a hatched border.
     TUIHandle handle(r, TUIHandle::HatchBorder, 5);
     handle.Paint(dc);
```

See also

THatch8x8Brush

Type definitions

TStyle

enum TStyle{HandlesIn, HandlesOut, Framed, DashFramed, Grapples, HatchBorder, HatchRect, InsideSpecial};

Enumerates the style of the border around the rectangle with combinations of the following styles:

Value	Description
Select one of the following styles:	
HandlesIn	Draw handles inside the rectangle
HandlesOut	Draw handles outside the rectangle
Select zero or one of the following styles:	
Framed	Draw a solid frame around the rectangle
DashFramed	Draw a dashed frame around the rectangle
Grapples	Draw eight grapples for resizing

Value	Description
Select zero or one of the following styles:	
HatchBorder	Draw a hatched handle for moving the border
HatchRect	Draw the entire rectangle filled in with hatch marks
InsideSpecial	Treat the inside hit area in a special way

If a hatched border with grapples is drawn inside a rectangle, it sits within the borders of the outer frame of the rectangle. If a hatched border with grapples is drawn outside the rectangle, it is drawn outside the boundary of the rectangle's frame. In the latter case, the function *GetBoundingRect* returns a larger rectangle.

TWhere

```
enum TWhere {TopLeft, TopCenter, TopRight, MidLeft, MidCenter, MidRight, BottomLeft, BottomCenter,
  BottomRight, Outside, Inside};
```

TWhere indicates which one of the grapples or handle was hit. *TWhere*can contain one of the following values:

Value	Description
TopLeft	The grapple in the top left corner of the border was hit.
TopCenter	The grapple in the top center of the border was hit.
TopRight	The grapple in the top left right corner of the border was hit.
MidLeft	The grapple in the middle of the left border was hit.
MidCenter	The area in the middle of the rectangle was hit. This indicates that either the hatched frame was hit or inside, non-special area was hit.
MidRight	The grapple in the middle of the right border was hit.
BottomLeft	The grapple in the left corner of the bottom border was hit.
BottomCenter	The grapple in the center of the bottom border was hit.
BottomRight	The grapple in the right corner of the bottom border was hit.
Outside	The hit area is completely outside the object.
Inside	The hit area for the grapple is inside the object and *InsideSpecial* is set.

The *InsideSpecial* designation refers to the area inside the rectangle when the hit area needs to be treated specially because it might contain text or graphics, for example. Normally, if the area inside the rectangle is hit, it means that user wants to move the rectangle. However, if there is text inside the rectangle, the user might click on this area in order to enter text. This latter situation is referred to as an *inside special* case.

The hit area (Where) can be converted to a row and a column by using the following equations:

Row = Where / 3

Column = Where mod 3

The value of *Where* ranges from 0 (*TopLeft*) to 8 (*BottomRight*) and corresponds to the following areas of a rectangle:

		Column 0	1	2
	0	TopL	TopC	TopR
Row	1	MidL	MidC	MidR
	2	BottomL	BottomC	BottomR

You can then use these values to calculate the movement of the object and to resize the object.

Public constructor

Constructor

TUIHandle(const TRect& frame, uint style = HandlesIn|Grapples|HatchBorder, int thickness = 5);

Constructs a *TUIHandle* object for the specified frame, with eight grapples drawn in a hatched border and a default thickness of 5 pixels drawn to the inside.

Public member functions

GetBoundingRect

GetBoundingRect()const;

GetBoundingRect returns a rectangle with the size adjusted according to the thickness. For example, if the handles are outside the rectangle, *GetboundingRect* returns a larger rectangle. The enum . *TStyle* defines the positions of the handles, that is, whether the handles are defined as *HandlesIn* or *HandlesOut*.

GetCursorId

static uint16 GetCursorId(TWhere where);

Returns the lD of a standard cursor that's appropriate for use over the location specified in the *where* parameter.

HitTest

TWhere HitTest(const TPoint& point)const;

Compares a given point (*point*) to various parts of the rectangle. If the hit was outside the rectangle, *HitTest* returns *Outside*. If the hatched border handle of the rectangle was hit, returns *MidCenter* (inside). For any other hits, *HitTest* returns the location of the grapple that was hit. The enum *TWhere* defines the possible hit areas.

Move

void Move(int dx, int dy);

Moves the rectangle relative to the values specified in *dx* and *dy*.

See also TUIHandle::MoveTo

MoveTo

void MoveTo(int x, int y);

Moves the rectangle to the given *x* and *y* coordinates.

See also TUIHandle::Move

Paint

void Paint(TDC& dc) const;

Paints the *TUIHandle* object onto the specified device context, *dc*.

Size

void Size(int w, int h);

Sets the size of the rectangle according to the measurements specified in *w*, the width, and *h*, the height.

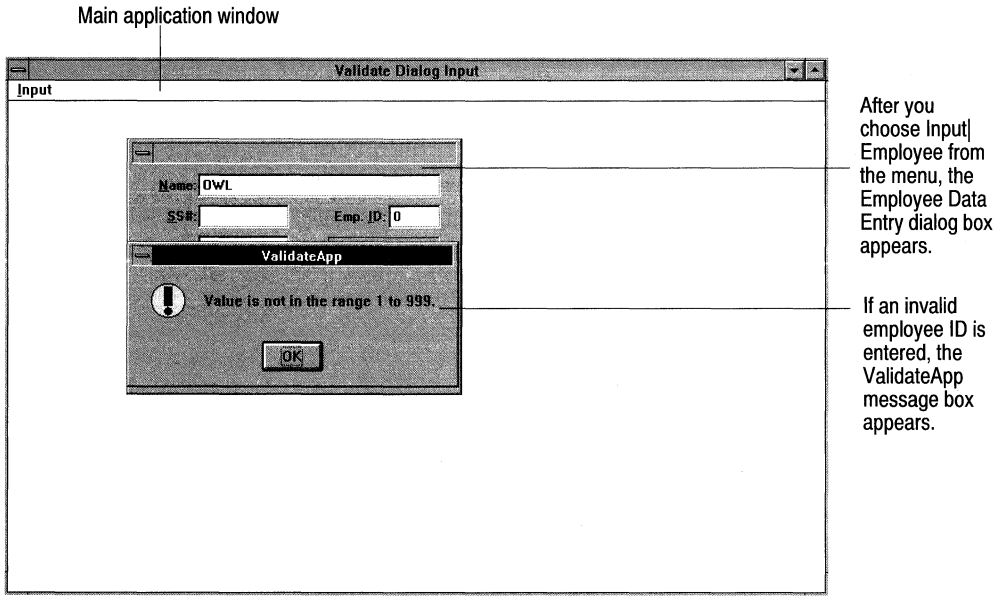
TValidator class — validate.h

A streamable class, *TValidator* defines an abstract data validation object. Although you will never actually create an instance of *TValidator*, it provides the abstract functions for the other data validation objects.

The VALIDATE.CPP sample program on your distribution disk derives *TValidateApp* from *TApplication* in the following manner:

```
class TValidateApp : public TApplication {
   public:
      TValidateApp() : TApplication("ValidateApp") {}
      void InitMainWindow() {
         MainWindow = new TTestWindow(0, "Validate Dialog Input");
}
```

and displays the following message box if the user enters an invalid employee ID:



Public constructor and destructor

Constructor

TValidator();

Constructs an abstract validator object and sets *Options* fields to 0.

Destructor

virtual ~TValidator();

Destroys an abstract validator object.

Public member functions

Error

virtual void Error();

Error is an abstract function called by *Valid* when it detects that the user has entered invalid information. By default, *TValidator::Error* does nothing, but derived classes can override *Error* to provide feedback to the user.

HasOption

bool HasOption(int option);

Gets the *Options* bits. Returns **true** if a specified option is set.

See also TValidator::Options, TValidatorOptions enum

IsValid

virtual bool IsValid(const char far* str);

By default, *TValidator::IsValid* returns **true**. Derived validator types can override *IsValid* to validate data for a completed edit control. If an edit control has an associated validator object, its *Valid* method calls the validator object's *Valid* method, which in turn calls *IsValid* to determine whether the contents of the edit control are valid.

See also TValidator::Valid

IsValidInput

virtual bool IsValidInput(char far* str, bool suppressFill);

If an edit control has an associated validator object, it calls *IsValidInput* after processing each keyboard event. This gives validators such as filter validators an opportunity to catch errors before the user fills the entire item or screen.

By default, *IsValidInput* returns **true**. Derived data validators can override *IsValidInput* to validate data as the user types it, returning **true** if *str* holds valid data and **false** otherwise.

str is the current input string. *suppressFill* determines whether the validator should automatically format the string before validating it. If *suppressFill* is **true**, validation takes place on the unmodified string *str*. If *suppressFill* is **false**, the validator should apply any filling or padding before validating data. Of the standard validator objects, only *TPXPictureValidator* checks *suppressFill*.

IsValidInput can modify the contents of the input string; for example, it can force characters to uppercase or insert literal characters from a format picture. *IsValidInput* should not, however, delete invalid characters from the string. By returning **false**, *IsValidInput* indicates that the edit control should erase the incorrect characters.

SetOption

void SetOption(int option);

Sets the bits for the *Options* data member.

See also TValidator::Options, TValidatorOptions enum

Transfer

virtual uint Transfer(char far* str, void* buffer, TTransferDirection direction);

Allows a validator to set and read the values of its associated edit control. This is primarily useful for validators that check non-string data, such as numeric values. For example, *TRangeValidator* uses *Transfer* to read and write values instead of transferring an entire string.

By default, edit controls with validators give the validator the first chance to respond to *DataSize*, *GetData*, and *SetData* by calling the validator's *Transfer* method. If *Transfer* returns anything other than 0, it indicates to the edit control that it has handled the appropriate transfer. The default action of *TValidator::Transfer* is to always return 0. If you want the validator to transfer data, you must override its *Transfer* method.

Transfer's first two parameters are the associated edit control's text string and the *tdGetData* or *tdSetData* data record. Depending on the value of *direction*, *Transfer* can set *str* from buffer or read the data from *str* into *buffer*. The return value is always the number of bytes transferred.

If direction is *tdSizeData*, *Transfer* doesn't change either *str* or *buffer*; it just returns the data size. If *direction* is *tdSetData*, *Transfer* reads the appropriate number of bytes from

buffer, converts them into the proper string form, and sets them into *str*, returning the number of bytes read. If *direction* is *tdGetData*, *Transfer* converts *str* into the appropriate data type and writes the value into *buffer*, returning the number of bytes written.

See also TTransferDirection enum

UnsetOption

void UnsetOption(int option);

Unsets the bits specified in the *Options* data member.

See also TValidator::Options, TValidatorOptions enum

Valid

bool Valid(const char far* str);

Returns true if *IsValid* returns **true**. Otherwise, calls *Error* and returns **false**. A validator's *Valid* method is called by the *Valid* method of its associated edit control.

Edit controls with associated validator objects call the validator's *Valid* method under two conditions. The first condition is when the edit control's *ofValidate* option is set and the edit control calls *Valid* when it loses focus. The second condition is when the dialog box that contains the edit control calls *Valid* for all its controls, usually because the user requested to close the dialog box or to accept an entry screen.

Protected data members

Options

uint16 Options;

A bitmap member used to control options for various descendants of *TValidator*. By default, the *TValidator* constructor clears all the bits in *Options*.

See also TVAlidatorOptions enum, TValidator::SetOption, TValidator::UnsetOption

Type definitions

validate.h

enum TValidatorOptions {voFill, voTransfer, voOnAppend, voReserved };

The *TValidatorOptions* enum constants represent bits in the bitmapped Options word in validator objects.

Constant	Meaning
voFill	Used by picture validators to indicate whether to fill in literal characters as the user types.
voTransfer	The validator handles data transfer for the input line. Currently only used by range validators.
voOnAppend	Used by picture validators to determine how to interact with edit controls.
voReserved	The bits in this mask are reserved.

TValidator::TXValidator class

validate.h

A nested class, *TXValidator* describes an exception that results from an invalid validator object. That is, if a validator expression is not valid, this exception is thrown.

Public constructor

Constructor

TXValidator(uint resId = IDS_VALIDATORSYNTAX);

Constructs a *TXValidator* object, setting the resource ID to IDS_VALIDATORSYNTAX string resource.

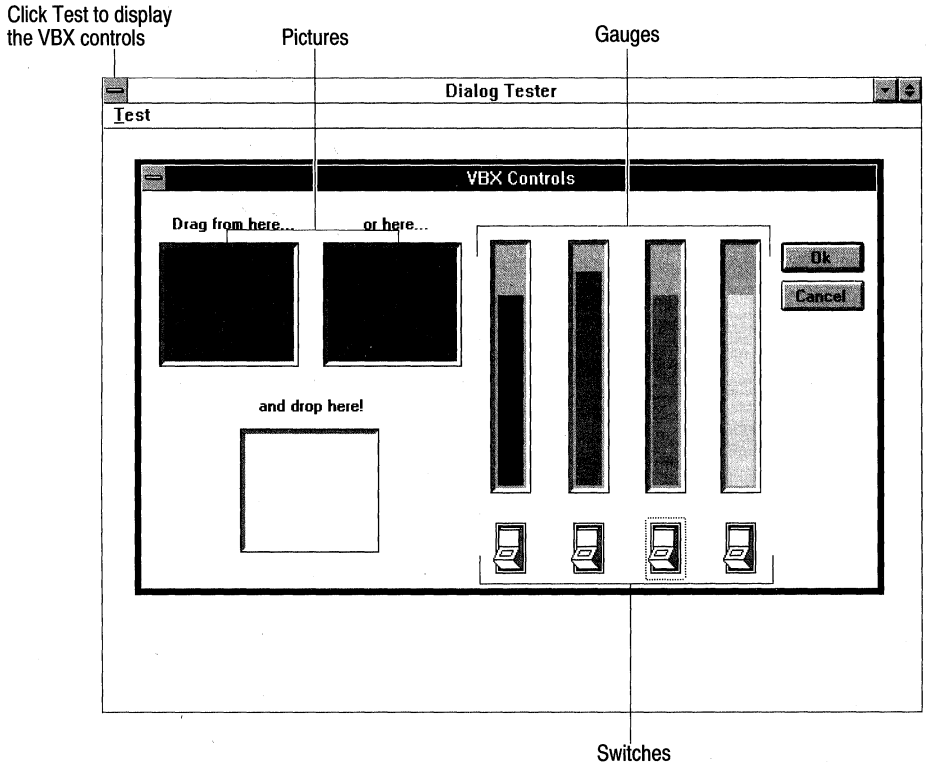
TVbxControl class

vbxctl.h

Derived from *TControl*, *TVbxControl* provides the interface for Visual Basic (VBX) controls. You can use this class to get or set the properties of VBX controls. Under certain conditions, you can also use additional methods for processing controls.

You can manipulate the control's properties using either an index value or a name. Several overloaded *GetProp* functions are provided so that you can access different types of properties. Similarly, several overloaded *SetProp* functions let you set the properties of controls using either the name of the VBX control or the index value. Consult the documentation for your VBX controls to find the name that corresponds to the property you want to manipulate.

The VBXCTLX.CPP sample program on your distribution disks displays several VBX controls.



TVbxControl provides event-handling functions that process requests to compare, delete, draw, or measure VBX controls. If you want the *TVbx* control object to process additional VBX events, you can derive a class from *TVbxControl* and add a response table that has entries for the events you want processed. For information about creating response tables and handling VBX messages, see *TVbxEventHandler*.

Public constructors and destructor

Constructors

Form 1 `TVbxControl(TWindow* parent, int id, const char far* vbxName, const char far* vbxClass, const char far* title, int x, int y, int w, int h, long initLen = 0, void far* initData = 0, TModule* module = 0);`

Constructs a VBX control where *parent* points to the parent window, *id* is the control's ID, *vbxName* is the name of the file containing the VBX control, *vbxClass* is the VBX class name, *title* is the control's caption, *x* and *y* are the coordinates in the parent window where the controls are to be placed, *w* and *h* are the width and height of the control, and *module* is the library resource ID for the control.

Form 2 `TVbxControl(TWindow* parent, int resourceId, TModule* module = 0);`

If a VBX control is part of a dialog resource, its ID can be used to construct a corresponding (or alias) ObjectWindows object. You can use this constructor if a VBX

control has already been defined in the application's resource file. *resourceId* is the resource ID of the VBX control in the resource file.

Destructor

~TVbxControl();
Destroys the *TVbxControl* object.

Public member functions

AddItem

bool AddItem(int index, const char far* item);
Adds an item (*item*) to the list of VBX control items at the specified index (*index*). Returns nonzero if successful.

Drag

bool Drag(int action);
Controls the drag and drop state of the VBX control according to the value of *action*, which can be 0 (cancel a drag operation), 1 (begin dragging a control), or 2 (end dragging a control).

GetEventIndex

int GetEventIndex(const char far* name);
Returns the index of the event associated with the name of the event passed in *name*. Returns –1 if an error occurs.

GetEventName

const char far* GetEventName(int eventindex);
Returns a string containing the name of an event associated with the integer event index number (*eventindex*). Returns 0 if an error occurs.

GetHCTL

HCTL GetHCTL();
Returns a handle to a VBX control associated with this *TVbxControl* object.

GetNumEvents

int GetNumEvents();
Returns the total number of events associated with the VBX control.

GetNumProps

int GetNumProps();
Returns the total number of properties associated with the VBX control.

GetProp

Form 1 bool GetProp(int propIndex, int& value, int arrayIndex = –1);
Gets an integer property value. An overloaded function, *GetProp* gets a VBX control property using an index value. *propIndex* is the index value of the property whose value you want to get. *value* is a reference to the variable that will receive the property values. *arrayIndex*, an optional argument, specifies the position of the value in an array property if the property is an array type. If the property isn't an array type, *arrayIndex* defaults to -1. See the third-party reference guide for your VBX controls to determine a property's

data type. *GetProp* returns nonzero if successful. To get the property by specifying the property index, use one of the following six *GetProp* functions.

Form 2 `bool GetProp(int propIndex, long& value, int arrayIndex = –1);`
Gets a *long* property value.

Form 3 `bool GetProp(int propIndex, ENUM& value, int arrayIndex=–1);`
Gets an enumerated property value. For example, a list of options associated with a font style might be defined as an enumerated type.

Form 4 `bool GetProp(int propIndex, HPIC& value, int arrayIndex=–1);`
Gets a picture (*value*). HPIC is a handle to the picture.

Form 5 `bool GetProp(int propIndex, float& value, int arrayIndex = –1);`
Gets a floating-point property value.

Form 6 `bool GetProp(int propIndex, string& value, int arrayIndex = –1);`
Gets a string property value.

Form 7 `bool GetProp(int propIndex, COLORREF value, int arrayIndex = –1);`
Gets a color property value.

Form 8 `bool GetProp(const char far* name, int& value, int arrayIndex = –1);`
Returns an integer data value. An overloaded function, *GetProp* gets a VBX control property. *propIndex* is the index value of the property whose value you want to get. *value* is a reference to the variable that will receive the property value. *arrayIndex*, an optional argument, specifies the position of the value in an array property if the property is an array type. If the property isn't an array type, *arrayIndex* defaults to –1. *GetProp* returns nonzero if successful. To get the property by specifying the name of the property, use one of the following five *GetProp* functions.

Form 9 `bool GetProp(const char far* name, long& value, int arrayIndex = –1);`
Gets a **long** property value.

Form 10 `bool GetProp(const char far* name, float& value, int arrayIndex = –1);`
Gets a floating-point property value.

Form 11 `bool GetProp(const char far* name, ENUM& value, int arrayIndex=–1);`
Gets an enumerated property value.

Form 12 `bool Getprop(const char far* name, HPIC& value, int arrayIndex=–1);`
Gets a picture (*value*) property value. HPIC is a handle to the picture.

Form 13 `bool GetProp(const char far* name, string& value, int arrayIndex = –1);`
Gets a string property value.

Form 14 `bool GetProp(const char far* name, COLORREF value, int arrayIndex = –1);`
Gets a string color property value.

See also TVbxControl::SetProp

GetPropIndex

`int GetPropIndex(const char far* name);`
Gets the integer index value for the property name passed in *name*. Returns –1 if an error occurs. This usually indicates that the property name passed in *name* couldn't be located.

GetPropName

```
const char far* GetPropName(int propIndex);
```

Gets the name for the property index passed in *index*. Returns 0 if an error occurs.

GetPropType

Form 1 `int GetPropType(int propIndex);`

Form 2 `int GetPropType(char far* name);`

Form 1 gets the type for the property specified by *index*. Form 2 gets the property string type specified by *name*. Returns 0 if an error occurs. The following table lists the names of the property types and their corresponding C++ data types.

Property type	C++ type
PTYPE_CSTRING	HSZ
PTYPE_SHORT	short
PTYPE_LONG	int32
PTYPE_BOOL	bool
PTYPE_COLOR	uint32 or COLORREF
PTYPE_ENUM	uint8 or ENUM
PTYPE_REAL	float
PTYPE_XPOS	int32 (Twips)
PTYPE_XSIZE	int32(Twips)
PTYPE_YPOS	int32 (Twips)
PTYPE_YSIZE	int32 (Twips)
PTYPE_PICTURE	HPIC
PTYPE_BSTRING	HLSTR

IsArrayProp

Form 1 `bool IsArrayProp(int propIndex);`

Form 2 `bool IsArrayProp(char far* name);`

Returns true if the property specified by *index* (Form 1) or *name* (Form 2) is an array property.

Method

```
bool Method(int method, long far* args);
```

Used for invoking customized methods, *Method* returns true if a VBX control can respond to the specified method (*method*).

Move

```
bool Move(long x, long y, long w, long h);
```

Moves a VBX control to the coordinates specified in *x* and *y*, which designate the upper left corner screen coordinates. Resizes the VBX control to *w twips* wide by *h twips* high. Returns nonzero if successful.

Refresh

```
bool Refresh();
```

Repaints the control's display area.

RemoveItem

bool RemoveItem(int index);

Removes an item (specified by *index*). The item could be removed from a list box, a combo box, or a database, for example.

SetProp

Form 1 bool SetProp(int propIndex, int value, int arrayIndex = –1);

Sets the property to an integer value. An overloaded function, *SetProp* sets a VBX control property. *propIndex* is the index number of the property whose value you want to set. *value* specifies the new value for the property. *arrayIndex* specifies the position of the value in an array property if the property is an array type. If the property isn't an array type, *arrayIndex* defaults to –1. To set the property by passing the property's index value, use one of the following six *SetProp* functions.

Form 2 bool SetProp(int propIndex, long value, int arrayIndex = –1);

Sets the property to a **long** value.

Form 3 bool SetProp(int propIndex, ENUM value, int arrayIndex=–1);

Sets the property to an enumerated value.

Form 4 bool SetProp(int propIndex, HPIC value, int arrayIndex=–1);

Sets a picture to an HPIC, or picture, value.

Form 5 bool SetProp(int propIndex, float value, int arrayIndex = –1);

Sets the property to a floating-point value.

Form 6 bool SetProp(int propIndex, const string& value, int arrayIndex = –1);

Sets the property to a string value.

Form 7 bool SetProp(int propIndex, const char far* value, int arrayIndex = –1);

Sets the property to a character string value.

Form 8 bool SetProp(int propIndex, COLORREF value, int arrayIndex = –1);

Sets the property to a color value.

Form 9 bool SetProp(const char far* name, int value, int arrayIndex = –1);

Sets the property to an integer value. An overloaded function, *SetProp* sets a VBX control property. *arrayIndex* specifies the position of the value in an array property if the property is an array type. If the property isn't an array type, *arrayIndex* defaults to –1. To set the property by using the property's name, use one of the following six *SetProp* functions.

Form 10 bool SetProp(const char far* name, long value, int arrayIndex = –1);

Sets the property to a *long* value.

Form 11 bool SetProp(const char far* name, ENUM value, int arrayIndex=–1);

Sets the property to an enumerated value.

Form 12 bool SetProp(const char far* name, HPIC value, int arrayIndex = –1);

Sets the picture property to an HPIC, or picture, value.

Form 13 bool SetProp(const char far* name, float value, int arrayIndex = –1);

Sets the property to a floating-point value.

Form 14 bool SetProp(const char far* name, const string& value, int arrayIndex = –1);

Sets the property to a string value.

Form 15 `bool SetProp(const char far* name, const char far* value, int arrayIndex = –1);`
Sets the property to a character string value.

Form 16 `bool SetProp(const char far* name, COLORREF value, int arrayIndex = –1);`
Sets the property to a color string value.

See also TVbxControl::GetProp

SetUpWindow

`void SetupWindow();`
A VBX control has an HWND plus a VBX handle. Usually, the VBX control handle is created first. However, if the window has already been created, you can use this function to extract the VBX control handle.

Protected member functions

GetClassName

`char far* GetClassName();`
Gets the name of the VBX window class.

GetVBXProperty

`bool GetVBXProperty(int propIndex, void far* value, int arrayIndex = –1);`
Returns nonzero if the specified property exists. *propIndex* specifies the index value of the integer property whose value you want to get. *value* points to the variable where the value will be stored.

PerformCreate

`void PerformCreate(int menuOrId);`
Creates a new control window and associates the VBX control with the window. Establishes the control ID, the VBX control name and class, and the window caption. Sets *Attr.style* to the window style of the control, *Attr.X* and *Attr.Y* to the upper left screen coordinates of the control, and *Attr.W* and *Attr.H* to the width and height of the control.

SetVBXProperty

`bool SetVBXProperty(int propIndex, int32 value, int arrayIndex=–1);`
Returns nonzero if the specified property value is set, or 0 if unsuccessful. *propindex* is the index number of the property whose value you want to set. *value* is the value to be stored. An optional argument, *arrayIndex*, which is –1 by default, specifies the index value in an array of values of the property to be set.

Response table entries

The *TVbxControl* class has no response table entries.

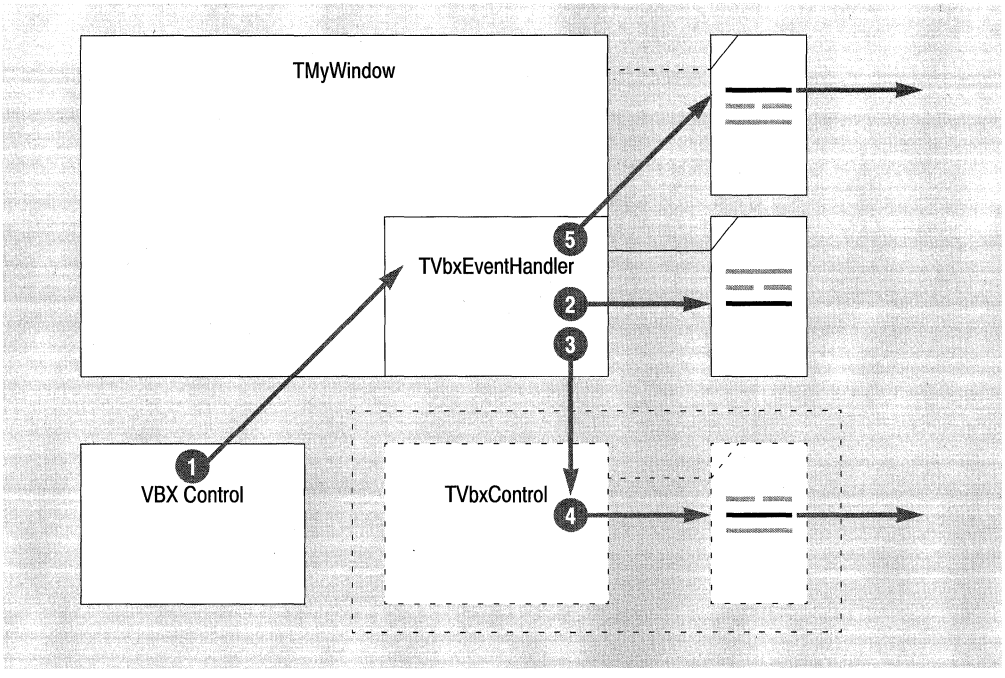
TVbxEventHandler class

vbxctl.h

Derived from *TEventHandler*, *TVbxEventHandler* handles events from VBX controls. Although you will never need to modify this class, *TVbxEventHandler* needs to be mixed in with your window class so that it can receive events from VBX controls. For example,

```
class TMyWindow:public TWindow, public TVbxEventHandler
{
// Include class definition here.
}
```

The following diagram illustrates the flow of information between VBX controls, parent windows, and response tables.



When a VBX control fires an event (sends an event message), the following sequence of events occurs:

1 The VBX Control sends a WM_VBXFIREEVENT message to *TMyWindow*.

2 *TMyWindow's TVbxEventHandler* finds a WM_VBXFIREEVENT message in its response table and calls *EvVbxDispatch*.

3 If a child window is present, *EvVbxDispatch* dispatches the event to the child.

4 If there is an event-handling function in the child window's response table, the child window handles the event.

5 If there is no child window, or the child window doesn't handle the event, *EvVbxDispatch* dispatches the event to *TMyWindow's* response table.

In other words, when a VBX control sends a WM_VBXFIREEVENT message, the parent window's *TVbxEventHandler* catches this message first, converts it into a form understood by a window's response table, and attempts to send the converted message to the child window. If there is no child window or if the child window doesn't handle the message, *TVbxEventHandler* sends the converted message to the parent window. When the parent window receives the message, it calls the handler function that corresponds to the message.

Two response table macros, EV_VBXEVENTNAME and EV_VBXEVENTINDEX, map VBX events to handler functions. Of the two macros, EV_VBXEVENTNAME is more commonly used. EV_VBXEVENTINDEX is intended for use with code generators, which can determine the event index values for a VBX control. Both macros call an event handler function and point to the VBXEVENT structure. A typical EV_VBXEVENTNAME response table entry might be

```
EV_VBXEVENTNAME(IDC_BUTTON1, "MouseMove", EvMouseMove);
```

where IDC_BUTTON1 is the event ID, "EvMouseMove" is the event name, and *EvMouseMove* is the handler function.

The *lparam* of a WM_VBX FIREEVENT message points to a VBXEVENT structure, which holds information about the event and the control that generated the event. The VBXEVENT structure contains the following members:

```
typedef struct VBXEVENT {
   HCTL    Control;
   HWND    Window;
   int     ID;
   int     EventIndex;
   LPCSTR  EventName;
   int     NumParams;
   LPVOID  ParamList;
} VBXEVENT;
```

where

- *Control* is a handle to the VBX control sending the message.
- *Window* is the handle of the VBX control window.
- *ID* is the ID of the VBX control.
- *EventIndex* is the event index.
- *EventName* is the name of the event.
- *NumParams* holds the number of event arguments.
- *ParamList* is a pointer to a list of pointers to the event's arguments. The *ParamList* data member provides access to the actual arguments of the event.

To handle VBX events, your program uses an event-handling function. In the following example, *EvMouseMove* is the name of the handler function, which passes a pointer to the VBXEVENT event structure. VBX_EVENTARGNUM is the macro that takes *event*, type, and an index number as its parameters. *event* references the VBXEVENT structure, **short** is the event argument type, and 0 and 1 are the index numbers of the argument.

The argument types and indexes can be found in the documentation for the VBX control.

```
void EvMouseMove(VBXEVENT FAR* event)
{
   short X = VBX_EVENTARGNUM(event, short, 0);
   short Y = VBX_EVENTARGNUM(event, short, 1);
}
```

Because VBX controls were originally designed to be used with Visual Basic, their event arguments are documented in terms of Basic data types. The following table lists the Basic types, their C++ equivalents, and macros.

Basic	C++	Macro
Boolean	short	VBX_EVENTARGNUM(*event, short, index*)
Control	HCTL	VBX_EVENTARGNUM(*event, HCTL, index*)
Double	double	VBX_EVENTARGNUM(*event, double, index*)
Enum	short	VBX_EVENTARGNUM(*event, short, index*)
Integer	short	VBX_EVENTARGNUM(*event, short, index*)
Long	long	VBX_EVENTARGNUM(*event, long, index*)
Single	float	VBX_EVENTARGNUM(*event, float, index*)
String	HLSTR	VBX_EVENTARGSTR(*event, index*)

The following table lists the standard VBX events and corresponding arguments that the Borland C++ VBX emulation library supports.

Event	Arguments
Click	None
DblClick	None
DragDrop	Source as Control, X as Integer, Y as Integer
DragOver	Source as Control, X as Integer, Y as Integer, State as Integer
GotFocus	None
KeyDown	Key as Integer, Shift as Integer
KeyPress	Key as Integer, Shift as Integer
KeyUp	Key as Integer, Shift as Integer
LostFocus	None
MouseDown	X as Integer, Y as Integer
MouseMove	X as Integer, Y as Integer, Shift as Integer, Button as Integer
MouseUp	X as Integer, Y as Integer, Shift as Integer, Button as Integer

For the DragOver event, the state argument can be one of the following values:

- 0, where the source control is being dragged within a target's range.
- 1, where the source control is being dragged out of a target's range.
- 2, where the source control is being moved from one position in the target to another.

For both the DragOver and DragDrop events, the Control argument type should be translated to HCTL (a handle to the VBX control) for C++. The X and Y values are in pixels, not twips.

If a standard VBX event has a Shift key argument, the argument has these bit values:

Key	Bit value
Shift	0x1
Ctrl	0x2
Alt	0x4 (Used in connection with a Menu selection)

If a standard VBX event has a Button key argument, the argument has these bit values:

Button	Bit value
Left	0x1
Right	0x2
Middle	0x4

The following example shows how you might use these Shift key arguments. For example, if you want the VBX control to perform some action when the mouse is moved and the Shift key is pressed, you could write a function such as

```
void EvMouseMove(VBXEVENT FAR* event)
{
   short X = VBX_EVENTARGNUM(event, short, 0);
   short Y = VBX_EVENTARGNUM(event, short, 1);
   short Shift = VBX_EVENTARGNUM(event, short, 2);
   short Button = VBX_EVENTARGNUM(event, short, 3);
   if (shift & 0x2)
      MessageBox ("The control key is pressed.");
}
```

Borland C++ uses pixels to express the X and Y coordinate arguments of standard VBX events. This differs from Visual Basic, which expresses coordinates in twips (1/20th of a point or 1/1440 of an inch). Custom events are usually expressed in terms of twips. You can use these functions to convert between pixels and twips.

Function	Meaning
VBXPix2TwpX	Converts an X argument from pixels to twips
VBXPix2TwpY	Converts a Y argument from pixels to twips
VBXTwp2PixX	Converts an X argument from twips to pixels
VBXTwp2PixY	Converts a Y argument from twips to pixels

Protected member functions

EvVbxDispatch

LRESULT EvVbxDispatch(WPARAM wp, LPARAM lp);

After *TVbxEventHandler* receives a WM_VBXFIREEVENT message from the parent window, it calls *EvVbxDispatch*, which sends the message to the correct event-handling function and passes a pointer to the VBXEVENT structure.

EvVbxInitForm
LRESULT EvVbxInitForm(WPARAM wp, LPARAM lp);

Response table entries

Response table entry	Member function
EV_MESSAGE(WM_VBXFIREEVENT, EvVbxDispatch)	EvVbxDispatch
EV_MESSAGE(WM_VBXINITFORM, EvVbxinitForm)	EvVbxInitForm

TView class

docview.h

Derived virtually from both *TEventHandler* and *TStreamableBase*, *TView* is the interface presented to a document so it can access its client views. Views then call the document functions to request input and output streams. Views own the streams and are responsible for attaching and deleting them.

Instead of creating an instance of *TView*, you create a derived class that can implement *TView's* virtual functions. The derived class must have a way of knowing the associated window (provided by *GetWindow*) and of describing the view (provided by *GetViewName*). The view must also be able to display the document title in its window (*SetDocTitle*).

TView uses several event handler functions to query views, commit, and close views. For example, if a view is associated with a window that can gain focus, then it should handle the *vnIsWindow* notification message.

View classes can take various forms. For example, a view class can be a window (through inheritance), can contain a window (an embedded object), can reference a window, or can be contained within a window object. A view class might not even have a window, as in the case of a voice mail or a format converter. Some remote views (for example, those displayed by DDE servers) might not have local windows.

Other viewer classes derived from *TView* include *TEditView*, *TListView*, and *TWindowView*. These classes display different types of data: *TEditView* displays unformatted text files, *TListView* displays text information in a list box, and *TWindowView* is a basic viewer from which you can derive other types of viewers such as hex file viewers.

For OLE-enabled applications, use *TOleView*, which supports views for embedded objects and compound documents.

See also
TOleView class

Public data members

enum

enum {PrevProperty = 0, ViewClass, ViewName, NextProperty,};

These property values, defined for *TView*, are available in classes derived from *TView*. *PrevProperty* and *NextProperty* are delimiters for every document's property list.

Tag

void far* Tag;

Tag holds a pointer to the application defined data. Typically, you can use *Tag* to install a pointer to your own application's associated data structure. *TView* zeros *Tag* during construction and doesn't access it again.

Public constructor and destructor

Constructor

TView(TDocument& doc);

Constructs a *TView* object of the document associated with the view. Sets *ViewId* to *NextViewId*. Calls *TDocument::AttachView* to attach the view to the associated document.

Destructor

virtual ~TView();

Frees a *TView* object and calls *DetachView* to detach the view from the associated document.

Public member functions

FindProperty

virtual int FindProperty(const char far* name);

FindProperty gets the property index, given the property name (*name*). Returns 0 if the name isn't found.

See also pfxxxx property access constants

GetDocument

TDocument& GetDocument();

Returns a reference to the view's document.

GetNextViewId

static unsigned GetNextViewId{};

Returns the next view ID to be assigned.

GetProperty

virtual int GetProperty(int index, void far* dest, int textlen=0);

Returns the total number of properties where *index* is the property index, *dest* contains the property data, and *textlen* is the size of the property array. If *textlen* is 0, property data is returned as binary data; otherwise, property data is returned as text data.

See also pfxxxx property access constants, TView::SetProperty

GetViewId
unsigned GetViewId();
Returns the unique ID for this view.

GetViewMenu
TMenuDescr* GetViewMenu();
Returns the menu descriptor for this view. This can be any existing *TMenuDescr* object. If no descriptor exists, *ViewMenu* is 0.

GetViewName
virtual const char far* GetViewName()=0;
Pure virtual function that returns 0. Override this function in your derived class to return the name of the class.

See also TEditView::StaticName, TEditView::GetViewName

GetWindow
virtual TWindow* GetWindow()
GetWindow returns the *TWindow* instance associated with the view or 0 if no view exists.

See also TeditView::GetWindow

IsOK
bool IsOK();
Returns nonzero if the view is successfully constructed.

See also TView::NotOK

PropertyCount
virtual int PropertyCount();
Gets the total number of properties for the *TDocument* object. Returns *NextProperty* –1.

See also pfxxxx property access constants

PropertyFlags
virtual int PropertyFlags(int index);
Returns the attributes of a specified property given the index (*index*) of the property whose attributes you want to retrieve.

See also pfxxxx property access constants, TView::FindProperty, TView::PropertyName

PropertyName
virtual const char* PropertyName(int index);
Returns the text name of the property given the index value.

See also pfxxxx property access constants, TView::FindProperty

SetDocTitle
virtual bool SetDocTitle(const char far* docname, int index)
Stores the document title.

See also TWindow::SetDocTitle

SetProperty

virtual bool SetProperty(int index, const void far* src);

Sets the value of the property, given the index of the property, and *src*, the data type (either binary or text) to which the property must be set.

See also pfxxxx property access constants, TView::GetProperty

SetViewMenu

void SetViewMenu(TMenuDescr* menu);

Sets the menu descriptor for this view. This can be any existing *TMenuDescr* object. If no descriptor exists, *ViewMenu* is 0.

See also TView::GetViewMenu

Protected data member

Doc

TDocument* Doc;

Holds the current document.

Protected member functions

NotOK

void NotOK();

Sets the view to an invalid state, thus causing *IsOK* to return 0.

See also TView::IsOK

TWidthHeight enum

layoutco.h

enum TWidthHeigh {lmWidth = lmCenter + 1, lmHeight};

Used by the *TLayoutConstraint* struct, *TWidthHeight* enumerates the values that control the width (*lmWidth*) and height (*lmHeight*) of the window.

See also TLayoutConstraint struct

TWindow class

window.h

TWindow, derived from *TEventHandler* and *TStreamableBase*, provides window-specific behavior and encapsulates many functions that control window behavior and specify window creation and registration attributes.

TWindow is a generic window that can be resized and moved. You can construct an instance of *TWindow*, though normally you'll use *TWindow* as a base for your specialized window classes. In general, to associate and disassociate a *TWindow* object with a window element, you need to follow these steps:

1 Construct an instance of a *TWindow*.

2 Call *Create* or *Execute*, which creates the interface element (HWND) and then calls *SetupWindow*, which calls the base *SetupWindow* for normal processing, which in turn involves
   - Creating the HWindow and any child HWindows.
   - Calling *TransferData* to setup the transfer of data between the parent and child windows.

3 To destroy the interface element, choose one of the following actions, depending on your application:
   - Call *Destroy* to destroy the interface element unconditionally.
   - Call *CloseWindow*, which calls *CanClose* to test whether it's OK to destroy the interface element.

4 There are two ways to destroy the interface object:
   - If the object has been **new**'d, use **delete**.
   - If the object hasn't been **new**'d, the compiler automatically destructs the object.

The ObjectWindows destroy process consists of two parts: (1) call *Destroy* to destroy the interface element and (2) then delete the C++ object. However, it is perfectly valid to call *Destroy* on the interface element without deleting the C++ object and then to call *Create* at a later time to re-create the window. Because it is also valid to construct a C++ window object on the stack or as an aggregated member, the *Destroy* function can't assume it should delete the C++ object.

The user-generated WM_CLOSE event handler, *EvClose*, also causes a C++ object to be deleted by passing the "**this**" pointer to the application. The C++ object is deleted automatically because the *EvClose* event frequently occurs in response to a user action, and this is the most convenient place for the deletion to take place. Later, when it's safe to do so, the application then deletes the window pointer. Because the stack often contains selectors that refer to the addresses of objects that may become invalid during the delete process, it's not safe to delete the "**this**" pointer while events are still being processed. If the addresses become invalid, they could cause trouble when they are reloaded from the stack.

TWindow is the base class for all window classes, including *TFrameWindow*, *TControl*, *TDialog*, and *TMDIChild*. The ObjectWindows hierarchy diagram shows the many classes that are derived from *TWindow*.

Public data members

Attr

TWindowAttr Attr;

Holds a *TWindowAttr* structure, which contains the window's creation attributes. These attributes, which include the window's style, extended style, position, size, menu ID, child window ID, and menu accelerator table ID, are passed to the function that creates the window.

See also TWindow::TWindow, TWindow::Create, TWindowAttr struct

DefaultProc

WNDPROC DefaultProc;

Holds the address of the default window procedure. *DefWindowProc* calls *DefaultProc* to process Windows messages that are not handled by the window.

See also TWindow::DefWindowProc

HWindow

HWND HWindow;

Holds the handle to the associated MS-Windows window, which you'll need to access if you make calls directly to Windows API functions.

Parent

TWindow* Parent;

Points to the interface object that serves as the parent window for this interface object.

Scroller

TScroller* Scroller;

Points to the scroller object that supports either the horizontal or vertical scrolling for this window.

Status

TStatus Status;

Status is used to signal an error in the initialization of an interface object. Setting *Status* to a nonzero value causes a *TXIncompatibility* exception to be thrown. Classes derived from *TWindow* do not attempt to associate an interface element with an object whose previous initialization has failed. *Status* is included only to provide backward compatibility with previous versions of ObjectWindows.

Title

char far* Title;

Title points to the window's caption. When there is a valid HWindow, *Title* will yield the same information as *::GetWindowText* if you use *TWindow::SetCaption* to set it.

See also TDialog::SetCaption, TDialog::SetupWindow, TWindow::GetWindowTextTitle, TWindow::SetCaption

Public constructors and destructor

Constructors

Form 1 TWindow(HWND hWnd, TModule* module = 0);

Constructs a *TWindow* that is used as an alias for a non-ObjectWindows window, and sets wfAlias. Because the HWND is already available, this constructor, unlike the other *TWindow* constructor, performs the "thunking" and extraction of HWND information instead of waiting until the function *Create* creates the interface element.

Both forms: The following paragraphs describe procedures common to both constructors. module specifies the application or DLL that owns the *TWindow* instance. ObjectWindows needs the correct value of module to find needed resources. If module is 0, *TWindow* sets its module according to the following rules:

- If the window has a parent, the parent's module is used.
- If the *TWindow* constructor is invoked from an application, the module is set to the application.
- If the *TWindow* constructor is invoked from a DLL that is dynamically linked with the ObjectWindows DLL and the currently running application is linked the same way, the module is set to the currently running application.
- If the *TWindow* constructor is invoked from a DLL that is statically linked with the ObjectWindows library or the invoking DLL is dynamically linked with ObjectWindows DLL but the currently running application is not, no default is used for setting the module. Instead, a *TXInvalidModule* exception is thrown and the object is not created.

Form 2 `TWindow(TWindow* parent, const char far* title = 0, TModule* module = 0);`
Adds **this** to the child list of *parent* if nonzero, and calls *EnableAutoCreate* so that **this** will be created displayed along with *parent*. Also sets the title of the window initializes the window's creation attributes.

See the previous constructor for a description of the procedures common to both constructors.

Destructor

`virtual ~TWindow();`
Destroys a still-associated interface element by calling *Destroy*. Deletes the window objects in the child list, then removes **this** from the parent window's child list. Deletes the *Scroller* if it is nonzero. Frees the cursor, if any exists, and the object instance (thunk).

See also TWindowFlag enum, TWindow::EnableAutoCreate

Public member functions

AdjustWindowRect

`static void AdjustWindowRect(TRect& rect, uint32 style, bool menu);`
AdjustWindowRect calculates the size of the window rectangle according to the indicated client-rectangle size. *rect* refers to the structure that contains the client rectangle's coordinates. *style* specifies the style of the window. *menu* is **true** if the window has a menu.

AdjustWindowRectEx

`static void AdjustWindowRectEx(TRect& rect, uint32 style, bool menu, uint32 exStyle);`
AdjustWindowRectEx calculates the size of a window rectangle that has an extended style. *TRect* refers to the structure that contains the client rectangle's coordinates. *style* specifies the window styles of the window to be adjusted, and *menu* returns **true** if the

window has a menu. *exStyle* indicates the extended styles to be used for the window. Extended styles include the following styles:

Value	Meaning
WS_EX_ACCEPTFILES	The window can make use of drag and drop files.
WS_EX_DLGMODALFRAME	The window has a double border that can be created with a title bar if the WS_CAPTION style flag is specified.
WS_EX_NOPARENTNOTIFY	The child window created from this style does not send parent notify messages to the parent window when the child is created or destroyed.
WS_EX_TOPMOST	A window having this style is placed above windows that aren't topmost and remains above the non-topmost windows even when it's deactivated.
WS_EX_TRANSPARENT	A window having this style is transparent, that is, any windows beneath this window are not concealed by this window.

See also TWindowAttr struct

BringWindowToTop

void BringWindowToTop();

BringWindowToTop brings a pop-up or child window to the top of the stack of overlapping windows and activates it.

CanClose

virtual bool CanClose();

Use this function to determine if it's okay to close a window. Returns **true** if the associated interface element can be closed. Calls the *CanClose* member function of each of its child windows. Returns **false** if any of the *CanClose* calls returns **false**.

In your application's main window, you can override *TWindow's CanClose* and call *TWindow::MessageBox* to display a YESNOCANCEL message prompting the user to

YES—Save the data, or

NO—Don't save the data, but close the window, or

CANCEL—Cancel the close operation and return to the edit window.

The following example shows how to write a *CanClose* function that displays a message box asking if the user wants to save a drawing that has changed. To save time, *CanClose* uses the *IsDirty* flag to see if the drawing has changed. If so, *CanClose* queries the user before closing the window.

```
bool TMyWindow::CanClose()
{
  if (IsDirty)
    switch(MessageBox("Do you want to save?", "Drawing has changed.",
                      MB_YESNOCANCEL | MB_ICONQUESTION)) {
      case IDCANCEL:
        // Choosing Cancel means to abort the close - return false.
        return false;

      case IDYES:
```

```
        // Choosing Yes means to save the drawing.
        CmFileSave();
      }
    return true;
  }
```

CheckDlgButton

void CheckDlgButton(int buttonId, uint check);

Places a checkmark in (or removes a checkmark from) the button specified in *buttonId*. If *check* is nonzero, the checkmark is placed next to the button; if 0, the checkmark is removed. For buttons having three states, *check* can be 0 (clear), 1 (checked), or 2 (gray).

CheckRadioButton

void CheckRadioButton(int firstButtonId, int lastButtonId, int checkButtonId);

Checks the radio button specified by *checkButtonId* and removes the checkmark from the other radio buttons in the group. *firstButtonId* and *lastButtonId* specify the first and last buttons, respectively, in the group.

ChildBroadcastMessage

void ChildBroadcastMessage(uint msg, WPARAM wParam, LPARAM lParam);

Sends the specified message to all immediate children using *SendMessage*.

See also TWindow::SendMessage

ChildWindowFromPoint

HWND ChildWindowFromPoint(const TPoint&) const;

Determines which of the child windows contains the point specified in *TPoint*. Returns a handle to the window that contains the point, or 0 if the point lies outside the parent window.

See also TWindow::WindowFromPoint

ChildWithId

TWindow* ChildWithId(int id) const;

Returns a pointer to the window in the child window list that has the supplied ID. Returns 0 if no child window has the indicated *id*.

ClearFlag

void ClearFlag(TWindowFlag mask);

Clears the specified *TWindow* wfXxxx constant flags (for example wfAlias, wfTransfer, and so on) in the *Flags* member.

See also TWindowFlag enum

ClientToScreen

void ClientToScreen(TPoint& point) const;

Converts the client coordinates specified in *TPoint* to screen coordinates for the new window.

CloseWindow

virtual void CloseWindow(int retval = 0);

Determines if it's okay to close a window before actually closing the window. If *this* is the main window of the application, calls *GetApplication->CanClose*. Otherwise, calls

this->CanClose to determine whether the window can be closed. After determining that it is okay to close the window, *CloseWindow* calls *Destroy* to destroy the HWND.

See also TApplication::CanClose, TWindow::CanClose

CmExit

void CmExit();

CmExit is called in response to the selection of a menu item that has an ID of CM_EXIT. If *this* is the main window, *CmExit* calls *CloseWindow*.

Create

virtual bool Create();

Creates the windows interface element to be associated with this ObjectWindows interface element.

CreateCaret

Form 1 void CreateCaret(HBITMAP hBitmap);

Creates a new caret for the system. *HBITMAP* specifies the bitmapped caret shape.

Form 2 void CreateCaret(bool isGray, int width, int height);

Create a new caret for the system with the specified shape, bitmap shade, *width*, and height. If *width* or *height* is 0, the corresponding system-defined border size is used.

CreateChildren

bool CreateChildren();

Creates the child windows in the child list whose auto-create flags (with wfAutoCreate mask) are set.

See also TWindow::EnableAutoCreate, TWindow::DisableAutoCreate, TWindowFlag enum

DefaultProcessing

LRESULT DefaultProcessing();

DefaultProcessing serves as a general-purpose default processing function that handles a variety of messages. After being created and before calling *DefaultProcessing*, however, a window completes this sequence of events:

- If the window is already created, *SubclassWindow* is used to install *StdWndProc* in place of the window's current procedure. The previous window procedure is saved in *DefaultProc*.
- If the window hasn't been created, *InitWndProc* is set up as the window proc in the class. Then, when the window first receives a message, *InitWndProc* calls *GetThunk* to get the window's instance thunk (created by the constructor by calling *CreateInstanceThunk*). *InitWndProc* then switches the message-receiving capability from the window's procedure to *StdWndProc*.

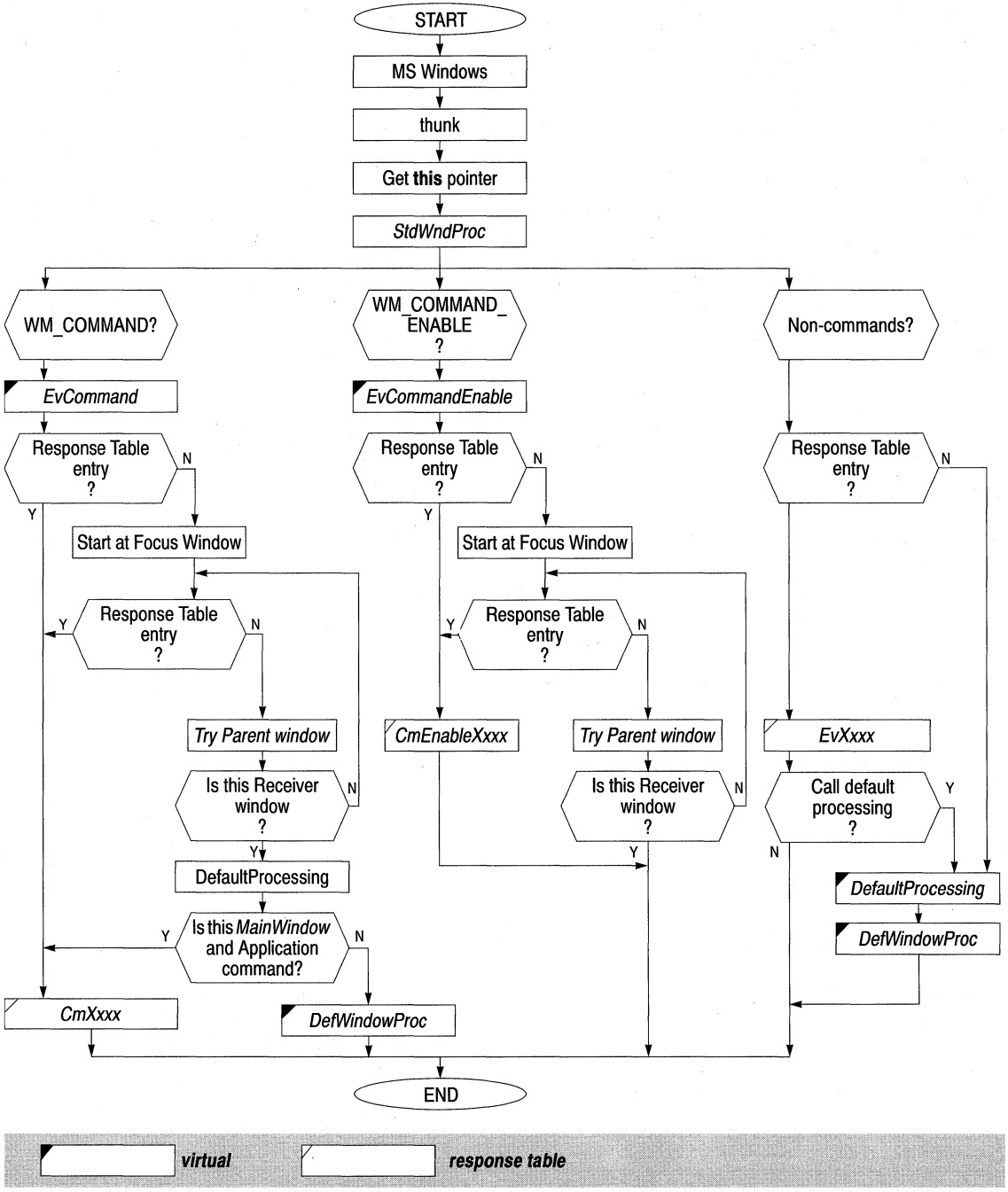
After this point, *StdWndProc* responds to incoming messages by calling the window's virtual *WindowProc* to process the messages. ObjectWindows uses the special registered message *::GetWindowPtrMsgId* to get the **this** pointer of an HWND. *StdWndProc* responds to this message by returning the **this** pointer obtained from the thunk.

If the incoming message is not a command or command enable message, *WindowProc* immediately searches the window's response table for a matching entry. If the incoming

message is a command or command enable message, *WindowProc* calls *EvCommand* or *EvCommandEnable*. *EvCommand* and *EvCommandEnable* begin searching for a matching entry in the focus window's response table. If an entry is found, the corresponding function is dispatched; otherwise ObjectWindows calls *DefaultProcessing* to finish the recursive walk back up the parent chain, searching for a match until the receiving window (the window that initially received the message) is reached. At this point, one of the following actions occurs:

- If there is still no match and this is the *MainWindow* of the application, the window searches the application's response table.
- If there are no matches and this is a command, *DefWindowProc* is called.
- If this is a *CommandEnable* message, no further action is taken.
- If this is not a command, and if a response table entry exists for the window, *WindowProc* dispatches the corresponding *EvXxxx* function to handle the message.
- If this is the application's *MainWindow*, and the message is designed for the application, the message is forwarded to the application.
- For any other cases, the window calls *DefWindowProc*.

The following diagram illustrates this sequence of message-processing events:



See also TWindow::DefWindowProc

DefWindowProc

virtual LRESULT DefWindowProc(uint msg, WPARAM wParam, LPARAM lParam);

Performs default Windows processing and passes the incoming Windows message. You usually don't need to call this function directly. Classes such as *TMDIFrame* and *TMDIChild's DefWindowProc* override this function to perform specialized default processing.

See also TWindow::DefaultProc, TWindow::WindowProc, TMDIFrame::DefWindowProc

Destroy

virtual void Destroy(int retVal = 0);

First, *Destroy* calls *EnableAutoCreate* for each window in the child list to ensure that windows in the child list will be re-created if *this* is re-created. Then, it destroys the associated interface element.

If a derived window class expects to be destructed directly, it should call *Destroy* as the first step in its destruction so that any virtual functions and event handlers can be called during the destroy sequence.

See also TWindow::EnableAutoCreate

DestroyCaret

static void DestroyCaret();

DestroyCaret first checks the ownership of the caret. If a window in the current task owns the caret, *DestroyCaret* destroys the caret and removes it from the screen.

See also TWindow::CreateCaret

DisableAutoCreate

void DisableAutoCreate();

Disables the feature that allows an associated child window interface element to be created and displayed along with its parent window. Call *DisableAutoCreate* for pop-up windows and controls if you want to create and display them at a time later than their parent windows.

See also TWindow::EnableAutoCreate

DisableTransfer

void DisableTransfer();

Disables (for the interface object) the transfer mechanism, which allows state data to be transferred to and from a transfer buffer.

See also TWindowFlag enum

Dispatch

virtual LRESULT Dispatch(TEventInfo& info, WPARAM wp, LPARAM lp = 0);

Cracks and dispatches a *TWindow* message. The *info* parameter is the event-handling function. The *wp* and *lp* parameters are the message parameters the dispatcher cracks.

DragAcceptFiles

void DragAcceptFiles(bool accept);

If a window can process dropped files, *DragAcceptFiles* sets accept to TRUE.

DrawMenuBar

void DrawMenuBar();

DrawMenuBar redraws the menu bar. This function should be called to redraw the menu if the menu is changed after the window is created.

EnableAutoCreate

void EnableAutoCreate();

Ensures that an associated child-window interface element is created and displayed along with its parent window. By default, this feature is enabled for windows and controls, but disabled for dialog boxes.

See also TWindow::DisableAutoCreate

EnableScrollBar

void EnableScrollBar(uint sbFlags = SB_BOTH, uint arrowFlags = ESB_ENABLE_BOTH);

Disables or enables one or both of the scroll bar arrows on the scroll bars associated with this window. *sbFlags*, which specifies the type of scroll bar, can be one of the Scroll Bar constants (SB_CTL, SB_HORZ, SB_VERT, or SB_BOTH). By default, the arrows on both the horizontal and vertical scroll bars are either enabled or disabled. *arrowFlags*, which indicates whether the scroll bar arrows are enabled or disabled, can be one of the Enable Scroll Bar constants (ESB_ENABLE_BOTH, ESB_DISABLE_LTUP, ESB_DISABLE_RTDN, ESB_DISABLE_BOTH). By default, the arrows on both the horizontal and vertical scroll bars are enabled.

See also SB_Xxxx ScrollBar Constants

EnableTransfer

void EnableTransfer();

Enables the transfer mechanism, which allows state data to be transferred between the window and a transfer buffer.

EnableWindow

virtual bool EnableWindow(bool enable);

Allows the given window to receive input from the keyboard of mouse. If enable is TRUE, the window can receive input. Use the function *IsWindowEnabled* to determine if the window has been enabled.

See also TWindow::IsWindowEnabled

EnumProps

int EnumProps(PROPENUMPROC proc);

Enumerates all the items in the property list of the current window and passes them one by one to the callback function indicated in *proc*. The process continues until every item has been enumerated or until *proc* returns zero. *proc* holds the address of the callback function.

EvChildInvalid

void EvChildInvalid(HWND hWnd);

Responds to a WM_CHILDINVALID message posted by a child edit control. Indicates that the contents of the child window are invalid.

EvCommand

virtual LRESULT EvCommand(uint id, HWND hWndCtl, uint notifyCode);

WindowProc calls *EvCommand* to handle WM_COMMAND messages. *id* is the identifier of the menu item or control. *hWndCtl* holds a value that represents the control sending the message. If the message is not from a control, it is 0. *notifyCode* holds a value that represents the control's notification message. If the message is from an accelerator, *notifyCode* is 1; if the message is from a menu, *notifyCode* is 0.

See also TWindow::DefaultProcessing

EvCommandEnable

virtual void EvCommandEnable(TCommandEnabler& ce);

Called by *WindowProc* to handle WM_COMMAND_ENABLE messages, *EvCommand* calls the *CmXxxx* command-handling function or calls *DefaultProcessing* to handle the incoming message.

See also TWindow::DefaultProcessing, TCommandEnabler

EvSysCommand

void EvSysCommand(uint cmdType, TPoint& point);

Responds to a user-selected command from the System menu or when the user selects the maximize or minimize box. Applications that modify the system menu must process *EvSysCommand* messages. Any *EvSysCommand* messages not handled by the application must be passed to *DefWindowProc*. The parameter *cmdType* can be one of the following system commands:

Constant	Meaning
SC_CLOSE	Close the window.
SC_HOTKEY	Activate the specified window.
SC_HSCROLL	Scroll horizontally.
SC_KEYMENU	Retrieve a menu through a keystroke.
SC_MAXIMIZE (or SC_ZOOM)	Maximize the window.
SC_MINIMIZE (or SC_ICON)	Minimize the window.
SC_MOUSEMENU	Retrieve a menu through a mouse click.
SC_NEXTWINDOW	Move to the next window.
SC_PREVWINDOW	Move to the previous window.
SC_SCREENSAVE	Execute the specified screen saver.
SC_SIZE	Size the window
SC_TASKLIST	Activate the Windows Task Manager.
SC_VSCROLL	Scroll vertically.

In the following example, *EvSysCommand* either processes system messages or calls *DefaultProcessing*:

```
void MyWindow::EvSysCommand(uint cmdType, TPoint&)
{
   switch (cmdType& 0xFFF0) {
      case SC_MOUSEMENU:
      case SC_KEYMENU:
```

```
            break;
         default:
            DefaultProcessing();
      }
   }
```

See also TWindow::DefaultProcessing

FirstThat

TWindow* FirstThat(TCondFunc test, void* paramList = 0);
TWindow* FirstThat(TCondMemFunc test, void* paramList = 0);

There are two *FirstThat* functions, both of which pass a pointer to an iterator function. The first *FirstThat* points to a nonmember function as its first parameter; the second *FirstThat* points to a member function instead.

Both *FirstThat* functions iterate over the child list, calling a Boolean *test* function and passing each child window in turn as an argument (along with *paramList*). If a *test* call returns TRUE, the iteration is stopped and *FirstThat* returns the child window object that was supplied to test. Otherwise, *FirstThat* returns 0.

In the following example, *GetFirstChecked* calls *FirstThat* to obtain a pointer (*p*) to the first check box in the child list that is checked:

```
bool IsThisBoxChecked(TWindow* p, void*) {
   return ((TCheckBox*)p)->GetCheck() == BF_CHECKED;
}
TCheckBox* TMyWindow::GetFirstChecked() {
   return FirstThat(IsThisBoxChecked);
}
```

See also TCondFunc type

FlashWindow

bool FlashWindow(bool invert);

FlashWindow changes the window from active to inactive or vice versa. If *invert* is nonzero, the window is flashed. If *invert* is 0, the window is returned to its original state—either active or inactive.

ForEach

Form 1 void ForEach(TActionFunc action, void* paramList = 0);

There are two *ForEach* functions. This *ForEach* takes a nonmember function as its first parameter; the other *ForEach* (see the following entry) takes a member function instead. This version of *ForEach* iterates over the child list, calling a nonmember function supplied as the action to be performed and passing each child window in turn as the argument (along with *paramList*).

In the following example, *CheckAllBoxes* calls *ForEach*, checking all the check boxes in the child list:

```
void CheckTheBox(TWindow* p, void*) {
   ((TCheckBox*)p)->Check();
}
```

```
void CheckAllBoxes() {
   ForEach(CheckTheBox);
}
```

Form 2 void ForEach(TActionMemFunc action, void* paramList = 0);
Refer to the previous *ForEach* description. The difference between the two *ForEach* members is that the first *ForEach* takes a nonmember function as a parameter and this *ForEach* takes a member function as a parameter.

See also TActionFunc typedef, TActionMemFunc typedef

ForwardMessage

Form 1 LRESULT ForwardMessage(bool send = true)
Forwards the window's current message. Calls *SendMessage* if send is TRUE; otherwise calls *PostMessage*.

Form 2 LRESULT ForwardMessage(HWND hWnd, bool send = true);
Forwards the window's current message to another HWND. Calls *SendMessage* if send is **true**; otherwise calls *PostMessage*.

See also TWindow::PostMessage

GetActiveWindow

static HWND GetActiveWindow();
GetActiveWindow retrieves the handle of the active window. Returns 0 if no window is associated with the calling thread.

GetApplication

TApplication* GetApplication()const;
Gets a pointer to the *TApplication* object associated with this. Use *GetApplication* to obtain access to data and functions in the *TApplication* object.

GetCapture

static HWND GetCapture();
Returns the handle of the window that has captured the mouse.

GetCaretBlinkTime

static uint GetCaretBlinkTime();
GetCaretBlinkTime retrieve the caret blink rate in milliseconds.

See also TWindow::SetCaretBlinkTime

GetCaretPos

static void GetCaretPos(TPoint& point);
GetCaretPos gets the position of the caret in the coordinates of the client window. *point* refers to the structure that receives the client coordinates of the caret.

See also TWindow::SetCaretPos

GetClassLong

long GetClassLong(int index) const;

GetClassLong retrieves the 32-bit value about the window class. If unsuccessful, returns 0. Depending on the value of *index*, *GetClassLong* can retrieve the following information:

Value	Meaning
GCL_CBCLSEXTRA	Size in bytes of memory associated with this class
GCL_CBWINDEXTRA	Size of extra window memory associated with each window
GCL_HBRBACKGROUND	Handle of the background brush associated with the class
GCL_HCURSOR	Handle of the cursor
GCL_HICON	Handle of the icon
GCL_HMODULE	Handle of the module that registered the class
GCL_MENUNAME	Address of the menu name string
GCL_STYLE	The style bits associated with a window class
GCL_WNDPROC	Address of the window procedure associated with this class

See also Twindow::SetClassLong

GetClassWord

uint16 GetClassWord(int index) const;

GetClassWord gets a 16-bit value containing information about the class or style of the window. If unsuccessful; returns 0. Depending on the value of *index*, *GetClassWord* can retrieve the following information:

Value	Meaning
GCW_CBCLSEXTRA	Number of additional class information
GCW_CBWINDEXTRA	Number of bytes of additional window information
GCW_HBRBACKGROUND	Handle of the background brush
GCW_HCURSOR	Handle of the cursor
GCW_HICON	Handle of the icon
GCW_HMODULE	Handle of the module
GCW_STYLE	The style bits associated with a window class

See also TWindow::SetClassWord

GetClientRect

Form 1 TRect GetClientRect() const;

GetClientRect gets the coordinates of the window's client area (the area in a window you can use for drawing).

Form 2 void GetClientRect(TRect& rect) const;

Gets the coordinates of the window's client area and then copies them into the object referred to by *TRect*.

GetCursorPos

static void GetCursorPos(TPoint& pos);

GetCursorPos retrieves the cursor's current position (in window screen coordinates) and copies the values into the structure pointed to by *TPoint*.

GetDesktopWindow

static HWND GetDesktopWindow();

GetDesktopWindow returns a handle to the desktop window.

GetDlgCtrlID

int GetDlgCtrlID() const;

GetDlgCtrlID returns the ID of the control.

GetDlgItem

HWND GetDlgItem(int childId) const;

GetDlgItem retrieves the handle of a control specified by *childId*.

See also TWindow::GetDlgItemInt

GetDlgItemInt

uint GetDlgItemInt(int childId, bool *translated = 0,bool isSigned = true) const;

GetDlgItemInt retrieves the text of a control specified by *childId*. translated points to the variable that receives the translated value. *isSigned* indicates that the retrieved value is signed (the default).

See also TWindow::GetDlgItem

GetDlgItemText

uint GetDlgItemText(int childId, char far* text, int max) const;

GetDlgItemText retrieves the text of a control specified by *childId*. *text* points to the text buffer to receive the text. *max* specifies the maximum length of the caption, which is truncated if it exceeds this length.

See also TWindow::SetDlgItemText

GetFirstChild

TWindow* GetFirstChild()

Returns a pointer to the first child window, which is the first window created, in the interface object's child list.

See also TWindowAttr struc

GetFocus

static HWND GetFocus();

Gets a handle to the window that has the focus. Use the function *SetFocus* to set the keyboard focus to this window.

See also TWindow::SetFocus

GetHWndState

void GetHWndState();

Copies the style, coordinate, and the resource *id* (but not the title) from the existing HWnd into the ObjectWindows' *TWindow* members.

GetId

int GetId();

Returns *Attr.Id*, the ID used to find the window in a specified parent's child list.

See also TWindowAttr struct

GetModule
TModule* GetModule() const;
Returns a pointer to the module object.

GetLastActivePopup
HWND GetLastActivePopup() const;
Returns the last active pop-up window in the list.

GetLastChild
TWindow* GetLastChild();
Returns a pointer to the last child window in the interface object's child list.

GetMenu
HMENU GetMenu() const;
GetMenu returns the handle to the menu of the indicated window. If the window has no menu, the return value is 0.

See also TWindow::SetMenu

GetNextDlgGroupItem
HWND GetNextDlgGroupItem(HWND hWndCtrl, bool previous = false) const;
GetNextDlgGroupItem returns either the next or the previous control in the dialog box. *hWndCtrl* identifies the control in the dialog box where the search begins. If previous is 0, *GetNextDlgGroupItem* searches for the next control; if nonzero, it searches for the previous control.

GetNextDlgTabItem
HWND GetNextDlgTabItem(HWND HWndCtrl, bool previous = false) const;
GetNextDlgTabItem returns the handle of the first control that lets the user press the TAB key to move to the next control (that is, the first control with the WS_TABSTOP style associated with it). *hWndCtrl* identifies the control in the dialog box where the search begins. If previous is 0, *GetNextDlgGroupItem* searches for the next control; if nonzero, it searches for the previous control.

GetNextWindow
HWND GetNextWindow(uint dirFlag) const;
GetNextWindow finds the handle associated with either the next or previous window in the window manager's list. *dirFlag* specifies the direction of the search. Under the Win 32 API, *GetNextWindow* returns either the next or the previous window's handle. If the application is not running under Win32, *GetNextWindow* returns the next window's handle.

GetWindowPtr
TWindow* GetWindowPtr(HWND hWnd);
This version of *GetWindowPtr* actually calls *TApplication*'s *GetWindowPtr* on the application associated with this window. Then, given the handle to this window (*hWnd*), *GetWindowPtr* returns the *TWindow* pointer associated with ths window.

See also TApplication:GetWindowPtr

GetParent
HWND GetParent() const;

GetParent retrieves the handle of the parent window. If none exists, returns 0.

See also TWindow::SetParent

GetProp

Form 1 HANDLE GetProp(uint16 atom) const;

GetProp returns a handle to the property list of the specified window. *atom* contains a value that identifies the character string whose handle is to be retrieved. If the specified string is not found in the property list for this window, returns NULL.

Form 2 HANDLE GetProp(const char far* string) const;

GetProp returns a handle to the property list of the specified window. Unlike the previous *GetProp* function, *string* points to the string whose handle is to be retrieved. If the specified string is not found in the property list for this window, returns NULL.

See also TWindow::SetProp

GetScrollPos

int GetScrollPos(int bar) const;

GetScrollPos returns the thumb position in the scroll bar. The position returned is relative to the scrolling range. If *bar* is SB_CTL, it returns the position of a control in the scroll bar; if *bar* is SB_HORZ, it returns the position of a horizontal scroll bar; if *bar* is SB_VERT, it returns the position of a vertical scroll bar.

See also TWindow::SetScrollPos, SB_Xxxx Scroll Bar Constants

GetScrollRange

void GetScrollRange(int bar, int& minPos, int& maxPos) const;

GetScrollRange returns the minimum and maximum positions in the scroll bar. If *bar* is SB_CTL, it returns the position of a control in the scroll bar; if *bar* is SB_HORZ, it returns the position of a horizontal scroll bar; if *bar* is SB_VERT, it returns the position of a vertical scroll bar. *minPos* and *maxPos* hold the lower and upper range, respectively, of the scroll bar positions. If there are no scroll bar controls or if the scrolls are non-standard, *minPos* and *maxPos* are zero.

See also TWindow::SetScrollRange, SB_Xxxx Scroll Bar Constants

GetSysModalWindow

static HWND GetSysModalWindow();

Retrieves the handle of the system-modal window.

See also TWindow::SetSysModalWindow

GetSystemMenu

HMENU GetSystemMenu(bool revert = false) const;

GetSystemMenu returns a handle to the system menu so that an application can access the system menu.

GetThunk

WNDPROC GetThunk()const;

Gets the instance thunk, a small piece of code created for use with exported callback functions. (A callback function is a function that exists within a program but is called

from outside the program by a Windows library routine, for example, a dialog box function.)

GetTopWindow

HWND GetTopWindow()const;

GetTopWindow returns a handle to the top window currently owned by this parent window. If no children exist, *GetTopWindow* returns 0.

GetUpdateRect

bool GetUpdateRect(TRect& rect, bool erase = true) const;

GetUpdateRect retrieves the screen coordinates of the rectangle that encloses the updated region of the specified window. *erase* specifies whether *GetUpdateRect* should erase the background of the updated region.

See also TWindow::RedrawWindow

GetUpdateRgn

bool GetUpdateRgn(TRegion& rgn, bool erase = true) const;

GetUpdateRgn copies a window's update region into a region specified by *region*. If *erase* is **true**, *GetUpdateRgn* erases the background of the updated region and redraws nonclient regions of any child windows. If *erase* is **false**, no redrawing occurs.

If the call is successful, *GetUpdateRgn* returns a value indicating the kind of region that was updated. If the region has no overlapping borders, it returns SIMPLEREGION; if the region has overlapping borders, it returns COMPLEXREGION; if the region is empty, it returns NULLREGION; if an error occurs, it returns ERROR.

See also TWindow::RedrawWindow

GetWindow

HWND GetWindow (uint cmd) const;

Returns the handle of the window that has the indicated relationship to this window. *cmd* , which indicates the type of relationship to be obtained, can be one of the following values:

See also TApplication:GetWindowPtr

GetWindowFont

HFONT GetWindowFont();

Gets the font the control uses to draw text. The return value is a handle of the font the control uses. If a system default font is being used, *GetWindowFont* returns NULL.

GetWindowLong

long GetWindowLong(int index) const;

GetWindowLong retrieves information about the window depending on the value stored in *index*. The values returned, which provide information about the window, include the following *GWL_Xxxx* window style constants:

Value	Description
GWL_EXSTYLE	The extended window style
GWL_STYLE	The window style (position, device context creation, size, and so on)
GWL_WNDPROC -	The address of the window procedure being processed

Value	Description
In the case of the dialog box, additional information can be retrieved, such as:	
DWL_DLGPROC	The address of the procedure processed by the dialog box
DWL_MSGRESULT	The value that a message processed by the dialog box returns
DWL_USER	Additional information that pertains to the application, such as pointers or handles the application uses.

See also TWindow::GetClassLong

GetWindowPlacement

bool GetWindowPlacement(WINDOWPLACEMENT* place) const;

GetWindowPlacement retrieves display and placement information (normal, minimized, and maximized positions) about the window and stores that information in the argument, *place*.

See also TWindow::SetWindowPlacement, TWindow::Show

GetWindowRect

Form 1 void GetWindowRect(Trect& rect) const;

Gets the screen coordinates of the window's rectangle and copies them into *rect*.

Form 2 TRect GetWindowRect() const;

Gets the screen coordinates of the window's rectangle.

See also TWindow::GetClientRect

GetWindowTask

HTASK GetWindowTask() const;

Returns a handle to the task that created the specified window.

GetWindowText

int GetWindowText(char far* string, int maxCount) const;

GetWindowText copies the window's title into a buffer pointed to by *string*. *maxCount* indicates the maximum number of characters to copy into the buffer. A string of characters longer than *maxCount* is truncated. *GetWindowText* returns the length of the string or 0 if no title exists.

See also TWindow::SetWindowText, TWindow::GetWindowTextTitle, TWindow::SetCaption

GetWindowTextLength

int GetWindowTextLength() const;

GetWindowTextLength returns the length, in characters, of the specified window's title. If the window is a control, returns the length of the text within the control. If the window does not contain any text, *GetWindowTextLength* returns 0.

See also TWindow::SetWindowText

GetWindowTextTitle

void GetWindowTextTitle();

Updates the *TWindow Title* data member from the current window's caption. *GetWindowTextTitle* is used to keep *Title* synchronized with the actual window state when there is a possibility that the state might have changed.

See also TWindow::SetCaption, TWindow::Title

GetWindowWord

uint16 GetWindowWord(int index) const;

GetWindowWord retrieves information about this window depending on the value of *index*. *GetWindowWord* returns one of the following values that indicate information about the window:

Value	Meaning
GWW_HINSTANCE	The instance handle of the module owning the window
GWW_HWNDPARENT	The handle of the parent window
GWW_ID	The ID number of the child window

See also TWindow::GetWindowLong, TWindow::SetWindowWord, TWindow::SetParent

HandleMessage

LRESULT HandleMessage(uint msg, WPARAM wParam = 0, LPARAM lParam = 0);

Handles message sent to a window. *HandleMessage* can be called directly to handle Windows messages without going through *SendMessage*.

See also TWindow::SendMessage

HideCaret

void HideCaret();

HideCaret removes the caret from the specified display screen. The caret is hidden only if the current task's window owns the caret. Although the caret is not visible, it can be displayed again using *ShowCaret*

See also TWindow::CreateCaret, TWindow::ShowCaret

HiliteMenuItem

bool HiliteMenuItem(HMENU hMenu, uint idItem, uint hilite);

HiliteMenuItem either highlights or removes highlighting from a top-level item in the menu. *idItem* indicates the menu item to be processed. *hilite* (which contains a value that indicates if the *idItem* is to be highlighted or is to have the highlight removed) can be one or more of the following constants:

Value	Meaning
MF_BYCOMMAND	The *idItem* parameter contains the menu item's identifier.
MF_BYPOSITION	The *idItem* parameter contains the zero-based relative position of the menu item.
MF_HILITE	Highlights the menu item. If this value is not specified, highlighting is removed from the item.
MF_UNHILITE	Removes the menu item's highlighting.

If the menu is set to the specified condition, *HiliteMenuItem* returns **true**; otherwise, returns **false**.

See also TWindow::GetMenu

HoldFocusHWnd

virtual bool HoldFocusHWnd(HWND hWndLose, HWND hWndGain);

Responds to a request by a child window to hold its HWND when it is losing focus. Stores the child's HWND in *HwndRestoreFocus*.

See also TFrameWindow::HoldFocusHwnd

HWND()

operator HWND()const;

Allows a *TWindow&* to be used as an HWND in Windows API calls by providing an implicit conversion from *TWindow* to HWND.

IdleAction

virtual bool IdleAction(long idleCount);

Called when no messages are waiting to be processed, *IdleAction* performs idle processing as long as **true** is returned. *idleCount* specifies the number of times *idleAction* has been called between messages.

Invalidate

void Invalidate(bool erase = true);

Invalidates (mark for painting) the entire client area of a window. The window then receives a message to redraw the window. By default, the background of the client area is marked for erasing.

See also TWindow::Validate, TWindow::InvalidateRect

InvalidateRect

void InvalidateRect(const TRect&, bool erase = true);;

Invalidates a specified client area. By default, the background of the client area to be invalidated is marked for erasing.

See also TWindow::ValidateRect, TWindow::Invalidate

InvalidateRgn

void InvalidateRgn(HRGN hRgn, bool erase = true);

InvalidateRgn invalidates a client area within a region specified by the *hRgn* parameter when the application receives a WM_PAINT message. The region to be invalidated is assumed to have client coordinates. If *hRgn* is 0, the entire client area is included in the region to be updated. The parameter *erase* specifies whether the background with the update region needs to be erased when the region to be updated is determined. If *erase* is **true**, the background is erased; if *erase* is **false**, the background is not erased when the *Paint* function is called. By default, the background within the region is marked for erasing.

See also TWindow::ValidateRgn, TWindow::Paint

IsChild

bool IsChild(HWND hWnd) const;

IsChild is TRUE if the window is a child window or a descendant window of this window. A window is considered a child window if it is the direct descendant of a given parent window and the parent window is in a chain of windows leading from the original overlapped or pop-up window down to the child window. *hWnd* identifes the window to be tested.

IsDlgButtonChecked

uint IsDlgButtonChecked(int buttonId) const;

IsDlgButtonChecked indicates if the child button specified in the integer parameter, *buttonId* , is checked or if a button is grayed, checked or neither. If the return value is 0, the button is unchecked. If the return value is 1, the button is checked. If the return value is 3, the button state is undetermined. This function sends a BM_GETCHECK message to the specified button control.

IsFlagSet

bool IsFlagSet(TWindowFlag mask);

Returns the state of the bit flag in *Attr.Flags* whose *mask* is supplied. Returns **true** if the bit flag is set, and **false** if not set.

See also TWindowAttr struc

IsIconic

bool IsIconic() const;

Is TRUE if window is iconic or minimized.

IsWindow

bool IsWindow() const;

Is TRUE if an HWND is being used.

IsWindowEnabled

bool IsWindowEnabled();

Is **true** if the window is enabled. Use the function *EnableWindow* to enable or disable a window.

See also TWindow::EnableWindow

IsWindowVisible

bool IsWindowVisible() const;

Is **true** if the window is visible. By default, *TWindow's* constructor sets the window style attribute (WS_VISIBLE) so that the window is visible.

IsZoomed

bool IsZoomed() const;

Is **true** if window is zoomed or maximized.

KillTimer

bool KillTimer(uint timerId);

KillTimer gets rid of the timer and removes any WM_TIMER messages from the message queue. *timerId* contains the ID number of the timer event to be killed.

See also TWindow::SetTimer

LockWindowUpdate

bool LockWindowUpdate();

LockWindowUpdate prevents or enables window drawing for one window at a time. If the window is locked, returns **true**; otherwise, returns **false**, which indicates an error occurred or some other window is already locked.

If there is any attempted drawing attempted within a locked window or locked child windows, the extent of the attempted operation is saved within a bounding rectangle. When the window is then unlocked, the area withing the rectangle is invalidated. This causes a paint message to be sent to this window. If any drawing occurred while the window was locked for updates, the area is invalidated.

MapWindowPoints

void MapWindowPoints(HWND hWndTo, TPoint* points, int count) const;

MapWindowPoints maps a set of points in one window to a relative set of points in another window. *hWndTo* specifies the window to which the points are converted. *points* points to the array containing the points. If *hWndTo* is 0, the points are converted to screen coordinates. *count* specifies the number of *points* structures in the array.

MessageBox

int MessageBox(const char far* text,const char far* caption = 0, uint type = MB_OK);

Creates and displays a message box that contains a message (*text*), a title (*caption*), and icons or push buttons (*type*). If *caption* is 0, the default title is displayed. Although *type* is set to one push button by default, it can contain a combination of the *MB_Xxxx* message constants. This function returns one of the following constants:

Value	Description
IDABORT	User selected the abort button.
IDCANCEL	User selected the cancel button.
IDIGNORE	User selected the ignore button.
IDNO	User selected the no button.
IDOK	User selected the OK button
IDRETRY	User selected the retry button.
IDYES	User selected the yes button.

If BWCC is already enabled, then the message box will be a BWCC enabled. If CTRL 3D is already enabled, then the message box will be CTRL 3D enabled. If neither BWCC nor CTRL 3D is enabled, the message box will be displayed as a standard windows message box.

See also TWindow::PostMessage, MB_Xxxx message constants

MoveWindow

Form 1 void MoveWindow(int x, int y, int w, int h, bool repaint = false);

MoveWindow repositions the specified window. *x* and *y* specify the new upper left coordinates of the window; *w* and *h* specify the new width and height, respectively. If *repaint* is **false**, the window is not repainted after it is moved.

Form 2 void MoveWindow(const TRect& rect, bool repaint = false);

MoveWindow repositions the window. *rect* references the left and top coordinates and the width and height of the new screen rectangle. If *repaint* is FALSE, the window is not repainted after it is moved.

See also TWindow::RedrawWindow

Next

TWindow* Next();

Returns a pointer to the next sibling window in the window's sibling list.

See also TWindow::Previous

NumChildren

unsigned NumChildren();

Returns the number of child windows of the window.

OpenClipboard

TClipboard& OpenClipboard();

Opens the clipboard and prevents other application from changing the contents of the clipboard. This function fails if another window has already opened the clipboard.

Paint

virtual void Paint(TDC& dc, bool erase, TRect& rect);

Repaints the client area (the area you can use for drawing) of a window. Called by base classes when responding to a WM_PAINT message, *Paint* serves as a placeholder for derived types that define *Paint* member functions. *Paint* is called by *EvPaint* and requested automatically by Windows to redisplay the window's contents. *dc* is the paint display context supplied to text and graphics output functions. The supplied reference to the *rect* structure is the bounding rectangle of the area that requires painting. *erase* indicates whether the background needs erasing.

PostMessage

bool PostMessage(uint msg, WPARAM wParam = 0, LPARAM lParam=0) const;

PostMessage posts a message (*msg*) to the window in the application's message queue. It returns without waiting for the corresponding window to process the message.

See also TWindow::ForwardMessage, TWindow::MessageBox

PerformCreate

virtual void PerformCreate(int menuOrId);

PerformCreate is called from within *Create* to perform the final step in creating an MS_Windows interface element to be associated with an ObjectWindows window. *PerformCreate* can be overridden to provide alternate HWND create implementation.

See also TWindow::Create

PreProcessMsg

virtual bool PreProcessMsg(MSG& msg);

PreProcessMsg allows preprocessing of queued messages prior to dispatching. If you override this method in a derived class, be sure to call the base class's *PreProcessMsg* because it handles the translation of accelerator keys. When nonzero is returned, message processing stops.

See also TApplication::ProcessAppMsg

Previous

TWindow* Previous();

Returns a pointer to the previous window in the window's sibling list.

See also TWindow::Next

ReceiveMessage

LRESULT ReceiveMessage(uint msg, WPARAM wParam = 0, LPARAM lParam = 0);

Called from *StdWndProc, ReceiveMessage* is the first member function called when a message is received. It calls *HandleMessage* from within the try block of the exception-handling code. In this way, exceptions can be caught and suspended before control is returned to exception-unsafe Windows code.

See also TWindow::HandleMessage

RedrawWindow

bool RedrawWindow(TRect* update, HRGN hUpdateRgn, uint redrawFlags = RDW_INVALIDATE | RDW_UPDATENOW | RDW_ERASE);

RedrawWindow redraws the rectangle specified by update, and the region specified by *hUpdateRgn*. *redrawFlags* can be a combination of one or more of the following *RDW_Xxxx* Redraw Window constants used to invalidate or validate a window.

Value	Description
These values invalidate a window:	
RDW_ERASE	When the window is repainted, it receives a WM_ERASEBKGND message. If RDW_INVALIDATE is not also specified, this flag has no effect.
RDW_FRAME	Any part of the non-client area of the window receives a WM_NCPAINT message if it intersects the region to be updated.
RDW_INTERNALPAINT	A WM_PAINT message is posted to the window whether or not it contains an invalid region.
RDW_INVALIDATE	Invalidates either *hUpdateRgn* or *update*. In cases where both are 0, the entire window becomes invalid.
These values validate a window:	
RDW_NOERASE	The window is prevented from receiving any WM_ERASEBKGND messages.
RDW_NOFRAME	The window is prevented from receiving any WM_NCPAINT messages. The flag RDW_VALIDATE must also be used with this flag.
RDW_NOINTERNALPAINT	The window is prevented from receiving internal WM_PAINT messages, but does not prevent the window from receiving WM_PAINT messages from invalid regions.
RDW_VALIDATE	Validates *update* and *hUpdateRgn*. However, if both are 0, the entire window area is validated. The flag does not have any effect on internal WM_PAINT messages.

Value	Description
These flags control when the window is repainted:	
RDW_ERASENOW	Before the function returns, the specified windows will receive WM_NCPAINT and WM_ERASEBKGND messages.
RDW_UPDATENOW	Before the function returns, the specified windows will receive WM_NCPAINT, WM_ERASEBKGND, as well as WM_PAINT messages.

See also TWindow::GetUpdateRect

Register

virtual bool Register();

Registers the Windows registration class of **this** window, if **this** window is not already registered. Calls *GetClassName* and *GetWindowClass* to retrieve the Windows registration class name and attributes of **this** window. Register returns **true** if **this** window is registered.

See also TWindow::GetClassName, TWindow::GetWindowClass, WNDCLASS struct

RegisterHotKey

bool RegisterHotKey(int idHotKey, uint modifiers, uint virtKey);

RegisterHotKey registers a hotkey ID with the current application. *modifiers* can be a combination of keys that must be pressed to activate the specified *hotkey*, for example, HOTKEYF_SHIFT, HOTKEYF_CONTROL, and HOTKEYF_ALT.

See also TWindow::UnRegisterHotKey

ReleaseCapture

static void ReleaseCapture();

Releases the mouse capture from this window.

RemoveProp

Form 1 HANDLE RemoveProp(uint16 atom) const;

RemoveProp removes the property specified by *atom* from the application's property list. *atom* indicates the string to be removed. Returns the handle of the given string or NULL if no string exists in the window's property list.

Form 2 HANDLE RemoveProp(const char far* str) const;

RemoveProp removes the property specified by *str*, a null-terminated string, from the application's property list. Returns the handle of the given string or NULL if no string exists in the window's property list.

See also TWindow::GetProp

ScreenToClient

void ScreenToClient(TPoint& point) const;

ScreenToClient uses the screen coordinates specified in point to calculate client window's coordinates and then place the new coordinates into point.

ScrollWindow

void ScrollWindow(int dx, int dy, const TRect* scroll = 0, const TRect far* clip = 0);

ScrollWindow scrolls a window in the vertical (*dx*) and horizontal (*dy*) directions. *TRect* indicates the area to be scrolled. If 0, the entire client area is scrolled. *clip* specifies the clipping rectangle to be scrolled. Only the area within *clip* is scrolled. If *clip* is 0, the entire window is scrolled.

See also TWindow::ScrollWindowEx

ScrollWindowEx

void ScrollWindowEx(int dx, int dy, const TRect far* scroll = 0, const TRect far* clip = 0, HRGN hUpdateRgn = 0, TRect far* update = 0, uint flags = 0);

ScrollWindowEx scrolls a window in the vertical (*dx*) and horizontal (*dy*) directions. scroll indicates the area to be scrolled. If 0, the entire client area is scrolled. *clip* specifies the clipping rectangle to be scrolled. Only the area within *clip* is scrolled. If *clip* is 0, the entire window is scrolled. *update* indicates the region that will receive the boundaries of the area that becomes invalidated as a result of scrolling. *flags*, which determines how the window's children are scrolled, can be one of the following *SW_Xxxx* Scroll Window constants:

Value	Description
SW_ERASE	Erases the invalidated region after sending an erase background message to the window indicated by the SW_INVALIDATE flag value.
SW_INVALIDATE	Invalidates the region indicated by the *hUpdate* parameter.
SW_SCROLLCHILDREN	Scrolls all the child window intersecting the rectangle pointed to by the *scroll* parameter.

See also TWindow::ScrollWindow, TWindow::Show

SendDlgItemMessage

LRESULT SendDlgItemMessage(int childId, uint msg, WPARAM wParam = 0, LPARAM lParam = 0);

SendDlgItemMessage sends a message (*msg*) to the control specified in *childId*.

See also TWindow::SendMessage

SendMessage

LRESULT SendMessage(uint msg, WPARAM wParam = 0, LPARAM lParam = 0);

SendMessage sends a message (*msg*) to a specified window or windows. After it calls the window procedure, it waits until the window procedure has processed the message before returning.

See also TWindow::ChildBroadcastMessage, TWindow::HandleMessage, TWindow::SendDlgItemMessage

SendNotification

void SendNotification(int id, int notifyCode, HWND hCtl, uint msg = WM_COMMAND);

Repacks a command message (*msg*) so that a child window (*hCtl*) can send a message to its parent regardless of whether this is a WIN16 or WIN32 application.

SetActiveWindow

HWND SetActiveWindow();

SetActiveWindow activates a top-level window. Returns a handle to the previously active window.

See also TWindow::GetActiveWindow

SetBkgndColor

void SetBkgndColor(uint32 color);

Sets the background color (*color*) for the window. You can also get the current color of an element displayed on the screen. For example,

```
layout -> SetBkgndColor(GetSysColor(COLOR_APPWORKSPACE));
```

uses one of the Windows COLOR values (in this case, the color of multiple document interface applications (*MDI*).

See also TWindow::BkgndColor

SetCaption

void SetCaption(const char far* title);

Copies *title* to an allocated string pointed to by *title*. Sets the caption of the interface element to *title*. Deletes any previous title.

See also TWindow::GetWindowTextTitle, TWindow::Title

SetCapture

HWND SetCapture();

Sets the mouse capture to the current window. All mouse input is directed to this window.

SetCaretBlinkTime

static void SetCaretBlinkTime(uint16 milliSecs);

SetCaretBlinkTime sets the caret blink rate in milliseconds.

See also TWindow::GetCaretBlinkTime

SetCaretPos

Form 1 static void SetCaretPos(int x, int y);

SetCaretPos sets the position of the caret in the coordinates of the client window. *x* and *y* indicate the client coordinates of the caret.

Form 2 static void SetCaretPos(const TPoint& pos);

SetCaretPos sets the position of the caret in the coordinates of the client window. *pos* indicates the client coordinates of the caret.

See also TWindow::GetCaretPos, TWindow::ShowCaret

SetClassLong

long SetClassLong(int index) const;

SetClassLong sets the long value at the specified offset (*index*). Depending on the value of *index*, *SetClassLong* sets a handle to a background brush, cursor, icon, module, menu, window function, or extra class bytes.

See also TWindow::GetClassLong

SetClassWord

uint16 SetClassWord(int index, uint16 newWord);

SetClassWord sets the word value at the specified offset (*index*). Depending on the value of *index* , *SetClassLong* sets the number of bytes of class information, of additional

window information, or the style bits. Unlike *SetClassLong*, *SetClassWord* uses one of the following *GCW_xxxx* Class Word constants:

Value	Meaning
GCW_HBRBACKGROUND	Sets a handle for a background brush.
GCW_HCURSOR	Sets a handle of a cursor.
GCW_HICON	Sets a handle of an icon.
GCW_STYLE	Sets a style bit for a window class.

See also TWindow::GetClassWord

SetCursor

bool SetCursor(TModule* module, TResId resId);

Sets the cursor position for the window using the given *module* and *ResId*. If the *module* parameter is 0, *CursorResId* can be one of the IDC_xxxx constants that represent different kinds of cursors. See the data member for a list of these cursor values. If the mouse is over the client area, *SetCursor* changes the cursor that is displayed.

See also TWindow::GetCursorPos, TWindow::CursorResId

SetDlgItemInt

void SetDlgItemInt(int childId, uint value, bool isSigned = true) const;

SetDlgItem sets the child window with the Id (*childId*) in the window to the integer value specified in *value*. If *isSigned* is **true**, the value is signed.

See also TWindow::GetDlgItem

SetDlgItemText

void SetDlgItemText(int childId, const char far* text) const;

SetDlgItemText sets the text of a child with Id. *text* points to the text buffer containing the window caption or text that is to be copied into the child window caption or text.

See also TWindow::GetDlgItemText

SetDocTitle

virtual bool SetDocTitle(const char far* docname, int index);

Stores the title of the document (*docname*). *index* is the number of the view displayed in the document's caption bar. In order to determine what the view number should be, *SetDocTitle* makes two passes: the first pass checks to see if there's more than one view, and the second pass, if there is more than one view, assigns the next number to the view. If there is only one view, *index* is 0, and therefore, the document doesn't display a view number. When *TDocument's* checking to see if more than one view exists, *index* is –1. In such cases, only the document's title is displayed in the caption bar.

SetDocTitle returns **true** if there is more than one view, and if *TDocument* displays the number of the view passed in *index*.

SetFlag

void SetFlag(TWindowFlag mask);

If TRUE is supplied, the bits in *Attr.Flags* in *Mask* are set. Otherwise, the bit is cleared. *Mask* can be any one, or a combination, of the *wfXxxx* constants.

See also TWindow::IsFlagSet, TWindow: Flag enum

SetFocus

HWND SetFocus();

Sets the keyboard focus to current window and activates the window that receives the focus by sending a WM_SETFOCUS message to the window. All future keyboard input is directed to this window, and any previous window that had the input focus loses it. If successful, returns a handle to the window that has the focus; otherwise, returns NULL.

See also TWindow::GetFocus

SetMenu

bool SetMenu(HMENU hMenu);

Sets the specified window's menu to the menu indicated by *hMenu*. If *hMenu* is 0, the window's current menu is removed. *SetMenu* returns 0 if the menu remains unchanged; otherwise, it returns a nonzero value.

See also TWindow::GetMenu, TMDIFrame::SetMenu

SetModule

void SetModule(TModule* module);

Sets the default module for this window.

See also TWindow::GetModule

SetNext

void SetNext(TWindow* next);

Sets the next window in the sibling list.

SetParent

virtual void SetParent(TWindow* newParent);

Sets the parent for the specified window by setting *Parent* to the specified new *Parent* window object. Removes **this** window from the child list of the previous parent window, if any, and **adds** this window to the new parent's child list.

See also TWindow::GetParent

SetProp

Form 1 bool SetProp(uint16 atom, HANDLE data) const;

Adds an item to the property list of the specified window. *atom* contains a value that identifies the data entry to be added to the property list.

Form 2 bool SetProp(const char far* str, HANDLE data) const;

Adds an item to the property list of the specified window. *str* points to the string used to identify the entry data to be added to the property list.

See also TWindow::GetProp

SetRedraw

void SetRedraw(bool redraw);

Sends a WM_SETREDRAW message to a window so that changes can be redrawn (redraw = **true**) or to prevent changes from being redrawn (redraw = **false**).

SetScrollPos

int SetScrollPos(int bar, int pos, bool redraw = true);

SetScrollPos sets the thumb position in the scroll bar. *bar* identifies the position (horizontal, vertical, or scroll bar control) to return and can be one of the SB_xxxx scroll bar constants.

See also TWindow::GetScrollPos, SB_Xxxx Scroll Bar Constants

SetScrollRange

void SetScrollRange(int bar, int minPos, int maxPos, bool redraw = true);

SetScrollRange sets the thumb position in the scroll bar. *bar* identifies the position (horizontal, vertical, or scroll bar control) to set and can be one of the SB_xxxx scroll bar constants. *minPos* and *maxPos* specify the lower and upper range, respectively, of the scroll bar positions.

See also TWindow::GetScrollRange, TWindow::SetScrollRange, SB_Xxxx Scroll Bar Constants

SetSysModalWindow

HWND SetSysModalWindow();

Makes the indicated window a system-modal window.

See also TWindow::GetSysModalWindow

SetTimer

uint SetTimer(uint timerId, uint timeout, TIMERPROC proc = 0);

SetTimer creates a timer object associated with this window. *timerID* contains the ID number of the timer to be created, *timeout* specifies the length of time in milliseconds, and *proc* identifies the address of the function that's to be notified when the timed event occurs. If *proc* is 0, WM_TIMER messages are placed in the queue of the application that called *SetTimer* for this window.

See also TWindow::KillTimer

SetTransferBuffer

void SetTransferBuffer(void* transferBuffer);

Sets *TransferBuffer* to *transferBuffer*.

See also TWindow::Transfer, TWindow::TransferData

SetWindowFont

void SetWindowFont(HFONT font, bool redraw);

Sets the font that a control uses to draw text. *font*, which specifies the font being used, is NULL if the default system font is used. If *redraw* is **true**, the control redraws itself after the font is set; if **false**, the control doesn't redraw itself. See the sample program, FILEBROW.CPP, for an example of how to set the font for a file browser list box.

SetWindowLong

long SetWindowLong(int index, long newLong);

SetWindowLong changes information about the window. Depending on the value of *index*, *SetWindowLong* sets a handle to a background brush, cursor, icon, module, menu, or window function. The window style can be one of the *GWL_xxxx* values that represent styles. See *GetWindowLong* for a description of these values.

See also TWindow::GetWindowLong, TWindow::SetWindowWord

SetWindowPlacement

bool SetWindowPlacement(const WINDOWPLACEMENT* place);

SetWindowPlacement sets the window to a display mode and screen position. *place* points to a window placement structure that specifies whether the window is to be hidden, minimized or displayed as an icon, maximized, restored to a previous position, activated in its current form, or activated and displayed in its normal position.

See also TWindow::GetWindowPlacement, TWindow::Show

SetWindowPos

Form 1 void SetWindowPos(HWND hWndInsertAfter, const TRect& rect, uint flags);

Changes the size of the window pointed to by *rect*. *flags* contains one of the SWP_XxxxSet Window Position constants that specify the size and position of the window. If *flags* is set to SWP_NOZORDER, *SetWindowPos* ignores the *hWndInsertAfter* parameter and retains the current ordering of the child, pop-up, or top-level windows.

Form 2 void SetWindowPos(HWND hWndInsertAfter, int x, int y, int w, int h, uint flags);

Changes the size of the window pointed to by *x*, *y*, *w*, and *h*. *flags* contains one of the SWP_Xxxx Set Window Position constants that specify the size and position of the window. If *flags* is set to SWP_NOZORDER, *SetWindowPos* ignores the *hWndInsertAfter* parameter and retains the current ordering of the child, pop-up, or top-level windows.

Value	Description
SWP_DRAWFRAME	Draws a frame around the window.
SWP_FRAMECHANGED	Sends a message to the window to recalculate the window's size. If this flag is not set, a recalculate size message is sent only at the time the window's size is being changed.
SWP_HIDEWINDOW	Hides the window.
SWP_NOACTIVATE	Does not activate the window. If this flag is not set, the window is activated and moved to the top of the stack of windows.
SWP_NOCOPYBITS	Discards the entire content area of the client area of the window. If this flag is not set, the valid contents are saved and copied into the window after the window is resized or positioned.
SWP_NOMOVE	Remembers the window's current position.
SWP_NOSIZE	Remembers the window's current size.
SWP_NOREDRAW	Does not redraw any changes to the window. If this flag is set, no repainting of any window area (including client, nonclient, and any window part uncovered as a result of a move) occurs. When this flag is set, the application must explicitly indicate if any area of the window is invalid and needs to be redrawn.
SWP_NOZORDER	Remembers the current Z-order (window stacking order).
SWP_SHOWWINDOW	Displays the window.

See also TWindow::GetWindowPlacement

SetWindowText

void SetWindowText(const char far* str);

SetWindowText sets the window's title to a buffer pointed to by *str*. *maxCount* indicates the number of characters to copy into the buffer. Note that this does not update this

window's *Title* member. Use *SetCaption* if the window's *Title* member needs to be synchronized with the window's title.

See also TWindow::GetWindowText

SetWindowWord

uint16 SetWindowWord(int index, uint16 newWord);

SetWindowWord changes information about the window. *index* specifies a byte offset of the word to be changed to the new value (*newWord*).

See also TWindow::GetWindowWord, TWindow::SetWindowLong

Show

void Show(int cmdShow);

After ensuring that the *TWindow* interface element has a valid handle, *Show* displays the *TWindow* on the screen in a manner specified by *cmdShow*, which can be one of the following SW_Xxxx show window constants:

ShowCaret

void ShowCaret();

ShowCaret displays the caret in the specified shape in the active window at the current position.

See also TWindow::CreateCaret, TWindow::HideCaret

ShowOwnedPopups

void ShowOwnedPopups(bool show);

Shows or hides all owned popup windows according to the value of *show*.

See also TWindow::Show

ShowScrollBar

void ShowScrollBar(int bar, bool show = true);

ShowScrollBar displays or hides the scroll bar. *bar* specifies whether the bar is a control itself or part of the window's nonclient area. If *bar* is part of the nonclient area, it can be one of the *SB_xxxx* scroll bar constants (specifically, SB_BOTH, SB_HORZ, or SB_VERT). If it is a control, it should be SB_CTRL. If *show* is **true**, the scroll bar is displayed; if **false**, it is hidden.

See also TWindow::GetScrollRange, SB_Xxxx Scroll Bar Constants

ShowWindow

virtual bool ShowWindow(int cmdShow);

Displays the window according to the value of *cmdShow*. See the function *Show* for a description of the SW_xxxx constants passed in *cmdShow*. *Show* is the preferred method of showing the window.

See also TWindow::Show

ShutDownWindow

Form 1 void ShutDownWindow(int retVal = 0);

This inline version of *ShutDownWindow* calls the static version of *ShutDownWindow*.

Form 2 static void ShutDownWindow(TWindow* win, int retVal = 0);

This version of *ShutDownWindow* unconditionally shuts down a given window, calls *Destroy* on the interface element, and then deletes the interface object. Instead of using *ShutDownWindow*, you can call *Destroy* directly and then delete the interface object.

SubclassWindowFunction

void SubclassWindowFunction();

Installs the instance thunk as the *WindowProc* and saves the old window function in *DefaultProc*.

See also TWindow::DefaultProc, TWindow::DefaultProcessing

Transfer

virtual uint Transfer(void* buffer, TTransferDirection direction);

Transfers data to or from any window with or without children and returns the total size of the data transferred. *Transfer* is a general mechanism for data transfer that can be used with or without using *TransferData*. The *direction* supplied specifies whether data is to be read from or written to the supplied buffer, or whether the size of the transfer data is simply to be returned. Data is not transferred to or from any child windows whose *wfTransfer flag* is not set. The return value is the size (in *bTransferData*).

virtual void TransferData(TTransferDirection direction);

A window usually calls on *TransferData* during setup and closing of windows and relies on the constructor to set *TransferBuffer* to something meaningful. *TransferData* calls the *Transfer* member function of each participating child window, passing a pointer to *TransferBuffer* as well as the direction specified in *direction* (*tdSetData*, *tdGetData*, or *tdSizeData*).

See also TWindow::EnableTransfer, TWindow::DisableTransfer, TWindow::SetupWindow, TWindow::TransferBuffer

UnregisterHotKey

bool UnregisterHotKey(int idHotKey);

UnregisterHotKey unregisters a hotkey ID with the current application.

See also TWindow::RegisterHotKey

UpdateWindow

void UpdateWindow();

UpdateWindow updates the client area of the specified window by immediately sending a WM_PAINT message.

Validate

void Validate();

Calls the function *ValidateRect* to validate (that is, remove from the area to be updated) the entire client area (the area you can use for drawing).

See also TWindow::InValidate

ValidateRect

void ValidateRect(const TRect& rect);

ValidateRect validates a portion of the client area indicated by *rect*.

ValidateRgn

void ValidateRgn(HRGN hRgn);

ValidateRgn validates the client area within a region of the current window. *hRgn* is a handle to the client area that's to be removed from the region to be updated. If *hRgn* is NULL, the entire client area is removed from the region to be updated.

See also TWindow::InvalidateRgn

WindowFromPoint

static HWND WindowFromPoint(const TPoint& point) const;

Returns the handle of the window in which the specified point (*point*) lies.

See also TWindow::ChildWindowFromPoint

WindowProc

virtual LRESULT WindowProc(uint msg, WPARAM wParam, LPARAM lParam);

WindowProc processes incoming messages by calling *EvCommand* to handle WM_COMMAND messages, *EvCommandEnable* to handle WM_COMMAND_ENABLE messages, and dispatching for all other messages.

See also TWindow::DefWindowProc

WinHelp

bool WinHelp(const char far* helpFile, uint command, uint32 data);

WinHelp invokes a specified help system. *helpFile* points to a string containing the directory path and name of the help file. *command*, which indicates the type of help requested, can be one of the Windows Help_xxxx constants such as HELP_CONTEXT, HELP_HELPONHELP, HELP_INDEX, HELP_MULTIKEY, HELP_QUIT, or HELP_SETINDEX. *data* contains keywords that indicate the help topic items. For example, in the sample ObjectWindows file, HELP.CPP, *WinHelp* is called with the arguments HELP_CONTEXT and HELP_MENUITEMA if the *F1* key is pressed.

```
void TOwlHelpWnd::CmMenuItemA()
{
   if (F1Pressed) {
      WinHelp(HelpFile, HELP_CONTEXT, HELP_MENUITEMA);
      F1Pressed = false;
} else {
   MessageBox("In Menu Item A command", Title, MB_ICONINFORMATION);
 }
}
```

You can also include bitmaps in your Help file by referencing their file names or by copying them from the Clipboard. For more information about how to create Help files, see the online Help documentation.

Protected data members

BkgndColor

uint32 BkgndColor;

Stores the current background color set by the *TWindow::SetBkgndColor* function.

See also TWindow::SetBkgndColor

CursorModule
TModule* CursorModule;
Holds the module ID for the specified cursor. A value of 0 indicates a standard system cursor.

See also TWindow::CursorResId

CursorResId
TResId CursorResId;
Holds the cursor resource ID for the window's cursor. If the data member, *CursorModule* is 0, *CursorResId* could be one of the following IDC_Xxxx constants that represent different kinds of cursors:

Value	Meaning
IDC_ARROW	Customary arrow cursor
IDC_CROSS	Crosshair cursor
IDC_IBEAM	I-beam cursor
IDC_ICON	Unfilled icon cursor
IDC_SIZE	A smaller square in the right inside corner of a larger square
IDC_SIZENESW	Dual-pointing cursor with arrows pointing southwest and northeast
IDC_SIZENS	Dual-pointing cursor with arrows pointing south and north
IDC_SIZENWSE	Dual-pointing cursor with arrows pointing southeast and northwest
IDC_SIZEWE	Dual-pointing cursor with arrows pointing east and west
IDC_UPARROW	Vertical arrow cursor
IDC_WAIT	Hourglass cursor

See also TWindow::SetCursor

hAccel
HACCEL hAccel;
Holds the handle to the current Windows accelerator table associated with this window.

HCursor
HCURSOR HCursor;
Holds a handle to the window's cursor. The cursor is retrieved using *CursorModule* and *CursorResId* and set using *SetCursor*.

TransferBuffer
void* TransferBuffer;
TransferBuffer points to a buffer to be used in transferring data in and out of the *TWindow*. A *TWindow* assumes that the buffer contains data used by the windows in its child list. If 0, no data is to be transferred.

See also tdxxxx constants

Protected constructor

Constructor
TWindow();

Constructor used with virtually derived classes. Immediate derived classes must call ***Init*** before the construction of the object is finished.

Protected member functions

CleanupWindow

virtual void CleanupWindow();

Always called immediately before the HWindow becomes invalid, *CleanupWindow* gives derived classes an opportunity to clean up HWND related resources. This function is the complement to *SetupWindow*.

Override this function in your derived class to handle window cleanup. *CleanupWindow* gives derived classes a chance to clean up hwnd-related resources. Derived classes should call the base class's version of *CleanupWindow* as the last step before returning. The following example from the sample program, appwin.cpp, illustrates this process:

```
//Tell windows that we are not accepting drag and drop transactions any more and perform
   other window cleanup.
void
TAppWindow::CleanupWindow()
{
  AppLauncherCleanup();
  DragAcceptFiles(false);
  TWindow::CleanupWindow();
}
```

See also TWindow::SetupWindow

DispatchScroll

void DispatchScroll(uint scrollCode, uint thumbPos, HWND hWndCtrl);

Called by *EvHScroll* and *EvVScroll* to dispatch messages from scroll bars.

GetClassName

virtual char far* GetClassName();

Returns the Windows registration class name. The default class name is generated using the module name plus "Window." If you are registering a new class or changing the name of an existing window class, override this function in your derived class.

See also TWindow::GetWindowClass, WNDCLASS struct

GetWindowClass

virtual void GetWindowClass(WNDCLASS& wndClass);

Redefined by derived classes, *GetWindowClass* fills the supplied MS-Windows registration class structure with registration attributes, thus, allowing instances of *TWindow* to be registered. This function, along with *GetClassName*, allows Windows classes to be used for the specified ObjectWindows class and its derivatives. It sets the

fields of the passed *WNDCLASS* parameter to the default attributes appropriate for a *TWindow*. The fields and their default attributes for the class are the following:

Value	Meaning
cbClsExtra	0 (the number of extra bytes to reserve after the Window class structure). This value is not used by ObjectWindows.
cbWndExtra	0 (the number of extra bytes to reserve after the Window instance). This value is not used by ObjectWindows.
hInstance	The instance of the class in which the window procedure exists
hIcon	0 (Provides a handle to the class resource.) By default, the application must create an icon if the application's window is minimized.
hCursor	IDC_ARROW (provides a handle to a cursor resource)
hbrBackground	COLOR_WINDOW + 1 (the system background color)
lpszMenuName	0 (Points to a string that contains the name of the class's menu.) By default, the windows in this class have no assigned menus.
lpszClassName	Points to a string that contains the name of the window class.
lpfnWndProc	The address of the window procedure. This value is not used by ObjectWindows.
style	CS_DBLCLKS

The style field can contain one or more of the following values:

Value	Meaning
CS_BYTEALIGNCLIENT	Aligns the window's client on a byte boundary in the x direction. This alignment, designed to improve performance, determines the width and horizontal position of the window.
CS_BYTEALIGNWINDOW	Aligns a window on a byte boundary in the x direction. This alignment, designed to improve performance, determines the width and horizontal position of the window.
CS_CLASSDC	Allocates a single device context (DC) that's going to be shared by all of the window in the class. This style controls how multi-threaded applications that have windows belonging to the same class share the same DC.
CS_DBLCLKS	Sends a double-click mouse message to the window procedure when the mouse is double-clicked on a window belonging to this class.
CS_GLOBALCLASS	Allows an application to create a window class regardless of the instance parameter. You can also create a global class by writing a DLL that contains the window class.
CS_HREDRAW	If the size of the window changes as a result of some movement or resizing, redraws the entire window.
CS_NOCLOSE	Disables the Close option on this window's system menu.
CS_OWNDC	Enables each window in the class to have a different DC.
CS_PARENTDC	Passes the parent window's DC to the child windows.
CS_SAVEBITS	Saves the section of the screen as a bitmap if the screen is covered by another window. This bitmap is later used to recreate the window when it is no longer obscured by another window.
CS_VREDRAW	If the height of the client area is changed, redraws the entire window.

After the Windows class structure has been filled with default values by the base class, you can override this function to change the values of the Windows class structure. For example, you might want to change the window's colors or the cursor displayed.

See also TWindow::GetClassName, WNDCLASS struct

Init

void Init(TWindow* parent, const char far* title, TModule* module);

Allows for further initialization after default construction of a window in virtually derived classes.

LoadAcceleratorTable

void LoadAcceleratorTable();

Loads a handle to the window's accelerator table specified in *TWindowsAttr* structure (*Attr.AccelTable*). If the accelerator does not exist, *LoadAcceleratorTable* produces an "Unable to load accelerator table" diagnostic message.

See also TWindowAttr structure

RemoveChild

void RemoveChild(TWindow* child);

Removes a child window. This family of ObjectWindows *TWindow* functions uses the ObjectWindows list of objects rather the Window's HWND list.

See also TWindow::GetFirstChild, TWindow::GetLastChild, TWindow::Next, TWindow::Previous

SetupWindow

virtual void SetupWindow();

SetupWindow is the first virtual function called when the HWindow becomes valid. *TWindow's* implementation performs window setup by iterating through the child list, attempting to create an associated interface element for each child window object for whom autocreation is enabled. (By default, autocreation is enabled for windows and controls, and disabled for dialog boxes.) *SetupWindow* then calls *TransferData*.

SetupWindow can be redefined in derived classes to perform additional special initialization. Note that the HWindow is valid when the overridden *SetupWindow* is called and that the children's HWindows are valid after calling the base classes' *SetupWindow* function.

The following example from the sample program, appwin.cpp, illustrates the use of an overridden *SetupWindow* to setup a window, initialize .INI entries, and tell Windows that we want to accept drag and drop transactions.

```
void TAppWindow::SetupWindow()
{
  TFloatingFrame::SetupWindow();
  InitEntries();   // Initialize .INI entries.
  RestoreFromINIFile(); // from Applaunc.ini in the startup directory
  UpdateAppButtons();
  DragAcceptFiles(true);
}
```

See also TFrameWindow::SetupWindow, TComboBox::SetupWindow, TWindow::CleanupWindow

Response table entries

Response table entry	Member function
EV_WM_CREATE	EvCreate
EV_WM_CLOSE	EvClose
EV_WM_DESTROY	EvDestroy
EV_WM_SIZE	EvSize
EV_WM_MOVE	EvMove
EV_WM_NCDESTROY	EvNcDestroy
EV_WM_QUERYENDSESSION	EvQueryEndSession
EV_WM_COMPAREITEM	EvCompareItem
EV_WM_DELETEITEM	EvDeleteItem
EV_WM_DRAWITEM	EvDrawItem
EV_WM_MEASUREITEM	EvMeasureItem
EV_WM_CHILDINVALID	EvChildInvalid
EV_WM_VSCROLL	EvVScroll
EV_WM_HSCROLL	EvHScroll
EV_WM_PAINT	EvPaint
EV_WM_SETCURSOR	EvSetCursor
EV_WM_LBUTTONDOWN	EvLButtonDown
EV_COMMAND(CM_EXIT, CmExit)	CmExit
EV_WM_SYSCOLORCHANGE	EvSysColorChange
EV_WM_KILLFOCUS	EvKillFocus
EV_WM_ERASEBKGND	EvEraseBkgnd
EV_MESSAGE(WM_CTLCOLORMSGBOX, EvWin32CtlColor)	EvWin32CtlColor[1]
EV_MESSAGE(WM_CTLCOLOREDIT, EvWin32CtlColor)	EvWin32CtlColor[1]
EV_MESSAGE(WM_CTLCOLORLISTBOX, EvWin32CtlColor)	EvWin32CtlColor[1]
EV_MESSAGE(WM_CTLCOLORBTN, EvWin32CtlColor)	EvWin32CtlColor[1]
EV_MESSAGE(WM_CTLCOLORDLG, EvWin32CtlColor)	EvWin32CtlColor[1]
EV_MESSAGE(WM_CTLCOLORSCROLLBAR, EvWin32CtlColor)	EvWin32CtlColor[1]
EV_MESSAGE(WM_CTLCOLORSTATIC, EvWin32CtlColor)	EvWin32CtlColor[1]

1. These response table entries are used only under the Win32 API.

TWindowAttr struct

window.h

Holds *TWindow* attributes set during construction of a window. Your program controls a window's creation by passing these values to one of *TWindow*'s creation routines. If the window is streamed, these attributes are also used for re-creation.

Public data members

AccelTable

TResID AccelTable;

Holds the resource ID for the window's accelerator table.

See also TApplication::HAccTable, TWindow::LoadAcceleratorTable

ExStyle

uint32 ExStyle;

Contains the extended style values of your window. These can be any one of the extended style constants (WS_EX_DLGMODALFRAME, WS_EX_NOPARENTNOTIFY, WS_EX_TOPMOST, WS_EX_SHADOW). See *TWindow::AdjustWindowRectEx* for a description of these constants.

See also TWindow::AdjustWindowRectEx

Id

int Id;

Contains the identifier of the child window. For a dialog box control, *Id* is its resource identifier. If Win32 is defined, *Id* is set to *GetWindowLong;* otherwise *Id* is set to *GetWindowWord.*

Menu

TResId Menu;

Contains the resource ID for the menu associated with this window. If no menu exists, Menu is 0.

Param

char far* Param;

Contains a value that is passed to Windows when the window is created. This value identifies a data block that is then available in the message response functions associated with WM_CREATE. *Param* is used by *TMDIClient* and can be useful when converting non-ObjectWindows code.

Style

uint32 Style;

Contains the values that define the style, shape, and size of your window. Although *TWindow* sets *Attr.Style* to WS_CHILD and WS_VISIBLE, you can also use other combinations of the following style constants:

Value	Meaning
WS_BORDER	Creates a window with a thin lined border
WS_CAPTION	Creates a window with a title bar.
WS_CHILD	Creates a child windows. Cannot be used with popup styles.
WS_CHILDWINDOW	Creates a child window.
WS_CLIPCHILDREN	Used when creating a parent window. Excludes the area occupied by child windows when drawing takes place within the parent window.
WS_CLIPSIBLINGS	Clips child windows relative to the child window that receives a paint message.

Value	Meaning
WS_DISABLED	Creates a window that cannot receive user input.
WS_DLGFRAME	Creates a window having a typical dialog box style (without a title bar).
WS_GROUP	Indicates the first control in a group of controls, which the user can change by pressing the direction keys.
WS_HSCROLL	Window has a horizontal scroll bar.
WS_MAXIMIZE	Window is initially maximized.
WS_MAXIMIZEBOX	Window has a maximize button.
WS_MINIMIZE	Window is initially minimized.
WS_MINIMIZEBOX	Window has a minimize button.
WS_OVERLAPPED	Creates an overlapped window with a title bar and a border.
WS_OVERLAPPEDWINDOW	Overlapped window has the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles.
WS_POPUP	Creates a popup window. Cannot be used with child window styles.
WS_POPUPWINDOW	Creates a popup window with WS_BORDER, WS_POPUP, and WS_SYSMENU styles. The WS_CAPTION and WS_POPUPWINDOW styles combine to create a system menu.
WS_SYSMENU	Window has a system menu box in its title bar. Must also indicate the WS_CAPTION style.
WS_TABSTOP	Control can receive the keyboard focus when TAB key is pressed.
WS_THICKFRAME	Window has a border that lets you change the window size.
WS_VISIBLE	Window is initially visible.
WS_VSCROLL	Window has a vertical scroll bar.

See also TWindow::IsWindowVisible

X, Y, W, H

int X, Y, W, H;

X and *Y* contain the screen coordinates of the top left corner of the window. *W* and *H* contain the width and height values of the window.

TWindowFlag enum — window.h

TWindowFlag enum

enum TWindowFlag {
 wfAlias, wfAutoCreate, wfFromResource, wfShrinkToClient, wfMainWindow, wfFullyCreated, wfStreamTop, wfPredefinedClass, wfTransfer, wfUnHidden, wfUnDisabled,
};

Define bit masks for the internally used flag attributes of *TWindow*. A *wfXxxx* mask is defined for each of these attributes:

Table 2.11 TWindow attribute masks

Constant	Meaning if set
wfAlias	The window object is an alias for an existing HWND.
wfAutoCreate	Create the *HWindow* when the parent window is created.

Constant	Meaning if set
wfFullyCreated	Window is fully created and not being destroyed.
wfFromResource	*HWindow* comes from an HWND created from a resource definition.
wfMainWindow	Window is a main window.
wfPredefinedClass	The window belongs to a predefined Window's not an ObjectWindows' class.
wfShrinkToClient	Tells a frame window to shrink itself to fit around the client window.
wfStreamTop	Indicates the topmost window of the collection of windows to be streamed.
wfTransfer	Participates in the *Transfer* mechanism.
wfUnDisabled	Temporarily used when an MDI child window is destroyed.
wfUnHidden	Temporarily used when an MDI child window is destroyed.

See also TWindow::CreateChildren, TWindow::EnableAutoCreate

TWindowDC class

dc.h

Derived from *TDC*, *TWindowDC* is a DC class that provides access to the entire area owned by a window. This is the base class for any DC class that releases its handle when it is finished.

See also

TWindowDC::Wnd

Public constructor and destructor

Constructor

```
TWindowDC(HWND wnd);
```

Creates a *TWindow* object with the given owned window. The data member *Wnd* is set to *wnd*.

Destructor

```
~TWindowDC();
```

Destroys this object.

Protected constructor

Constructor

```
TWindowDC();
```

Used for derived classes only.

Protected data member

Wnd

```
HWND Wnd;
```

Holds a handle to the window owned by this DC.

See also TWindowDC::TWindowDC

TWindowView class

docview.h

Derived from both *TWindow* and *TView*, *TWindowView* is a streamable base class that can be used for deriving window-based views. *TWindowView*'s functions override *TView*'s virtual function to provide their own implementation. By deriving a window-view class from *TWindow* and *TView*, you add window functionality to the view of your document.

Public constructor and destructor

Constructor

TWindowView (TDocument& doc, TWindow* parent = 0);

Constructs a *TWindowView* interface object associated with the window view. Sets *ViewId* to *NextViewId*. Calls the associated document's *AttachView* to attach the view to the document.

Destructor

~TWindowView();

Destroys a *TWindowView* object and calls the associated document's *DetachView* function to detach the view from the associated document.

Public member functions

CanClose

bool CanClose();

Overrides *TView::CanClose* and returns nonzero if the window can be closed. Checks all of the associated document's *CanClose* functions. These must return nonzero before the window view can be closed.

See also TWindow::CanClose

GetViewName

const char far* GetViewName();

Overrides *TView::GetViewName* and returns *StaticName*, the name of the view.

See also TView::GetViewName

GetWindow

TWindow* GetWindow()

Overrides *TView::GetWindow* and returns the *TWindowView* object as a *TWindow*.

See also TEditView::GetWindow, TView::GetWindow

SetDocTitle

bool SetDocTitle(const char far* docname, int index)

Overrides *TView::SetDocTitle* and stores the document title. This name is forwarded up the parent chain until a *TFrameWindow* object accepts the data and displays it in its caption.

See also TView::SetDocTitle, TWindow::SetDocTitle

StaticName

static const char far* StaticName();

Returns "Window View," the descriptive name of the view. This title is displayed in the user-interface box.

Response table entries

Response table entry	Member function
EV_VN_ISWINDOW	*VnIsWindow*

TWindow::TXWindow class

window.h

A nested class, *TXWIndow* describes an exception that results from trying to create an invalid window.

Public constructors

Constructors

Form 1 TXWindow(TWindow* win = 0, uint resourceId = IDS_INVALIDWINDOW);

Constructs a *TXWindow* object with a default resource ID of IDS_INVALIDWINDOW.

Form 2 TXWindow(const TXWindow& src);

Constructs a *TXWindow* object with the window that failed.

Public data members

Window

TWindow* Window;

Points to the window object that is associated with the exception.

Public member functions

Clone

TXOwl* Clone();

Makes a copy of the exception object. *Clone()* must be implemented in any class derived from *TXOwl*.

See also TXOwl

Msg
static string Msg(TWindow*, uint resourceid);
Converts the resource ID to a string and returns the string message.

Throw
void Throw();
Throws the exception object. *Throw()* must be implemented in any class derived from *TXOwl*.

See also TXOwl

Unhandled
int Unhandled(TModule* app, unsigned promptResId);
Called if an exception caught in the window's message loop has not been handled. *Unhandled()* deletes the window. This type of exception can occur if a window cannot be created.

TXCompatibility class

except.h

Describes an exception that results from setting *TModule::Status* to nonzero. This exception is included for backward compatibility with ObjectWindows 1.0.

Public constructors

Constructors
Form 1 TXCompatibility(int statusCode);
Constructs a *TXCompatibility* object.

Form 2 TXCompatibility(const TXCompatibility& src);
Constructs a *TXCompatibility* object with the window that failed.

Public member functions

Clone
TXOwl* Clone();
Makes a copy of the exception object. *Clone* must be implemented in any class derived from *TXOwl*.

MapStatusCodeToString
static string MapStatusCodeToString(int statusCode);
Retrieves *Tmodule's* status code and coverts it to a string.

Throw
void Throw();
Throws the exception object. *Throw* must be implemented in any class derived from *TXOwl*.

Unhandled

int Unhandled(TModule* app, unsigned promptResId);

If an exception caught in the message loop has not been handled, *Unhandled* is called. *Unhandled* deletes the window. This type of exception could occur if a window can't be created.

TMenu::TXMenu class

menu.h

A nested class, *TXMenu* describes an exception that occurs when a menu item cannot be constructed.

Public constructors

Constructors

TXMenu(unsigned resId = IDS_GDIFAILURE);

Constructs a *TXMenu* exception object with a default IDS_GDIFAILURE message.

Public member functions

Clone

TXOwl* Clone();

Makes a copy of the exception object.

Throw

void Throw();

Throws a *TXMenu* exception.

TXOutOfMemory class

except.h

Describes an exception that results from running out of memory.

Public constructors

Constructors

TXOutOfMemory();

Constructs a *TXOutOfMemory* object.

Public member functions

Clone

TXOwl* Clone();

Makes a copy of the exception object. *Clone* must be implemented in any class derived from *TXOwl*.

Throw

void Throw();

Throws the exception object. *Throw* must be implemented in any class derived from *TXOwl*.

TXOwl class

except.h

TXOwl is a parent class for several classes designed to describe exceptions, that is, abnormal conditions outside the program's control. In most cases, you will derive a new class from *TXOwl* instead of using this one directly. The ObjectWindows classes derived from *TXOwl* include

TXCompatibility—Describes an exception that occurs if *TModule::Status* is not zero.

TValidator::TXValidator—Describes an exception that occurs if there is an invalid validator expression.

TWindow::XWindow—Describes an exception that results from trying to create an invalid window.

TGdiObject::TXGdi—Describes an exception that results from creating an invalid GDI object.

TApplication::TXInvalidMainWindow—Describes an exception that results from creating an invalid main window.

TMenu::XMenu—Describes an exception that occurs when a menu object is invalid.

TModule::XInvalidModule—Describes an exception that occurs if a *TModule* object is invalid.

TPrinter::TXPrinter—Describes an exception that occurs if a printer DC is invalid.

TXOutOfMemory—Describes an exception that occurs if an out of memory error occurs.

Each of the exception classes describes a particular type of exception. When your program encounters a given situation that's likely to produce this exception, it passes control to the specified exception-handling object. If you use exceptions in your code, you can avoid having to scatter error-handling procedures throughout your program.

To create an exception handler, place the keyword **try** before the block of code that might produce the abnormal condition (the code that might generate an exception object) and the keyword **catch** before the block of code that follows the **try** block. If an exception is thrown within the **try** block, the classes within each of the subsequent **catch** clauses are checked in sequence. The first one that matches the class of the exception object is executed.

The following example from MDIFILE.CPP, a sample program on your distribution disk, shows how to set up a **try/catch** block around the code that might throw an exception.

```
void TMDIFileApp::CmRestoreState()
{
```

```
    char* errorMsg = 0;
    ifpstream is(DskFile);
    if (is.bad())
      errorMsg = "Unable to open desktop file.";
  // try block of code //
    else {
    if (Client->CloseChildrenParen) {
      try {
        is >>* this;
        if (is.bad())
          errorMsg = "Error reading desktop file.";
        else
          Client->CreateChildren();
        }
  // catch block of code //
    catch (xalloc) {
      Client->CloseChildren();
      errorMsg = "Not enough memory to open file.";
      }
    }
  }
   if (errorMsg)
    MainWindow->MessageBox(errorMsg, "Error",
      MB_OK | MB_ICONEXCLAMATION);
  }
```

See also
TXBase

Public constructors and destructor

Constructors
Form 1 TXOwl(const string& msg, unsigned resId = 0);
Constructs a *TXOwl* object with a string message (*msg*).

Form 2 TXOwl(unsigned resId, TModule* module = ::module);
Loads the string resource identified by the *resId* parameter and uses this resource to initialize the *TXBase* object.

Destructor
~TXOwl();
Destroys a *TXOwl* object.

Public data member

ResId
unsigned ResId;
Resource ID for a *TXOwl* object.

Public member functions

Clone

TXOwl* Clone();

Makes a copy of the exception object. *Clone* must be overridden in any class derived from *TXOwl*.

GetErrorCode

unsigned GetErrorCode () const;

Returns the resource ID.

ResourceIdToString

static string ResourceIdToString(bool* found, unsigned resId, TModule* module = ::module);

Converts the resource ID to a string and returns a string that identifies the exception. If the string message cannot be loaded, returns a "not found" message. Sets the *found* parameter to TRUE if the resource is located; otherwise, sets *found* to FALSE.

Throw

void Throw();

Throws the exception object. *Throw* must be implemented in any class derived from *TXOwl*.

Unhandled

virtual int Unhandled(TModule* app, unsigned promptResId);

Unhandled is called when an unhandled exception is caught at the main message loop level.

vnxxxx view notification constants

docview.h

The *view notification* constants are used to notify the view of a given event.

Constant	Meaning
vnCommit	Changes are committed to the document.
vnCustomBase	Base event for document notifications.
vnDocOpened	Document has been opened.
vnDocClosed	Document has been closed.
vnIsDirty	Is true if uncommitted changes are present.
vnInvalidate	Is true if uncommitted changes are present.
vnIsWindow	Is true if the HWND passed belongs to this view.
vnRevert	Document's previous data is reloaded and overwrites the view's current data.
vnViewOpened	A new view has been constructed.
vnViewClosed	A view is about to be destroyed.

See also TListView, TEditView

Chapter

3

ObjectWindows event handlers

These topics include several tables that list predefined ObjectWindows response-table macros and event-handling functions. Each table lists the name of the ObjectWindows macro, any required macro arguments, and the associated event-handling function.

These button macros handle BN_xxxx notification codes. To determine the name of the notification code that corresponds to the EV_XXXX macro, remove the EV_ prefix.

Table 3.1 Button notification messages

Macro	Macro arguments	Response function declaration
EV_BN_CLICKED	ID, *UserName*	`void UserName()`
EV_BN_DISABLE	ID, *UserName*	`void UserName()`
EV_BN_DOUBLECLICKED	ID, *UserName*	`void UserName()`
EV_BN_HILITE	ID, *UserName*	`void UserName()`
EV_BN_PAINT	ID, *UserName*	`void UserName()`
EV_BN_UNHILITE	ID, *UserName*	`void UserName()`

The following macros handle messages that a child window sends to its parent:

Table 3.2 Child ID notification messages

Macro	Macro arguments	Response function declaration
EV_CHILD_NOTIFY_AND_CODE	ID, Code, *UserName*	`void UserName(WPARAM)`
EV_CHILD_NOTIFY_ALL_CODES	ID, *UserName*	`void UserName(UINT)`
EV_NOTIFY_AT_CHILD	Code, *UserName*	`void UserName`
EV_CHILD_NOTIFY	ID, Code, *UserNam*	`void UserName`

These combo box macros handle CBN_xxxx notification codes. To determine the name of the notification code that corresponds to the EV_XXXX macro, remove the EV_ prefix.

Table 3.3 Combo box notification messages

Macro	Macro arguments	Response function declaration
EV_CBN_CLOSEUP	ID, *UserName*	`void UserName()`
EV_CBN_DBLCLK	ID, *UserName*	`void UserName()`
EV_CBN_DROPDOWN	ID, *UserName*	`void UserName()`
EV_CBN_EDITCHANGE	ID, *UserName*	`void UserName()`
EV_CBN_EDITUPDATE	ID, *UserName*	`void UserName()`
EV_CBN_ERRSPACE	ID, *UserName*	`void UserName()`
EV_CBN_KILLFOCUS	ID, *UserName*	`void UserName()`
EV_CBN_SELCHANGE	ID, *UserName*	`void UserName()`
EV_CBN_SELENDCANCEL	ID, *UserName*	`void UserName()`
EV_CBN_SELENDOK	ID, *UserName*	`void UserName()`
EV_CBN_SETFOCUS	ID, *UserName*	`void UserName()`

These macros handle WM_COMMAND messages:

Table 3.4 Command messages

Macro	Macro arguments	Response function declaration
EV_COMMAND	CMD ID, *UserName*	`void UserName()`
EV_COMMAND_AND_ID	CMD ID, *UserName*	`void UserName(WPARAM)`
EV_COMMAND_ENABLE	CMD ID, *UserName*	`void UserName(TCommandEnabler&)`

These macros handle messages generated by the document manager:

Table 3.5 Document manager messages

Macro	Macro arguments	Response function declaration
EV_OWLDOCUMENT	ID, *UserName*	`void UserName(TDocument& document)`
EV_OWLNOTIFY	ID, *UserName*	`bool UserName(LPARAM&)`
EV_OWLVIEW	ID, *UserName*	`void UserName(TView& view)`

These macros handle view-related messages generated by the document manager. *VnHandler* is a generic term for the view notification handler function.

Table 3.6 Document view messages

Macro	Response function declaration
EV_VN_VIEWOPENED	`bool VnViewOpened(TView* view)`
EV_VN_VIEWCLOSED	`bool VnViewClosed(TView* view)`
EV_VN_DOCOPENED	`bool VnDocOpened(int openMode)`
EV_VN_DOCCLOSED	`bool VnDocClosed(int openMode)`
EV_VN_COMMIT	`bool VnCommit(bool force)`
EV_VN_REVERT	`bool VnRevert(bool clear)`

Table 3.6 Document view messages (continued)

Macro	Response function declaration
EV_VN_ISDIRTY	`bool VnIsDirty()`
EV_VN_ISWINDOW	`bool VnIsWindow(HWND hWnd)`

These edit control macros handle EN_xxxx notification codes. To determine the name of the notification code that corresponds to the EV_XXXX macro, remove the EV_ prefix.

Table 3.7 Edit control notification messages

Macro	Macro arguments	Response function declaration
EV_EN_CHANGE	ID, *UserName*	`void UserName()`
EV_EN_ERRSPACE	ID, *UserName*	`void UserName()`
EV_EN_HSCROLL	ID, *UserName*	`void UserName()`
EV_EN_KILLFOCUS	ID, *UserName*	`void UserName()`
EV_EN_MAXTEXT	ID, *UserName*	`void UserName()`
EV_EN_SETFOCUS	ID, *UserName*	`void UserName()`
EV_EN_UPDATE	ID, *UserName*	`void UserName()`
EV_EN_VSCROLL	ID, *UserName*	`void UserName()`

These list box macros handle LBN_xxxx notification codes. To determine the name of the notification code that corresponds to the EV_XXXX macro, remove the EV_ prefix.

Table 3.8 List box notification messages

Macro	Macro arguments	Response function declaration
EV_LBN_DBLCLK	ID, *UserName*	`void UserName()`
EV_LBN_ERRSPACE	ID, *UserName*	`void UserName()`
EV_LBN_KILLFOCUS	ID, *UserName*	`void UserName()`
EV_LBN_SELCANCEL	ID, *UserName*	`void UserName()`
EV_LBN_SELCHANGE	ID, *UserName*	`void UserName()`
EV_LBN_SETFOCUS	ID, *UserName*	`void UserName()`

These macros handle ObjectComponents messages.

Table 3.9 ObjectComponents messages

Macro	Response function declaration
EV_OC_VIEWPARTINVALID	`bool EvOcViewPartInvalid(TOcPart far&)`
EV_OC_VIEWTITLE	`const char far* EvOcViewTitle()`
EV_OC_VIEWBORDERSPACEREQ	`bool EvOcViewBorderSpaceReq(TRect far* rect)`
EV_OC_VIEWBORDERSPACESET	`bool EvOcViewBorderSpaceSet(TRect far&)`
EV_OC_VIEWDROP	`bool EvOcViewDrop (TOcDragDrop far&)`
EV_OC_VIEWDRAG	`bool EvOcViewDrag(TOcDragDrop far&)`
EV_OC_VIEWSCROLL	`bool EvOcViewScroll (TOcScrollDir)`
EV_OC_VIEWGETSCALE	`bool EvOcViewGetScale(TOcScaleFactor&)`
EV_OC_VIEWGETSITERECT	`bool EvOcViewGetSiteRect(TRect far* )`

Table 3.9 ObjectComponents messages (continued)

Macro	Response function declaration
EV_OC_VIEWSETSITERECT	`bool EvOcViewSetSiteRect(TRect far*)`
EV_OC_VIEWPAINT	`bool EvOcViewPaint (TOcViewPaint far&)`
EV_OC_VIEWSAVEPART	`bool EvOcViewSavePart(TOcSaveLoad far& ocSave)`
EV_OC_VIEWLOADPART	`bool EvOcViewLoadPart(TOcSaveLoad far& ocLoad)`
EV_OC_VIEWINSMENUS	`bool EvOcViewInsMenus (TOcMenuDescr far&)`
EV_OC_VIEWSHOWTOOLS,	`bool EvOcViewShowTools(TOcToolBarInfo far& tbi)`
EV_OC_VIEWGETPALETTE	`bool EvOcViewGetPalette (LOGPALETTE far* far*)`
EV_OC_VIEWCLIPDATA,	`HANDLE EvOcViewClipData(TOcFormat far&)`
EV_OC_VIEWCLOSE	`bool EvOcViewClose ()`
EV_OC_VIEWPARTSIZE	`bool EvOcViewPartSize(TRect far&)`
EV_OC_VIEWOPENDOC	`bool EvOcViewOpenDoc(const char far&)`
EV_OC_VIEWATTACHWINDOW	`bool EvOcViewAttachWindow()`
EV_OC_VIEWSETSCALE	`bool EvOcViewSetScale(TOcScaleFactor&)`
EV_OC_APPINSMENUS	`bool EvOcAppInsMenus(TOcMenuDescr far&)`
EV_OC_APPMENUS	`bool EvOcAppMenus(TOcMenuDescr far&)`
EV_OC_APPPROCESSMSG	`bool EvOcAppProcessMsg(MSG far*)`
EV_OC_APPFRAMERECT	`bool EvOcAppFrameRect(TRect far*)`
EV_OC_APPBORDERSPACEREQ	`bool EvOcAppBorderSpaceReq(TRect far*)`
EV_OC_APPBORDERSPACESET	`bool EvOcAppBorderSpaceSet(TRect far*)`
EV_OC_APPSTATUSTEXT	`void EvOcAppStatusText(const char far*)`
EV_OC_APPRESTOREUI	`void EvOcAppRestoreUI()`
EV_OC_APPDIALOGHELP	`void EvOcAppDialogHelp(TOcDialogHelp far& dh)`
EV_OC_APPSHUTDOWN	`void EvOcAppShutDown()`

The following scrollbar control macros handle SB_xxxx notification codes. To determine the name of the notification code that corresponds to the EV_XXXX macro, remove the EV_ prefix.

Table 3.10 Scroll bar notification messages

Macro	Macro arguments	Response function declaration
EV_SB_LINEDOWN	ID, *UserName*	`void UserName()`
EV_SB_LINEUP	ID, *UserName*	`void UserName()`
EV_SB_PAGEDOWN	ID *UserName*	`void UserName()`
EV_SB_PAGEUP	ID, *UserName*	`void UserName()`
EV_SB_TOP	ID, *UserName*	`void UserName()`
EV_SB_BOTTOM	ID, *UserName*	`void UserName()`
EV_SB_THUMBPOSITION	ID, *UserName*	`void UserName()`
EV_SB_ENDSCROLL	ID, *UserName*	`void UserName()`
EV_SB_BEGINTRACK	ID, *UserName*	`void UserName()`

These macros handle Windows messages. These macros are defined in windowev.h. To determine the name of the Windows message that corresponds to the EV_XXXX macro,

remove the EV_ prefix. For example, WM_ACTIVATE is the name of the Windows message that the EV_WM_ACTIVATE macro handles. These macros, which crack the standard Windows messages (break the LPARAM and WPARAM parameters into separate parts), take no arguments. They pass the cracked parameters directly to the predefined *EVxxxx* message function. The standard Windows messages are described in your Windows documentation.

Table 3.11 Standard Windows messages

Macro	Response function declaration
EV_WM_ACTIVATE	`void EvActivate(uint active, bool minimized, HWND hWndOther)`
EV_WM_ACTIVATEAPP	`void EvActivateApp(bool active, HANDLE threadId)` (WIN32 only)
EV_WM_ACTIVATEAPP	`void EvActivateApp(bool active, HTASK hTask)` (WIN16 only)
EV_WM_ASKCBFORMATNAME	`void EvAskCBFormatName(uint bufLen, char far* buffer)`
EV_WM_CANCELMODE	`void EvCancelMode()`
EV_WM_CHANGECBCHAIN	`void EvChangeCBChain(uint bufLen, char far* buffer)`
EV_WM_CHAR	`void EvChar(uint key, uint repeatCount, uint flags)`
EV_WM_CHARTOITEM	`int EvCharToItem(uint key, HWND hWndListBox, uint caretPos)`
EV_WM_CHILDACTIVATE	`void EvChildActivate()`
EV_WM_CHILDINVALID	`void EvChildInvalid(HWND)`
EV_WM_CLOSE	`void EvClose()`
EV_WM_COMPACTING	`void EvCompacting(uint compactRatio)`
EV_WM_COMPAREITEM	`LRESULT EvCompareItem(uint ctrlId, COMPAREITEMSTRUCT far& compareInfo)`
EV_WM_CREATE	`int EvCreate(CREATESTRUCT far &)`
EV_WM_CTLCOLOR	`HBRUSH EvCtlColor(HDC, HWND hWndChild, uint ctlType)`
EV_WM_DEADCHAR	`void EvDeadChar(uint deadKey, uint repeatCount, uint flags)`
EV_WM_DELETEITEM	`void EvDeleteItem(uint ctrlId, DELETEITEMSTRUCT far& deleteInfo)`
EV_WM_DESTROY	`void EvDestroy()`
EV_WM_DESTROYCLIPBOARD	`void EvDestroyClipboard()`
EV_WM_DEVMODECHANGE	`void EvDevModeChange(char far* devMode)`
EV_WM_DRAWCLIPBOARD	`void EvDrawClipboard()`
EV_WM_DRAWITEM	`void EvDrawItem(uint ctrlId, DRAWITEMSTRUCT far& drawInfo)`
EV_WM_DROPFILES	`void EvDropFiles(TDropInfo dropInfo)`
EV_WM_ENABLE	`void EvEnable(bool enabled)`
EV_WM_ENDSESSION	`void EvEndSession(bool endSession)`
EV_WM_ENTERIDLE	`void EvEnterIdle(uint source, HWND hWndDlg)`
EV_WM_ERASEBKGND	`bool EvEraseBkgnd(HDC)`
EV_WM_FONTCHANGE	`void EvFontChange()`
EV_WM_GETDLGCODE	`uint EvGetDlgCode(MSG far*)`
EV_WM_GETFONT	`HFONT EvGetFont();`
EV_WM_GETMINMAXINFO	`void EvGetMinMaxInfo(MINMAXINFO far &)`
EV_WM_GETTEXT	`void EvGetText(uint bufLen, char far* buffer)`
EV_WM_GETTEXTLENGTH	`uint EvGetTextLength()` (WIN32 only)

Table 3.11 Standard Windows messages (continued)

Macro	Response function declaration
EV_WM_HOTKEY	void EvHotKey(int idHotKey)
EV_WM_HSCROLL	void EvHScroll(uint scrollCode, uint thumbPos, HWND hWndCtl)
EV_WM_HSCROLLCLIPBOARD	void EvHScrollClipboard(HWND hWndCBViewer, uint scrollCode, uint pos)
EV_WM_ICONERASEBKGND	void EvIconEraseBkgnd(HDC)
EV_WM_INITMENU	void EvInitMenu(HMENU)
EV_WM_INITMENUPOPUP	void EvInitMenuPopup(HMENU hPopupMenu, uint index, bool sysMenu)
EV_WM_INPUTFOCUS	void EvInputFocus(bool gainingFocus) (WIN32 only)
EV_WM_KEYDOWN	void EvKeyDown(uint key, uint repeatCount, uint flags)
EV_WM_KEYUP	void EvKeyUp(uint key, uint repeatCount, uint flags)
EV_WM_KILLFOCUS	void EvKillFocus(HWND hWndGetFocus)
EV_WM_LBUTTONDBLCLK	void EvLButtonDblClk(uint modKeys, TPoint& point)
EV_WM_LBUTTONDOWN	void EvLButtonDown(uint modKeys, TPoint& point)
EV_WM_LBUTTONUP	void EvLButtonUp(uint modKeys, TPoint& point)
EV_WM_MBUTTONDBLCLK	void EvMButtonDblClk(uint modKeys, TPoint& point)
EV_WM_MBUTTONDOWN	void EvMButtonDown(uint modKeys, TPoint& point)
EV_WM_MBUTTONUP	void EvMButtonUp(uint modKeys, TPoint& point)
EV_WM_MDIACTIVATE	void EvMDIActivate(HWND hWndActivated, HWND hWndDeactivated)
EV_WM_MDICREATE	LRESULT EvMDICreate(MDICREATESTRUCT far& createStruct)
EV_WM_MDIDESTROY	void EvMDIDestroy(HWND hWnd)
EV_WM_MENUCHAR	uint EvMenuChar(uint nChar, uint menuType, HMENU hMenu)
EV_WM_MENUSELECT	void EvMenuSelect(uint menuItemId, uint flags, HMENU hMenu)
EV_WM_MEASUREITEM	void EvMeasureItem(uint ctrlId, MEASUREITEMSTRUCT far& measureInfo)
EV_WM_MOUSEACTIVATE	uint EvMouseActivate(HWND hWndTopLevel, uint hitTestCode, uint msg)
EV_WM_MOUSEMOVE	void EvMouseMove(uint modKeys, TPoint& point)
EV_WM_MOVE	void EvMove(TPoint &clientOrigin)
EV_WM_NCACTIVATE	bool EvNCActivate(bool active)
EV_WM_NCCALCSIZE	uint EvNCCalcSize(bool calcValidRects, NCCALCSIZE_PARAMS far &)
EV_WM_NCCREATE	bool EvNCCreate(CREATESTRUCT far &)
EV_WM_NCDESTROY	void EvNCDestroy()
EV_WM_NCHITTEST	uint EvNCHitTest(TPoint& point)
EV_WM_NCLBUTTONDBLCLK	void EvNCLButtonDblClk(uint hitTest, TPoint& point)
EV_WM_NCLBUTTONDOWN	void EvNCLButtonDown(uint hitTest, TPoint& point)
EV_WM_NCLBUTTONUP	void EvNCLButtonUp(uint hitTest, TPoint& point)
EV_WM_NCMBUTTONDBLCLK	void EvNCMButtonDblClk(uint hitTest, TPoint& point)
EV_WM_NCMBUTTONDOWN	void EvNCMButtonDown(uint hitTest, TPoint& point)
EV_WM_NCMBUTTONUP	void EvNCMButtonUp(uint hitTest, TPoint& point)
EV_WM_NCMOUSEMOVE	void EvNCMouseMove(uint hitTest, TPoint& point)
EV_WM_NCPAINT	void EvNCPaint()

Table 3.11 Standard Windows messages (continued)

Macro	Response function declaration
EV_WM_NCRBUTTONDBLCLK	`void EvNCRButtonDblClk(uint hitTest, TPoint& point)`
EV_WM_NCRBUTTONDOWN	`void void EvNCRButtonDown(uint hitTest, TPoint& point)`
EV_WM_NCRBUTTONUP	`void EvNCRButtonUp(uint hitTest, TPoint& point)`
EV_WM_OTHERWINDOWCREATED	`void EvOtherWindowCreated(HWND hWndOther)` (WIN32 only)
EV_WM_OTHERWINDOWDESTROYED	`void EvOtherWindowDestroyed(HWND hWndOther)` (WIN32 only)
EV_WM_PAINT	`void EvPaint()`
EV_WM_PAINTCLIPBOARD	`void EvPaintClipboard(HWND, HANDLE hPaintStruct)` (WIN32 only)
EV_WM_PAINTICON	`void EvPaintIcon()`
EV_WM_PALETTECHANGED	`void EvPaletteChanged(HWND hWndPalChg)`
EV_WM_PALETTEISCHANGING	`void EvPaletteIsChanging(HWND hWndPalChg)`
EV_WM_PARENTNOTIFY	`void EvParentNotify(uint event, uint childHandleOrX, uint childIDOrY)`
EV_WM_POWER	`int EvPower(uint powerEvent)`
EV_WM_QUERYDRAGICON	`HANDLE EvQueryDragIcon()`
EV_WM_QUERYENDSESSION	`bool EvQueryEndSession()`
EV_WM_QUERYNEWPALETTE	`bool EvQueryNewPalette()`
EV_WM_QUERYOPEN	`bool EvQueryOpen()`
EV_WM_RBUTTONDBLCLK	`void EvRButtonDblClk(uint modKeys, TPoint& point)`
EV_WM_RBUTTONDOWN	`void EvRButtonDown(uint modKeys, TPoint& point)`
EV_WM_RBUTTONUP	`void EvRButtonUp(uint modKeys, TPoint& point)`
EV_WM_RENDERALLFORMATS	`void EvRenderAllFormats()`
EV_WM_RENDERFORMAT	`void EvRenderFormat(uint dataFormat)`
EV_WM_SETCURSOR	`bool EvSetCursor(HWND hWndCursor, uint hitTest, uint mouseMsg)`
EV_WM_SETFOCUS	`void EvSetFocus(HWND hWndLostFocus)`
EV_WM_SETFONT	`void EvSetFont(HFONT hFont, bool redraw)`
EV_WM_SETTEXT	`void EvSetText(char far* text)`
EV_WM_SHOWWINDOW	`void EvShowWindow(bool show, uint status)`
EV_WM_SIZE	`void EvSize(uint sizeType, TSize& size)`
EV_WM_SIZECLIPBOARD	`void EvSizeClipboard(HWND hWndViewer, HANDLE hRect)`
EV_WM_SPOOLERSTATUS	`void EvSpoolerStatus(uint jobStatus, uint jobsLeft)`
EV_WM_SYSCHAR	`void EvSysChar(uint key, uint repeatCount, uint flags)`
EV_WM_SYSCOLORCHANGE	`void EvSysColorChange()`
EV_WM_SYSCOMMAND	`void EvSysCommand(uint cmdType, TPoint& point)`
EV_WM_SYSDEADCHAR	`void EvSysDeadChar(uint key, uint repeatCount, uint flags)`
EV_WM_SYSKEYDOWN	`void EvSysKeyDown(uint key, uint repeatCount, uint flags)`
EV_WM_SYSKEYUP	`void EvSysKeyUp(uint key, uint repeatCount, uint flags)`
EV_WM_SYSTEMERROR	`void EvSystemError(uint error)`
EV_WM_TIMECHANGE	`void EvTimeChange()`
EV_WM_TIMER	`void EvTimer(uint timerId)`
EV_WM_VKEYTOITEM	`int EvVKeyToItem(uint key, HWND hWndListBox, uint caretPos)`
EV_WM_VSCROLL	`void EvVScroll(uint scrollCode, uint thumbPos, HWND hWndCtl)`

Table 3.11 Standard Windows messages (continued)

Macro	Response function declaration
EV_WM_VSCROLLCLIPBOARD	`void EvVScrollClipboard(HWND hWndCBViewer, uint scrollCode, uint pos)`
EV_WM_WINDOWPOSCHANGED	`void EvWindowPosChanged(WINDOWPOS far &windowPos)`
EV_WM_WINDOWPOSCHANGING	`void EvWindowPosChanging(WINDOWPOS far &windowPos)`
EV_WM_WININICHANGE	`void EvWinIniChange(char far* section)`

These macros handle WM_VBXFIREEVENT messages generated by VBX controls. *EvHandler* is a generic term for a specific VBX control message (such as *EvClick*).

Table 3.12 VBX messages

Macro	Macro arguments	Response function declaration
EV_VBXEVENTNAME	ID, event, *EvHandler*	`void EvHandler(VBXEVENT FAR *event)`
EV_VBXEVENTINDEX	ID, event, *EvHandler*	`void EvHandler(VBXEVENT FAR *event)`

These macros handle user-defined messages:

Table 3.13 User-defined messages

Macro	Macro arguments	Response function declaration
EV_MESSAGE	UserMessage, *UserName*	`LRESULT UserName(WPARAM,LPARAM)`
EV_REGISTERED	Registered name, *UserName*	`LRESULT UserName(WPARAM,LPARAM)`

Some event-handling functions or messages have no predefined names. In these cases, the generic term *UserName* is used to indicate that you can use any function name you want to as long as the function's signature matches the signature required by the response table macro. Similarly, the term *UserMessage* indicates that you can define your own message ID.

The descriptions of these macros and functions use the following conventions when listing the arguments of the message macros:

- *ID* refers to the child window's ID (for example, ID_GROUPBOX)
- *CMD ID* refers to the command ID (for example, CM_FILENEW)
- *Code* refers to the notification code (for example, BN_CLICKED) that is being sent.

Chapter

4

ObjectWindows dispatch functions

Dispatch functions separate the *lParam* and *wParam* parameters of Windows messages into their respective data types and pass control to an ObjectWindows member function.

For example, when Windows sends an application a WM_CTLCOLOR message, the *wParam* is really an HDC, and the *lParam* has a HWND and a **uint** hidden inside. After the dispatch function cracks the *wParam* and *lParam* parameters into their constituent parts, it dispatches the Windows WM_CTLCOLOR message to the following ObjectWindows member function:

```
HBRUSH EvCtlColor(HDC, HWND, uint);
```

Although dispatch functions are written for specific Windows API messages, they have no knowledge of the actual Windows messages they are cracking. Instead, at run time, TEventHandler::Dispatch calls the appropriate dispatch function which then cracks the message and calls the appropriate member function, using the response table's *pmf* (pointer to a member function). These dispatch functions are never called directly.

The following four parameters are common to all dispatch functions:

- *GENERIC& generic* is the pointer to the object (for example, *TEdit*).
- *GENERIC::*pmf* is the pointer to the member function (for example, *EvActivate*).
- WPARAM is one of the message parameters the dispatch function cracks.
- LPARAM is one of the message parameters the dispatch function cracks.

The name of the dispatch function used in the response table entry or macro depends on the type of message cracking the function performs. The first letter of the name indicates the return type (for example, U indicates an unsigned integer). The second group of letters signifies the arguments of the function (for example, *POINT* of type TPOINT, or

U of type **uint**). The following table lists the abbreviations of the member functions' signatures and their corresponding data types.

Abbreviation	Data type
i	**int**
v	**void**
H	HANDLE (same as **uint** in size)
I32	**int32**
POINT	*TPoint*& (*TPoint* object constructed)
POINTER	**void*** (model's ambient size)
U	**uint**

Message crackers that serve customized messages have names based on the specific message they crack. For example, the message cracker for WM_MDIACTIVATE messages has the following prototype:

```
int32_v_MdiActivate_Dispatch(GENERIC& generic, HBRUSH (GENERIC::*pmf)(uint, uint),
   uint  wParam, int32 lParam);
```

ObjectWindows uses the same dispatch functions for messages that require the same type of cracking. For example, both the WM_HSCROLL and WM_VSCROLL event handlers have `void (*)( uint, uint, HWND)` as their signature. The Windows message also has the *wParam* and *lParam* parameters in the same place. Therefore, ObjectWindows uses the same dispatch functions for both of these messages.

The following aliases have been defined for purposes of backward compatibility:

```
#define HBRUSH_HDC_W_U_Dispatch          U_U_U_U_Dispatch
#define LRESULT_U_U_W_Dispatch           I32_MenuChar_Dispatch
#define LRESULT_WPARAM_LPARAM_Dispatch   I32_WPARAM_LPARAM_Dispatch
#define v_U_B_W_Dispatch                 v_Activate_Dispatch
#define v_W_W_Dispatch                   v_MdiActivate_Dispatch
```

See also

TEventHandler

List of ObjectWindows dispatch functions

i_LPARAM_Dispatch — Dispatch.h

int32 i_LPARAM_Dispatch(GENERIC& generic, int (GENERIC::*pmf)(int32), uint wParam, int32 lParam);

Passes *lParam* as an **int32** and returns an **int**.

i_U_W_U_Dispatch — Dispatch.h

int32 i_U_W_U_Dispatch(GENERIC& generic, int (GENERIC::*pmf)(uint, uint, uint), uint wParam, int32 lParam);

For Win32, passes *wParam.lo* (the LOWORD of *wParam*) as a **uint**, *lParam* as a **uint** and *wParam.hi* (the HIWORD of *wParam*) as a **uint** and returns an **int**. For Win16, passes *wParam* as a **uint**, *lParam.lo* as a **uint** and *lParam.hi* as a **uint** and returns an **int**. This is a semi-customized dispatch function used for WM_CHARTOITEM and WM_KEYTOITEM messages.

i_WPARAM_Dispatch — Dispatch.h

int32 i_WPARAM_Dispatch(GENERIC& generic, int (GENERIC::*pmf)(uint), uint wParam, int32 lParam);
Passes *wParam* as a **uint** and returns an **int**.

I32_Dispatch — Dispatch.h

int32 I32_Dispatch(GENERIC& generic, int32 (GENERIC::*pmf)(), uint, int32);
This dispatcher passes nothing and returns an **int32**.

I32_LPARAM_Dispatch — Dispatch.h

int32 I32_LPARAM_Dispatch(GENERIC& generic, int32 (GENERIC::*pmf)(int32), uint, int32 lParam);
This dispatcher passes *lParam* as an **int32** and returns an **int32**.

I32_WPARAM_LPARAM_Dispatch — Dispatch.h

int32 I32_WPARAM_LPARAM_Dispatch(GENERIC& generic, int32 (GENERIC::*pmf)(uint, int32),
uint wParam, int32 lParam);
This dispatcher passes *wParam* as a **uint** and *lParam* as an **int32**, and returns an **int32**.

I32_MenuChar_Dispatch — Dispatch.h

int32 I32_MenuChar_Dispatch(GENERIC& generic, uint (GENERIC::*pmf) (uint, uint, uint), uint wParam,
int32 lParam);
Cracker for WM_PARENTNOTIFY messages. Passes three **uints** and returns an **int32**. Under Win32, the first **uint** is *wParam.lo*, the second **uint** is *wParam.hi*, and the **uint** is *lParam*. Under Win 16, the first **uint** is *wParam*, the second **uint** is *lParam.lo*, and the third **uint** is *lParam.hi*.

I32_U_Dispatch — Dispatch.h

int32 I32_U_Dispatch(GENERIC& generic, int32 (GENERIC::*pmf)(uint), int32 lParam);
This dispatcher passes *lParam* and **uint** and returns an **int32**.

U_Dispatch — Dispatch.h

int32 U_Dispatch(GENERIC& generic, uint (GENERIC::*pmf) (), uint wParam, int32 lParam);
Passes no arguments and returns a **uint**.

U_LPARAM_Dispatch — Dispatch.h

int32 U_LPARAM_Dispatch(GENERIC& generic, uint (GENERIC::*pmf) (int32), uint wParam, int32 lParam);
Passes *lParam* as an **int32** and returns a **uint**.

U_POINT_Dispatch — Dispatch.h

int32 U_POINT_Dispatch(GENERIC& generic, uint (GENERIC::*pmf) (TPoint&), uint wParam, int32 lParam);
Passes *lParam* as a *TPoint&* and returns a **uint**.

U_POINTER_Dispatch — Dispatch.h

int32 U_POINTER_Dispatch(GENERIC& generic, uint (GENERIC::*pmf) (void*), uint, int32 lParam);
Passes *lParam* as a **void*** and returns a **uint**.

U_U_Dispatch — Dispatch.h

int32 U_U_Dispatch(GENERIC& generic, uint (GENERIC::*pmf) (uint), uint, int32 lParam);
Passes *lParam* as a **uint** and returns a **uint**.

U_U_U_U_Dispatch — Dispatch.h

int32 U_U_U_U_Dispatch(GENERIC& generic, uint (GENERIC::*pmf) (uint, uint, uint), uint wParam, int32 lParam);
Passes three **uints** and returns a **uint**. The first **uint** is *wParam*, the second **uint** is the LOWORD of *lParam*, and the third **uint** is the HIWORD of *lParam*.

U_WPARAM_LPARAM_Dispatch — Dispatch.h

int32 U_WPARAM_LPARAM_Dispatch(GENERIC& generic, uint (GENERIC::*pmf) (uint, int32), uint wParam, int32 lParam);
Passes *wParam* as a **uint** and *lParam* as an **int32** and returns a **uint**.

v_Activate_Dispatch — Dispatch.h

int32 v_Activate_Dispatch(GENERIC& generic, void (GENERIC::*pmf) (uint, uint, uint), uint wParam, int32 lParam);
Cracker for WM_ACTIVATE messages. Passes three **uints** and always returns 0. Under Win32, the first **uint** is *wParam.lo*, the second **uint** is *wParam.hi*, and the third **uint** is *lParam*. Under Win16, the first **uint** is *wParam*, the second **uint** is *lParam.hi*, and the third **uint** is *lParam.lo*.

v_Dispatch — Dispatch.h

int32 v_Dispatch (GENERIC& generic, void (GENERIC::*pmf) (), uint uint, int32);

Passes nothing and always returns 0.

v_LPARAM_Dispatch — Dispatch.h

int32 v_LPARAM_Dispatch (GENERIC& generic, void (GENERIC::*pmf) (int32), uint wParam, int32 lParam);
Passes *lParam* as an **int32** and always returns 0.

v_MdiActivate_Dispatch — Dispatch.h

int32 v_MdiActivate_Dispatch(GENERIC& generic, void (GENERIC::*pmf)(uint, uint), uint wParam, int32 lParam);
Specifically designed to handle *EvMDIActivate* messages, *v_MdiActivate_Dispatch* passes two **uint**s and always returns 0. Under Win32, the first **uint** is *lParam* and the second **uint** is *wParam*. Under Win16, the first **uint** is *lParam.lo*. If *lParam* is nonzero, the second **uint** is *lParam.hi*; otherwise, the second **uint** is *wParam*.

v_ParentNotify_Dispatch — Dispatch.h

int32 v_ParentNotify_Dispatch(GENERIC& generic, int(GENERIC::*pmf) (uint, uint, uint), uint wParam, int32 lParam);
Performs message cracking for WM_PARENTNOTIFY messages, which notify a parent window that a given event has occurred in the child window. Passes three **uint**s and always returns 0. For Win16, the first **uint** is *wParam*, the second **uint** is *lParam.lo*, and the third **uint** is *lParam.hi*. For Win32, if *wparam.lo* is a mouse message, then the first **uint** is *wparam*, the second **uint** is *lparam.lo*, and the third **uint** is *lParam.hi*. Otherwise, the first **uint** is *wparam.lo*, the second **uint** is *lParam*, and the third **uint** is *wParam.hi*.

v_POINT_Dispatch — Dispatch.h

int32 v_POINT_Dispatch (GENERIC& generic, void (GENERIC::*pmf) (TPoint&), uint wParam, int32 lParam);
Passes a reference to a *POINT* (*TPoint&*) and always returns 0. Under Win32, passes *lParam*; under Win16, passes a reference to *lParam*.

v_POINTER_Dispatch — Dispatch.h

int32 v_POINTER_Dispatch(GENERIC& generic, void (GENERIC::*pmf)(void*), uint wParam, int32 lParam);
Passes an *lParam* as a **void***pointer and always returns 0.

v_U_Dispatch — Dispatch.h

int32 v_U_Dispatch(GENERIC& generic, void (GENERIC::*pmf) (uint, int32 lParam);
Passes *lParam* as a **uint** and always returns 0.

v_U_POINT_Dispatch — Dispatch.h

int32 v_U_POINT_Dispatch(GENERIC& generic, void (GENERIC::*pmf) (uint, TPoint&), uint wParam, int32 lParam);

Passes *wparam* as a **uint** and *lParam* as a reference to a POINT *and always returns 0.*

v_U_U_Dispatch — Dispatch.h

int32 v_U_U_Dispatch(GENERIC& generic, void (GENERIC::*pmf) (uint, uint), uint wParam, int32 lParam);

Passes two **uint** and always returns 0. The first **uint** is *wParam* and the second **uint** is *lParam.*

v_U_U_U_Dispatch — Dispatch.h

int32 v_U_U_U_Dispatch(GENERIC& generic, void (GENERIC::*pmf) (uint, uint, uint), uint wParam, int32 lParam);

Passes three **uints** and always returns 0. The first **uint** is *wParam,* the second **uint** is *lParam.lo* and the third **uint** is *lParam.hi.*

v_U_U_W_Dispatch — Dispatch.h

int32 v_U_U_W_Dispatch(GENERIC& generic, void (GENERIC::*pmf) (uint, uint, uint), uint wParam, int32 lParam);

Passes three **uints** and always returns 0. Under Win32, the first **uint** is *wParam.lo.*, the second **uint** is *wParam.hi,* and the third **uint** is *lParam* Under Win16, the first **uint** is *wParam,* the second **uint** is *lParam.lo,* and the third **uint** is *lParam.hi.* This is a semi-customized message cracker for WM_HSCROLL, WM_VSCROLL, and WM_MENUSELECT messages.

v_WPARAM_Dispatch — Dispatch.h

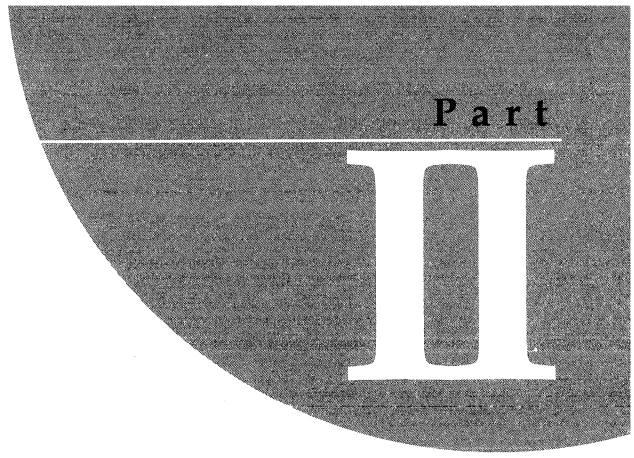
int32 v_WPARAM_Dispatch(GENERIC& generic, void (GENERIC::*pmf) (uint), uint wParam, int32 lParam);

Passes *wParam* as a **uint** and always returns 0.

v_WPARAM_LPARAM_Dispatch — Dispatch.h

int32 v_WPARAM_LPARAM_Dispatch(GENERIC& generic, void (GENERIC::*pmf) (uint, int32), uint wParam, int32 lParam);

Passes *wParam* as a **uint** and *lParam* as an **int32** and always returns 0.



ObjectComponents reference

Chapter

5

Overview of ObjectComponents

This chapter lists alphabetically the ObjectComponents classes, macros, constants, data types, and registration keys. The header file that defines each entry is listed opposite the entry name. Class members are grouped according to their access specifiers—**public** or **protected**. Within these categories, data members, then constructors and destructors, and member functions are grouped separately and listed alphabetically.

The members listed for each class include only those that are new or redefined in the class itself. Members inherited from a base class are not listed again in the derived class. No private members are listed. In some cases members that are public or protected are omitted as well because they are meant only for the use of other ObjectComponents classes.

Some entries related to ObjectComponents programs appear in other sections of this book. OLE-enabled ObjectWindows classes, such as *TOleWindow* and *TOleFactory*, are listed with the other ObjectWindows classes. Registration macros and some locale items are listed later in the Object Support Library (OSL) reference.

ObjectComponents libraries

These are the libraries an ObjectComponents application uses for linking.

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

Medium model	Large model	DLL libraries	Description
MATHWM.LIB	MATHWL.LIB		Math support
CWM.LIB	CWL.LIB	CRTLDLL.LIB	C run-time libraries

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 result codes, declaration specifiers
ocdoc.h	*TOcDocument*
ocfevx.h	Message cracker macros for WM_OCEVENT messages
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

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

Item	Meaning
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

ObjectComponents exception classes

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

Automation classes

ObjectComponents provides the following classes that support automation.

Class	Purpose
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

Class	Purpose
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

Automation enumerated types and type definitions

ObjectComponents provides the following items that support automation.

Item	Purpose
AutoDataType enum	Identifiers for automation data types
ObjectPtr typedef	**void** pointer to a C++ object

Automation data types

ObjectComponents provides the following data types that support automation. To use these data types, see Declarations and Definitions.

Data type
TAutoBool struct
TAutoCurrency struct
TAutoDate struct
TAutoDouble struct
TAutoFloat struct
TAutoInt
TAutoLong struct
TAutoShort struct
TAutoString struct
TAutoStruct struct
TAutoVoid struct

Declarations and definitions of automation data types ocf/autodefs.h

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.

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.)

C++ type	Declaration type	Definition type
short	short	TAutoShort
unsigned short	short or unsigned	TAutoShort or TAutoLong
long	long	TAutoShort
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
TAutoDate far *	TAutoDate far*	TAutoDateRef
TAutoCurrency far*	TAutoCurrency far*	TAutoCurrencyRef
char*	TAutoString	TAutoString
const char*	TAutoString	TAutoString
char far*	TAutoString	TAutoString
const char far*	TAutoString	TAutoString
string	string	TAutoString
enum	short or int	TAutoShort or user-defined AUTOENUM
T*	TAutoObject<T>	T (class T must be automated)
T&	TAutoObject<T>	T (class T must be automated)
const T*	TAutoObject<const T>	T (class T must be automated)
const T&	TAutoObject<const T>	T (class T must be automated)
T* (returned)	TAutoObjectDelete<T>	(C++ object deleted if no refs)
T& (returned)	TAutoObjectDelete<T>	(C++ object deleted if no refs)
T (returned)	TAutoObjectByVal<T>	T (T copied, deleted when refs==0)
void (no return)	(use AUTOFUNCxV macros)	TAutoVoid
short far*	short far*	TAutoShortRef
long far*	long far*	TAutoLongRef
float far*	float far*	TAutoFloatRef
double far*	double far*	TAutoDouble Ref

Automation declaration macros

ocf/automacr.h

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. 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*.

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
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.

Automation definition macros

ocf/automacr.h

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.

Macro	Meaning
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

Automation hook macros

ocf/automacr.h

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.

Macro	Meaning
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)));
```

Automation proxy macros

ocf/automacr.h

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

Macro	Description
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

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 macro, but there are others such as REGFORMAT and REGSTATUS.

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
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.

Key	Meaning
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.

Linking and embedding classes

ObjectComponents provides the following classes for use by applications that support linking and embedding.

Class	Purpose
TOcApp	Connector object that implements BOCOLE interfaces for the application
TOcDocument	Manages the parts in a container's compound document
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*)
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
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

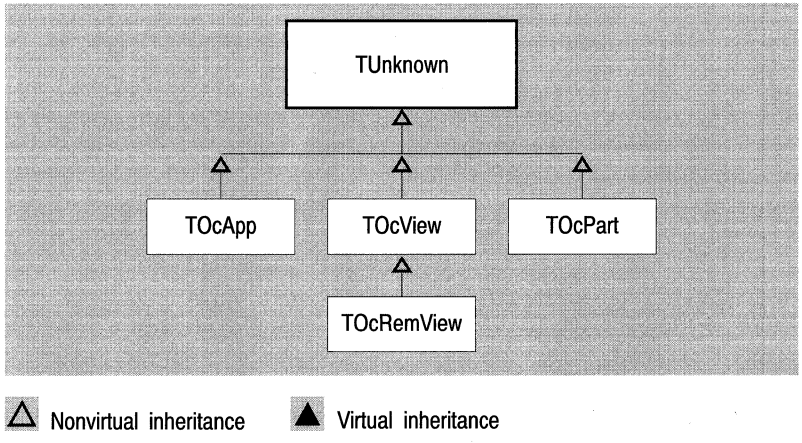
Class	Purpose
TOcRemView	Connector object that a server uses to draw an object linked or embedded in a container's document
TOcScaleFactor	Carries information from a container to a server requesting that linked or embedded objects be drawn to a certain scale
TOcVerb	Holds information about an action 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

A few of the ObjectComponents classes used for linking and embedding 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 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.

Figure 5.1 shows how the connector objects are related.

Figure 5.1 Hierarchy of ObjectComponents connector classes



Linking and embedding enums

ObjectComponents provides the following enumerated types for use by applications that support linking and embedding.

Item	Description
TOcAppMode enum	Flags identifying the application's running conditions
TOcAspect enum	Flags identifying object presentation aspects
TOcDialogHelp enum	Constants identifying standard OLE dialog boxes where a user can ask for help
TOcDropAction enum	Constants identifying actions that can result from dropping an object on a window
TOcInitHow enum	Constants identifying the action a container is to take on receiving a new object—either link or embed
TOcInitWhere enum	Constants identifying places the data for an object can reside
TOcInvalidate enum	Flags indicating whether an object is invalid because of a change in its data or just in its appearance
TOcPartName enum	Constants identifying different strings a container might request when asking for the name of an object linked or embedded in it
TOcScrollDir enum	Constants identifying directions a container might be asked to scroll its window

Linking and embedding messages

ObjectComponents provides the following messages for use by applications that support linking and embedding.

Message	Purpose
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

Linking and embedding structs

ObjectComponents provides the following structs for use by applications that support linking and embedding.

Item	Purpose
TOcDragDrop struct	Holds information a container needs in order to receive an object dropped on its window
TOcInitInfo struct	Holds information a container needs in order to place a new object in its document
TOcMenuDescr struct	Holds information about a shared menu where the container and server merge their commands for in-place editing
TOcSaveLoad struct	Carries information an application needs to save or load a linked or embedded object

Item	Purpose
TOcToolbarInfo struct	Carries handles to a server's tool bars to be displayed in the container's window during in-place editing
TOcViewPaint struct	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

ocf/ocreg.h

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)

Chapter

6

ObjectComponents library reference

_ICLASS macro

ocf/oleutil.h

Modifies the declaration of an interface class, one that defines or implements an interface for OLE or for the BOCOLE support library.

_IFUNC macro

ocf/oleutil.h

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.

_OCFxxxx macros

ocf/ocfdefs.h

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.

aspectall registration key

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 **enum** 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 enum, REGSTATUS macro

aspectcontent registration key

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 **enum** 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 enum, REGSTATUS macro

aspectdocprint registration key

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 **enum** value as the second parameter.

See also

aspectall registration key, aspectcontent registration key, aspecticon registration key, aspectthumbnail registration key, ocrxxxx object status enum, REGSTATUS macro

aspecticon registration key

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 **enum** 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 enum, REGSTATUS macro

aspectthumbnail registration key

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 **enum** value as the second parameter.

See also

aspectall registration key, aspectcontent registration key, aspectdocprint registration key, aspecticon registration key, ocrxxxx object status enum, REGSTATUS macro

AUTOARGS macros

ocf/automacr.h

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.

The automacr.h header defines AUTOARG macros that accept up to six arguments. To generate versions that accept more arguments, use the MACROGEN.EXE utility.

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*.
AUTOARGS4(a1, a2, a3, a4)	The automation command requires arguments *a1*, *a2*, *a3*, and *a4*.

AUTOCALL_xxxx macros

ocf/automacr.h

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.

Macro	Meaning
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.

_AUTOCLASS macro

ocf/autodefs.h

_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

ocf/automacr.h

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. See Automation Data Types for more details.

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.

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.

AutoDataType enum

ocf/autodefs.h

enum AutoDataType

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**
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
atByRef	The value is a reference to an object.
atEnum	The value is an enumeration of some type.

See also

TAutoVal class, TAutoEnumTpublic constructor

AUTODETACH macro

ocf/automacr.h

AUTODETACH

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.

AUTOENUM macros

ocf/automacr.h

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.

AUTOFLAG macro

ocf/automacr.h

AUTOFLAG(name, data, mask, options)

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.

AUTOFUNC macros

ocf/automacr.h

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. See Automation Data Types for more details. 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. See Automation Hook Macros for more details. *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**.

AUTOINVOKE macro

ocf/automacr.h

AUTOINVOKE(code)

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.

AUTOITERATOR macros

ocf/automacr.h

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);
```

AUTONAMES macros

ocf/automacr.h

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, 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.

The automacr.h header defines AUTONAMES macros that accept up to ten arguments. To generate versions that accept more arguments, use the MACROGEN.EXE utility.

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*.
AUTONAMES4(fname, n1, n2, n3, n4)	Function *fname* has four named arguments, *n1*, *n2*, *n3*, and *n4*.

AUTONOHOOK macro

ocf/automacr.h

AUTONOHOOK

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.

AUTOPROP macros

ocf/automacr.h

An automation server uses AUTOPROP macros in its automation declaration (after DECLARE_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. See Automation Data Types for more information.

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.

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).

AUTORECORD macro

ocf/automacr.h

AUTORECORD(code)

An automation server uses AUTORECORD in an automation declaration macro 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.

AUTOREPORT macro

ocf/automacr.h

AUTOREPORT(code)

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.

AutoSymFlag enum

ocf/autodefs.h

enum AutoSymFlag

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).
asArgument	Property that returns an object.
asArgByVal	The value of the argument is passed.
asArgByRef	The argument is a pointer to a value.
asFactory	For creating objects or determining class.
asClass	Extension to another class symbol table.
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

ocf/automacr.h

Use the AUTOSTAT in an automation declaration (after DECLARE_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. See Automation Data Types for more information.

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.

AUTOTHIS macro

ocf/automacr.h

AUTOTHIS(name, type, options)

An automation server uses the AUTOTHIS macro in its automation declaration (after DECLARE_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. See Automation Hook Macros for more details. *options* can be omitted, but a comma must follow the preceding argument anyway.

AUTOUNDO macro

ocf/automacr.h

AUTOUNDO(code)

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.

AUTOVALIDATE macro

ocf/automacr.h

AUTOVALIDATE(condition)

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.

clsid registration key

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 documented in the Help file OLE2HELP.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}")
```

cmdline registration key

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")
```

debugclsid registration key

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

debugdesc registration key

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

debugger registration key

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

debugprogid registration key

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 program's 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

DECLARE_AUTOCLASS macro — ocf/automacr.h

DECLARE_AUTOCLASS(cls)

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.

DECLARE_COMBASES*n* macros — ocf/oleutil.h

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.

DEFINE_AUTOAGGREGATE macro

ocf/automacr.h

DEFINE_AUTOAGGREGATE(cls, AggregatorFunction)

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.

DEFINE_AUTOCLASS 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
```

See also

DEFINE_AUTOCLASS, END_AUTOAGGREGATE

DEFINE_AUTOCLASS macro

ocf/automacr.h

DEFINE_AUTOCLASS(cls)

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

END_AUTOCLASS macro

DEFINE_COMBASES*n* macros

ocf/oleutil.h

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 DECLARE_AUTOAGGREGATE macros.

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

Registers a long descriptive name, up to 40 characters, meant for the user to see. The string describes the application or its document types and appears in the Insert Object dialog box. This value should be localized.

A description string is required in every registration table. 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, "OWL Drawing Pad 2.0")
```

See also
REGDATA macro

directory registration key

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")
```

docfilter registration key

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, dt document view constants, extension registration key, REGDATA macro

docflags registration key

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, dt document view constants, REGDOCFLAGS macro

DynamicCast function — ocf/autodefs.h

const void far* DynamicCast(const void far* obj, const typeinfo& src, const typeinfo& dst);
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, typeinfo class

END_AUTOAGGREGATE macro — ocf/automacr.h

END_AUTOAGGREGATE(cls, name, doc, help)
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.

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 an .HLP file is registered for the object, then the context ID in *help* points to a screen that describes the object.

See also
DEFINE_AUTOAGGREGATE macro, END_AUTOCLASS macro

END_AUTOCLASS macro — ocf/automacr.h

END_AUTOCLASS(cls, name, doc, help)
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 an .HLP file is registered for the object, then the context ID in *help* points to a screen that describes the object.

See also
DEFINE_AUTOCLASS macro

EXPOSE_APPLICATION macro

ocf/automacr.h

EXPOSE_APPLICATION(cls, extName, doc, help)
An automation server uses EXPOSE_APPLICATION in its automation definition (after DEFINE_AUTOCLASS) 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.

EXPOSE_DELEGATE macro

ocf/automacr.h

EXPOSE_DELEGATE(cls, extName, locator)
An automation server uses EXPOSE_DELEGATE in its automation definition (after DEFINE_AUTOCLASS) 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
EXPOSE_INHERIT macro

EXPOSE_INHERIT macro

ocf/automacr.h

EXPOSE_INHERIT(cls, extName);
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
EXPOSE_DELEGATE macro

EXPOSE_ITERATOR macro

ocf/automacr.h

EXPOSE_ITERATOR(retType, doc, help);
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
AUTOITERATOR macros

EXPOSE_METHOD macros

ocf/automacr.h

```
EXPOSE_METHOD(intName, retType, extName, doc, help);
EXPOSE_METHOD_ID(id, intName, retType, extName, doc, help);
```

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
Optional	OPTIONAL_ARG
Object passed by reference	REQBYREF_ARG

See also

AUTOFUNC macros, OPTIONAL_ARG macro, REQBYREF_ARG macro, REQUIRED_ARG macro

EXPOSE_PROPxxxx macros

ocf/automacr.h

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*).

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.

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).

EXPOSE_QUIT macro

ocf/automacr.h

EXPOSE_QUIT(extName, docString, helpContext)

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.

extension registration key

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

filefmt registration key

Registers a name for a server's 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

format*n* registration key

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 medium constants, REGFORMAT macro

handler registration key

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

helpdir registration key

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, typehelp registration key

HR_xxxx return constants

ocf/ocfdefs.h

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.

Constant	Meaning
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.

iconindex registration key

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

insertable registration key

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.

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
REGDATA macro

language registration key

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.

menuname registration key

A short name for the server. The name appears as a menu item 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, "OWL Drawing Pad")
```

See also
REGDATA macro

MostDerived function

ocf/autodefs.h

const void far* MostDerived(const void far* obj, const typeinfo& src);
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, typeinfo class

ObjectPtr typedef

ocf/autodefs.h

typedef void* ObjectPtr;
ObjectPtr is a **void*** that points to a C++ object.

OC_APPxxxx messages

ocf/ocapp.h

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*, look up the corresponding event handlers (such as *EvOcAppBorderSpaceSet* and *EvOcAppMenus*.)

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

OC_VIEWxxxx messages, WM_OCEVENT message

OC_VIEWxxxx messages

ocf/ocview.h

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*, look up the corresponding event handlers (such as *EvOcViewBorderSpaceSet* and *EvOcViewDrag*.)

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	Requests border space for in-place editing tools in the container's view.
OC_VIEWBORDERSPACESET	Requests border space for in-place editing tools in the container's view.
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_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_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_VIEWSETSITERECT	Asks the container to set the site rectangle.
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

OC_APPxxxx constants

ocrxxxx aspect constants

ocf/ocreg.h

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, REGFORMAT macro, TOcAspect enum

ocrxxxx Clipboard constants

ocf/ocreg.h

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

Constant	Windows format name	Meaning
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, REGFORMAT macro

ocrxxxx direction constants
ocf/ocreg.h

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, REGFORMAT macro

ocrxxxx limit constants
ocf/ocreg.h

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
ocf/ocreg.h

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, REGFORMAT macro

ocrxxxx object status constants

ocf/ocreg.h

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, REGSTATUS macro

ocrxxxx usage constants

ocf/ocreg.h

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

ocf/ocreg.h

enum ocrVerbAttributes

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, REGVERBOPT macro

ocrxxxx verb menu flags

ocf/ocreg.h

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, REGVERBOPT macro

OPTIONAL_ARG macro

ocf/automacr.h

OPTIONAL_ARG(cls, extName, default)
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.

path registration key

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

permid registration key

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, version registration key

permname registration key

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, permid registration key, REGDATA macro, version registration key

progid registration key

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.

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
REGDATA macro

REQUIRED_ARG macro

ocf/automacr.h

REQUIRED_ARG(type, extName);
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.

TAutoBase class

ocf/autodefs.h

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.

See also
_AUTOCLASS macro

Public destructor

Destructor
virtual ~TAutoBase();
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

ocf/autodefs.h

Use *TAutoBool* in an automation definition to describe the parameters and return values of automated methods.

Public data member

ClassInfo
static TAutoType ClassInfo;
The *ClassInfo* member of *TAutoBool* holds information that identifies the Boolean data type.

TAutoCommand class

ocf/autodefs.h

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 constructor and destructor

Constructor
TAutoCommand(int attr);
Creates a command having the attributes set in the *attr* flag mask. The flags are defined in the *AutoSymFlag* enum.

Destructor
virtual ~TAutoCommand();
Destroys the *TAutoCommand* object.

See also AutoSymFlag enum

Type definitions

TCommandHook
typedef bool (*TCommandHook)(TAutoCommand& cmdObj);
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

TErrorMsgHook

```
typedef const char* (*TErrorMsgHook)(long errCode);
```

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

Public member functions

ClearFlag

```
void ClearFlag(int mask);
```

Clears all the flags in *mask*. The flags are defined in the *AutoSymFlag* enum.

See also AutoSymFlag enum

Execute

```
virtual void Execute();
```

Executes the automation command by invoking the internal C++ member of the automated class to which the command belongs.

Fail

```
void Fail(TXAuto::TError);
```

Throws whatever exception is indicated by the parameter.

See also TXAuto::TError enum

GetSymbol

```
TAutoSymbol* GetSymbol();
```

Retrieves the symbol that generates this command.

Invoke

```
virtual TAutoCommand& Invoke();
```

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

IsPropSet

```
bool IsPropSet();
```

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.

LookupError

```
static const char* LookupError(long errCode);
```

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

Record

virtual int Record(TAutoStack& q);

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

Report

virtual long Report();

The AUTOREPORT macro invokes this function to translate the status code a command returns into an error code.

See also AUTOREPORT Macro

TAutoCommand::SetErrorMsgHook

Return

virtual void Return(TAutoVal& v);

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.

SetCommandHook

static TCommandHook SetCommandHook(TCommandHook callback);

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

SetErrorMsgHook

static TErrorMsgHook SetErrorMsgHook(TErrorMsgHook callback);

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

SetFlag

void SetFlag(int mask);

Sets all the flags in *mask*. The flags are defined in the *TAutoSymFlag* enum.

See also AutoSymFlag enum

SetSymbol

void SetSymbol(TAutoSymbol* sym);

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.

TestFlag

bool TestFlag(int mask);

Returns **true** if any of the flags in *mask* are set for this command. The flags are defined in the *AutoSymFlag* enum.

See also AutoSymFlag enum

Undo

virtual TAutoCommand* Undo();

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

Validate

virtual bool Validate();

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

Protected data members

Attr

int Attr;

Attribute and state flags. The flags are defined in the *AutoSymFlag* enum.

See also AutoSymFlag enum

Symbol

TAutoSymbol* Symbol;

The symbol entry that generates this command. OLE passes this symbol to make this command execute.

TAutoCurrency struct

ocf/autodefs.h

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

ClassInfo

static TAutoType ClassInfo;

The *ClassInfo* member of *TAutoCurrency* holds information that identifies data as a currency value.

TAutoDate struct

ocf/autodefs.h

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

ClassInfo
static TAutoType ClassInfo;
The *ClassInfo* member of *TAutoDate* holds information that identifies the date data type.

Date
double Date;
Stores a date as a 32-bit value.

Public constructors

Constructors

Form 1 TAutoDate();
Creates an empty *TAutoDate*.

Form 2 TAutoDate(double d);
Creates a *TAutoDate* that initially holds the value in *d*, assumed to be a date.

Public member function

operator double
operator double();
Returns the value stored in the Date field of *TAutoDate*.

TAutoDouble struct

ocf/autodefs.h

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

ClassInfo
static TAutoType ClassInfo;
The *ClassInfo* member of *TAutoDouble* holds information that identifies the **double** data type.

TAutoEnumerator<> class

ocf/autodefs.h

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

Constructors

Form 1 `TAutoEnumerator();`

Constructs an enumerator object but does not attach it to any automated collection.

Form 2 `TAutoEnumerator(const TAutoEnumerator& copy);`

Constructs a new enumerator object by copying an existing one. Both enumerators are attached to the same collection of objects.
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.

Destructor

`~TAutoEnumerator();`

Detaches the enumerator from its collection before allowing the enumerator to be destroyed.

See also TAutoEnumerator::StepTAutoEnumerator_Step

Public member functions

Bind

`void Bind(TAutoVal& val);`

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

Clear

`void Clear();`

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

Object

void Object(TAutoProxy& prx);

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

Step

bool Step();

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

Unbind

void Unbind();

Disconnects the enumerator object from the collection it currently enumerates.

See also TAutoEnumerator::Bind

Value

void Value(T& v);

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

ocf/autodefs.h

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

ClassInfo

static TAutoType ClassInfo;

The *ClassInfo* member of *TAutoFloat* holds information that identifies the **float** data type.

TAutoIterator class

ocf/autodefs.h

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

Copy

```
virtual TAutoIterator* Copy()=0;
```

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

GetSymbol

```
TAutoSymbol* GetSymbol();
```

Retrieves the automation symbol associated with the iterator. Usually you do not need to call this function.

See also TAutoIterator::SetSymbol

Init

virtual void Init()=0;

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

IUnknown()

operator IUnknown*();

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.

Return

virtual void Return(TAutoVal& value)=0;

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

SetSymbol

void SetSymbol(TAutoSymbol* sym);

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

Step

virtual void Step()=0;

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

Test
virtual bool Test()=0;
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

Protected constructors

Constructors
Form 1 TAutoIterator (TServedObject& owner);
Constructs an iterator to enumerate items held in the *owner* class. *owner* can be any automated class.

Form 2 TAutoIterator (TAutoIterator& copy);
Constructs an iterator by creating a copy of another iterator. Both iterators enumerate the same collection of objects.

The constructors are protected because only a derived class should construct a *TAutoIterator*.

Protected data member

Owner
TServedObject& Owner;
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

ocf/autodefs.h

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

ClassInfo
static TAutoType ClassInfo;

The *ClassInfo* member of *TAutoLong* holds information that identifies the **long** data type.

TAutoObject <> class

ocf/autodefs.h

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

Constructors

Form 1 `TAutoObject();`
Constructs an empty *TAutoObject* that contains no pointer.

Form 2 `TAutoObject(T* point);`
Constructs a *TAutoObject* that holds the pointer *point*.

Form 3 `TAutoObject(T& ref);`
Constructs a *TAutoObject* that holds a pointer to the object *ref*.

Form 4 `TAutoObject(IDispatch* dispatch);`
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.

Public member functions

operator *()

`T& operator *();`
The dereference operator returns a reference to the object whose pointer *TAutoObject* holds.

operator =

Form 1 `void operator =(T* point);`
Places the *point* pointer in the *TAutoObject*.

Form 2 `void operator =(T& ref);`
Places a pointer to the object *ref* in the *TAutoObject*.

Form 3 `void operator =(IDispatch* dispatch);`
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.

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.

operator T& ()

T& operator*();

The dereference operator returns a reference to the object whose pointer *TAutoObject* holds.

operator T* ()

operator T*();

Returns a pointer to the object *TAutoObject* holds.

TObjectDescriptor()

operator TObjectDescriptor();

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.

Protected data member

operator P

T* P;

Returns the pointer that *TAutoObject* holds.

TAutoObjectByVal<> class

ocf/autodefs.h

Base class

TAutoObjectDelete

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*.

See also

TAutoObjectDelete

Public data member

operator =

void operator =(T obj);

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.

Public constructors

Constructors

Form 1 TAutoObjectByVal();
Creates an empty *TAutoObjectByVal*.

Form 2 TAutoObjectByVal(T obj);
Creates a *TAutoObjectByVal* that holds a copy of the object *obj*. *T* is the data type passed into the template.

TAutoObjectDelete <> class

ocf/autodefs.h

Base class

TAutoObject

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.

See also

TAutoObject class

Public constructors

Constructors

Form 1 TAutoObjectDelete();
Creates an empty *TAutoObjectDelete* object.

Form 2 TAutoObjectDelete(T* p);
Creates a *TAutoObjectDelete* object from a pointer to another object.

Form 3 TAutoObjectDelete(T& r);
Creates a *TAutoObjectDelete* object from a reference to another object.

The *TAutoObjectDelete* constructors do nothing but pass their parameters back to the parent class, *TAutoObject*.

Public member functions

operator =

Form 1 void operator =(T& r);
Tells *TAutoObjectDelete* to hold a pointer to the object referred to by *r*.

Form 2 void operator =(T* p);
Tells TAutoObjectDelete to hold the pointer *p*.

TObjectDescriptor()
operator TObjectDescriptor();
Returns type information describing the object.

TAutoProxy class

ocf/autodefs.h

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

Destructor
~TAutoProxy();
Destroys the *TAutoProxy* object.

The constructors are protected because only derived proxy classes should call them.

Public member functions

Bind

Form 1 void Bind(IUnknown* obj);
Binds the proxy object to a server identified by a pointer to its *IUnknown* interface. Throws a *TXOle* exception for failure.

Form 2 void Bind(IUnknown& obj);
Binds the proxy object to a server identified by a reference to its *IUnknown* interface. Throws a *TXOle* exception for failure.

Form 3 void Bind(const GUID& guid);
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 void Bind(char far* progid);
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 void Bind(TAutoVal& val);
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 void Bind(IDispatch* obj);
Accepts *obj* as the proxy object's server.

Form 7 void Bind(IDispatch& obj);
Accepts *obj* as the proxy object's server.

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.

See also TAutoProxy::Bind

IDispatch&()
operator IDispatch&();
Returns a reference to the *IDispatch* interface of the proxy object's server.

IDispatch*()
operator IDispatch*();
Returns a pointer to the *IDispatch* interface of the proxy object's server and calls *AddRef* on the interface pointer.

IsBound
bool IsBound();
Returns **true** if the server already has a pointer to the *IDispatch* interface of its server and **false** if it does not.

Lookup
Form 1 long Lookup(char far* name);
Calls the server to get the ID that matches the name.

Form 2 long Lookup(const long id);
Returns the value passed in as *id*.

Form 3 void Lookup(const char* names, long* ids, unsigned count);

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.

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.

MustBeBound

```
void MustBeBound();
```

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.

SetLang

```
void SetLang(TLangId lang);
```

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 Locale IDs, TAutoStack

Unbind

```
void Unbind();
```

Decrements the reference count of the proxy object's server and erases internal references to the server.

See also: TAutoProxy::Bind

Protected constructor

Constructor

```
TAutoProxy(TLangId lang);
```

Constructs a *TAutoProxy* object and sets the object to use the language identified by the *lang* locale ID.

See also Locale IDs

Protected member function

Invoke

```
TAutoVal& Invoke(int attr, TAutoProxyArgs& args, long* ids, unsigned named=0);
```

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

ocf/autodefs.h

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

ClassInfo

```
static TAutoType ClassInfo;
```

The *ClassInfo* member of *TAutoShort* holds information that identifies the **short** data type.

TAutoStack class

ocf/autodefs.h

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.

See also

TAutoCommand class

Public constructor and destructor

Constructor

```
TAutoStack(TServedObject& owner, VARIANT far* stack, TLocaleId locale,int argcount, int namedcount,
  long far* map);
```

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

~TAutoStack();
Destroys the *TAutoStack* object.

Public member function

operator []

TAutoVal& operator[](int index);
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.

Public data members

ArgCount

const int ArgCount;
Holds the number of arguments passed on the command stack (named or unnamed).

ArgSymbolCount

int ArgSymbolCount;
Holds the number of command arguments exposed to automation.

CurrentArg

int CurrentArg;
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.

LangId
TLangId LangId;
Holds a number that identifies the language the controller is using to send commands.

Owner
TServedObject& Owner;
Refers to the automated object that is processing the command on the stack.

Symbol
int ArgSymbolCount;
Holds the number of command arguments exposed to automation.

Constant

SetValue
TAutoStack::SetValue
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.

See also TAutoVal class, TLocaleId

TAutoString struct

ocf/autodefs.h

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

Constructors

Form 1 TAutoString(const string& s);
Creates a *TAutoString* and assigns it the string held in a C++ *string* object.

Form 2 TAutoString(const TAutoString& copy);
Creates a new *TAutoString* that holds the same string value as the *copy* object.

Form 3 TAutoString(TAutoVal& val);

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 TAutoString(const char far* str);
Initializes the new object with the value in a **const** C string.

Form 5 TAutoString(BSTR s, bool loan)
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();
TAutoString maintains a reference count on the string object it contains. The destructor decrements the reference count.

Public member functions

int()

operator int()
Returns the length of the string value (as *strlen* would calculate the length).

operator =

Form 1 TAutoString& operator =(const char far* str);
Accepts a C-style **const** string as the new value of the *TAutoString*.

Form 2 TAutoString& operator =(char* s);
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 TAutoString& operator =(const TAutoString& copy)
Sets the value of the *TAutoString* object to be a string copied from another *TAutoString* object.

char*()

operator char*();
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.

const char far*()

operator const char far*();
Returns the object's string value in the form of a **const** C-style string.

Public data member

ClassInfo

static TAutoType ClassInfo;

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.

TAutoType struct

ocf/autodefs.h

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

GetType
short GetType();
Returns an integer that identifies a particular data type. The identifiers are defined in the *AutoDataType* **enum**.

See also AutoDataType enum

TAutoVal class

ocf/autodefs.h

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

operator =

```
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);
```

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 Automation data types, String class

Clear

void Clear();

Clears the value stored in the object, leaving it empty. This method cannot be called on the objects managed by *TAutoStack*.

See also TAutoStack class

Copy

void Copy(const TAutoVal& copy);

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.

double far*()

operator double far*();

Returns a pointer to a **double** value.

double()

operator double();

Returns the value in the object as a **double** value.

int far*()

operator int far*();

Returns a pointer to an **int**.

float far*()

operator float far*();

Returns a pointer to a floating-point value.

float()

operator float();

Returns the value in the object as a floating-point value.

GetDataType

int GetDataType();

Returns an integer identifying the type of value that was assigned to the union.

IDispatch&()

operator IDispatch&();

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*()

IDispatch*()

operator IDispatch*();

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&()

int()
operator int();
Returns the value in the object as an integer.

int far*()
operator int far*();
Returns a pointer to an **int**.

IsRef
bool IsRef();
Returns **true** if the value assigned to the union is a reference to a value.

IUnknown&()
operator IUnknown&();
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*()

IUnknown*()
operator IUnknown*();
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&()

long()
operator long();
Returns the value in the object as a **long** integer.

long far*()
operator long far*();
Returns a pointer to a **long** integer.

short()
operator short();
Returns the value in the object as a **short** integer.

short far*()
operator short far*();
Returns a pointer to a **short** integer.

string()
operator string();
Returns the value in the object as a C++ *string* object.

TAutoCurrency()
operator TAutoCurrency();
Returns the value in the object as a currency value.

See also TAutoCurrency class

TAutoCurrency far*()

operator TAutoCurrency far*();
Returns a pointer to a currency value.

See also TAutoCurrency

TAutoDate()

operator TAutoDate();
Returns the value in the object as a date value.

See also TAutoDate class

TAutoDate far*()

operator TAutoDate far*();
Returns a pointer to a date value.

See also TAutoDate class

TBool()

operator TBool();
Returns the value in the object as a Boolean value.

TBool far*()

operator TBool far*();
Returns a pointer to a Boolean value.

TUString*()

operator TUString*();
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*.

unsigned long()

operator unsigned long();
Returns the value in the object as an **unsigned long** integer.

unsigned long far*()

operator unsigned long far*();
Returns a pointer to a **long** integer. (*TAutoVal* does not distinguish **long** from **unsigned long**.)

See also TAutoStack class

TAutoVoid struct — ocf/autodefs.h

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*.

See also
TAutoVal struct

Public data member

ClassInfo

static TAutoType ClassInfo;
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

ocf/ocreg.h

typedef IUnknown* (*TComponentFactory)(IUnknown* outer, uint32 options, uint32 id = 0);
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).

See also TOcAppMode enum, TOcRegistrar class, TRegistrar class

TLocaleId type definition

ocf/autodefs.h

typedef unsigned long TLocaleId;
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, TLangId typedef

TOcApp class

ocf/ocapp.h

Base class

TUnknown

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 it is created for you by your *TOcRegistrar* object.

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.

See also

TOcModule::OnInit, TOcRegistrar class, TOleFactory<> class, TRegistrar class, TUnknown class

Type definitions

TOcMenuEnable

enum TOcMenuEnable

These enumeration values are flags that can be combined with the bitwise OR operator (|). A container passes them to the *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
EnablePaste	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.
EnablePasteLink	The Paste Link command adds to the open document a link to the object on the Clipboard.

Constant	Menu command enabled
EnableBrowseClipboard	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.
EnableBrowseLinks	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 REGFORMAT macro, TOcApp::EnableEditMenu

Public member functions

AddUserFormatName

void AddUserFormatName(char far* name, char far* resultName, char far* id = 0);

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 REGFORMAT macro

Browse

bool Browse(TOcInitInfo& initInfo);

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

BrowseClipboard

bool BrowseClipboard(TOcInitInfo& initInfo);

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

CanClose

bool CanClose();

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.

Clip

bool Clip(IBPart far* part, bool link, bool embed, bool delay = false);

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

Convert

bool Convert(TOcPart* ocPart, bool activate);

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

Drag

bool Drag(IBPart far* part, TOcDropAction inAction, TOcDropAction& outAction);

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

EnableEditMenu

uint EnableEditMenu(TOcMenuEnable enable, IBDataConsumer far* ocview);

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.)

TOleContainer and *TOleView* call *TOcApp::EnableEditMenu* in the command enabler functions for the Edit menu.

See also TOcApp::TOcMenuEnable enum, TOcView

EvActivate

void EvActivate(bool active);

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

EvResize

void EvResize();

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

EvSetFocus

bool EvSetFocus(bool set);

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

GetName

string GetName() const;

Returns a string object containing the application's name.

GetNameList

TOcNameList& GetNameList();

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

GetRegistrar()

```
TOcRegistrar& GetRegistrar();
```

Returns the application's registrar object. This is the same object passed into the *TOcApp* constructor.

See also TOcApp public constructor and destructor, TOcRegistrar class

IsOptionSet

```
bool IsOptionSet(uint32 option) const;
```

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.

See also TOcApp::SetOption, TOcAppMode enum

Paste

```
bool Paste(TOcInitInfo& initInfo);
```

Fills *initInfo* with information about the object on the Clipboard. Returns **true** if it succeeds in gathering information and **false** if it fails.

Create *initInfo* first. The *TOcInitInfo* constructor receives the view object where the new part will be inserted. Then call *Paste* to put information in *initInfo*. Finally, call *TOcView::Drop* to put the object in the view.

This function is called by *TOcView::Paste*.

See also TOcInitInfo, TOcView, TOcView::Drop, TOcView::Paste

RegisterClass

```
bool RegisterClass(const string& progid, BCID classId, bool multiUse);
```

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

RegisterClasses

```
void RegisterClasses(const TDocTemplate* tplHead = ::DocTemplateStaticHead);
```

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

ReleaseObject

virtual void ReleaseObject();

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

SetOption

void SetOption(uint32 bit, bool state);

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

SetupWindow

void SetupWindow(HWND frameWnd);

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.

TranslateAccel

bool TranslateAccel(MSG far* msg);

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.

UnregisterClass

bool UnregisterClass(const string& progid);

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

UnregisterClasses

```
void UnregisterClasses(const TDocTemplate* tplHead = ::DocTemplateStaticHead);
```

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

Protected constructor and destructor

Constructor

```
TOcApp(TOcRegistrar& registrar, uint32 options = ULONG_MAX, IUnknown* outer = 0,
  const TDocTemplate* tplHead = ::DocTemplateStaticHead);
```

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

```
~TOcApp();
```

The *TOcApp* destructor notifies OLE that the application is no longer available.

Usually the creation and destruction of an application's *TOcApp* object are managed by the *TOcRegistrar* object.

See also TComponentFactory typedef, TOcRegistrar class, TOcApp::ReleaseObject

Protected member functions

ForwardEvent

```
uint32 ForwardEvent(int eventId, const void far* param);
uint32 ForwardEvent(int eventId, uint32 param = 0);
```

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

ForwardEvent

uint32 ForwardEvent(int eventId, const void far* param);
uint32 ForwardEvent(int eventId, uint32 param = 0);
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 — ocf/ocreg.h

enum TOcAppMode
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.

Constant	What the Server Should Do
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

ocf/ocobject.h

enum TOcAspect

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

TOcDialogHelp enum

ocf/ocobject.h

enum TOcDialogHelp

The OC_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.

Constant	Dialog box	Purpose
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

EvOcAppDialogHelp event handler, OC_APPxxxx messages, TOleFrame::EvOcAppDialogHelp, WM_OCEVENT message

TOcDocument class

ocf/ocdoc.h

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 undocumented *TOcStorage* and *TOcStream* classes. These classes are thin wrappers around standard OLE interfaces. The implementation of *TOcDocument* makes use of both objects.

See also
TOcRemView class, TOcView class

Public constructors and destructor

Constructors
Form 1 `TOcDocument(TOcApp& app, const char far* fileName = 0);`
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 `TOcDocument(TOcApp& app, const char far* fileName, IStorage far* storageI);`
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 its internal, undocumented *TOcStorage* class. It is usually not necessary to manipulate the *IStorage* interface or the *TOcStorage* class directly in an ObjectComponents application.

Destructor
`~TOcDocument();`
Destroys the *TDocument* object.

Public member functions

Close
`void Close();`
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 class

GetActiveView
`TOcView* GetActiveView();`
Returns a pointer to the active view. *TOcPart* calls this method to coordinate changing focus among active parts.

See also TOcDocument::SetActiveView

GetName
`string GetName() const;`
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

GetParts

TOcPartCollection& GetParts();

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 class, TOcPartCollection class, TOcPartCollectionIter class

GetStorage

TOcStorage* GetStorage();

Returns the document file's root storage.

See also TOcDocument::SetStorage

LoadParts

bool LoadParts();

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

RenameParts

void RenameParts(IBRootLinkable far* BLDocumentI);

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

SaveParts

bool SaveParts(IStorage* storage = 0, bool sameAsLoaded = true);

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

SaveToFile

```
bool SaveToFile(const char far* newName);
```

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

SetActiveView

```
void SetActiveView(TOcView* view);
```

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

SetName

```
void SetName(const string& newName);
```

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

SetStorage

Form 1

```
void SetStorage(const char far* path);
```

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

```
void SetStorage(IStorage* storage);
```

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.)

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.

See also TOcDocument::GetStorage

TOcDragDrop struct

ocf/ocview.h

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 *lParams*. *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 *TOleView* and *TOleWindow*.

See also

OC_VIEWxxxx messages, TOleView::EvOcViewDrag, TOleView::EvOcViewDrop, TOleWindow::EvOcViewDrag, TOleWindow::EvOcViewDrop

Public data members

InitInfo

TOcInitInfo far* InitInfo;

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 class

Pos

TRect Pos;

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 class

Where

TPoint Where;

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 class

TOcDropAction enum

ocf/ocobject.h

enum TOcDropAction

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.

Constant	Meaning
daDropLink	Create a link to the object at the drop site.
daDropNone	No action occurred.

See also
TOcApp::Drag

TOcFormatList class

ocf/ocview.h

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.

See also
TOcFormat class, TOcFormatListIter class, TOcView class

Public constructor and destructor

Constructor

```
TOcFormatList();
```

Creates an empty list object. To insert items in the list, call the *Add* method.

Destructor

```
~TOcFormatList();
```

Deletes all the items in the list.

See also TOcFormatList::Add

Public member functions

Add

```
int Add(TOcFormat* format);
```

Inserts a new Clipboard format item in the list. Returns 0 for failure and 1 for success.

See also TOcFormatList::Clear

Clear

void Clear(int del = 1);

Removes all the items from the list. If *del* is 1, *Clear* also deletes all the *TOcFormat* objects.

See also TOcFormatList::Add, TOcFormatList::Detach

Count

virtual uint Count() const;

Returns the number of items in the list.

See also TOcFormatList::IsEmpty

Detach

int Detach(const TOcFormat* format, int del = 0);

Removes one format item from the list. If *del* is 1, then *Detach* also deletes the *TOcFormat* object.

See also TOcFormatList::Add, TOcFormatList::Clear

Find

unsigned Find(const TOcFormat* format) const;

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.

IsEmpty

int IsEmpty() const;

Returns 1 if the list object currently contains no *TOcFormat* items and 0 if the list is not empty.

See also TOcFormatList::Count

operator []

TOcFormat*& operator [(unsigned index)];

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.

TOcFormatListIter class — ocf/ocview.h

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.

See also

TOcFormat class, TOcFormatList class, TOcView class

Public constructor

Constructor

TOcFormatListIter(const TOcFormatList& collection)

Constructs an iterator to enumerate the Clipboard formats contained in collection.

Public member functions

Current

TOcFormat* Current() const;

Returns the format that the iterator currently points to.

operator ++

Form 1 TOcFormat* operator++();

Returns the current format and then advances the iterator to point to the next format (postincrement).

Form 2 TOcFormat* operator++(int);

Advances the iterator to point to the next format in the list and then returns that format (preincrement).

operator int()

operator int() const;

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.

Restart

Form 1 void Restart();

Resets the iterator to begin again with the first format in the list.

Form 2 void Restart(unsigned start, unsigned stop);

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

ocf/ocapp.h

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.

See also

TOcApp class, TOcNameList class

Public constructors and destructor

Constructors

Form 1 `TOcFormatName();`

Constructs an empty format name object.

Form 2 `TOcFormatName(char far* fmtName, char far* fmtResultName, char far* id = 0);`

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

`~TOcFormatName();`

Releases the object.

Public member functions

GetId

`const char far* GetId();`

Returns a pointer to the string that the system uses to designate the format.

GetName

`const char far* GetName();`

Returns a pointer to the name of the format.

GetResultName

`const char far* GetResultName();`

Returns the descriptive string that tells the user what pasting data of this format produces.

operator ==

`bool operator ==(const TOcFormatName& other);`

Returns **true** if *other* is the same object as **this**.

TOcInitHow enum

ocf/ocobject.h

enum TOcInitHow

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

ocf/ocobject.h

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, *TOleView* manages these details for you. If you are programming without ObjectWindows, you can find sample code for using *TOcInitInfo* objects in the *TOleView* methods that insert objects: look at the code for *CmEditInsertObject* and *CmEditPasteSpecial*. Look also at the code for *TOcView::Drop*.

See also

TOcPart Class, TOcView::Drop, TOleView::CmEditInsertObject, TOleView::CmEditPasteSpecial

Public data members

Container

IBContainer far* Container;

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 class

HIcon

HICON HIcon;

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.

How

```
TOcInitHow How;
```

Tells whether the object should be linked or embedded when it is added to the document.

See also TOcInitHow enum

Storage

```
IStorage far* Storage;
```

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.

Where

```
TOcInitWhere Where;
```

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

Data

```
IDataObject* Data;
```

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.

Path

```
LPCOLESTR Path;
```

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.

ClId

```
BCID ClId;
```

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. *ClId* 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

Handle

```
struct{
```

```
  HANDLE Data;
  uint DataFormat;
} Handle;
```

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.

Public constructors

TOcInitInfo

```
IBContainer far* Container;
```

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 class

TOcInitInfo

Form 1

```
TOcInitInfo(IBContainer far* container);
```

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

```
TOcInitInfo(TOcInitHow how, TOcInitWhere where, IBContainer far* container);
```

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.

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*.

Public member functions

ReleaseDataObject

```
uint32 ReleaseDataObject();
```

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

ocf/ocobject.h

```
enum TOcInitWhere
```

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

ocf/ocobject.h

enum TOcInvalidate

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::InvalidateTOc, TOleView::InvalidatePart, TOleWindow::InvalidatePart

TOcMenuDescr struct

ocf/ocapp.h

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*.

See also
OC_APPxxxx messages, OC_VIEWxxxx messages, TMenuDescr class

Public data members

HMenu
HMENU HMenu;
Holds a handle to the shared menu. The handle is valid only while the menu is constructed. Do not store it for later use.

Width
int Width[6];
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 *TMenuDescr* object. The numbers it holds control how the menu is merged.

See also TMenuDescr public constructors and destructors

TOcModule class

ocf/ocapp.h

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.

See also
TApplication class, TOcApp class

Public constructor and destructor

Constructor

TOcModule();

Builds a *TOcModule*. After creating a *TOcModule* object, you need to call *OcInit*.

Destructor

~TOcModule();

Releases the *TOcApp* object. An application that derives from *TOcModule* does not need to call the *TOcApp::ReleaseObject* method when it closes down. (Never call **delete** to destroy a *TOcApp* object, either.)

See Also TOcModule::OcInitTOcModuleOcInit

Public member functions

GetRegistrar

TRegistrar& GetRegistrar();

Returns the application's registrar object. Be sure to call *OcInit* first.

See also TOcModule::OcInit, TRegistrar class

IsOptionSet

bool IsOptionSet(uint32 option) const;

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*.

See also TOcAppMode class, TOcRegistrar class

OcInit

void OcInit(TOcRegistrar& registrar, uint32 options);

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 *OcInit* 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 class, TOcAppMode enum, TOcModule::IsOptionSet, TOcModule::IsOptionSet,, TOcRegistrar class

Public data members

OcApp

TOcApp* OcApp;

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 class, TOcModule::OcInit

OleMalloc

TOleAllocator OleMalloc;

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 class, TOleAllocator::Mem

TOcNameList class — ocf/ocapp.h

TOcApp uses this class internally to manage a collection of *TOcFormatName* objects. 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.)

See also

TOcApp class, TOcApp::AddUserFormatName, TOcFormatName class

Public constructor and destructor

Constructor

TOcNameList();

Constructs a name list containing no items. To insert names in the list, call *Add*.

Destructor

~TOcNameList();

Destroys the list and the objects in the list.

See also TOcNameList::Add

Public member functions

operator []

Form 1 `TOcFormatName*& operator[](unsigned index);`

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 `TOcFormatName* operator[](char far* id);`

Returns the format name object whose format ID string matches *id*. The return value is 0 if no match is found.

See also TOcFormatName

Add

`int Add(TOcFormatName* name);`

Inserts the object *name* into the list. Returns 1 for success and 0 for failure.

See also TOcNameList::Clear, TOcNameList::Detach

Clear

`void Clear(int del = 1);`

Empties the list. If *del* is 1, *Clear* also deletes each object in the list.

See also TOcNameList::Add, TOcNameList::Detach

Count

`virtual uint Count() const;`

Returns the number of items in the list.

See also TOcNameList::IsEmpty

Detach

`int Detach(const TOcFormatName* name, int del = 0);`

Removes the single object *name* from the list. If *del* is 1, *Detach* also deletes the object *name*.

See also TOcNameList::Clear, TOcNameList::Add

Find

`unsigned Find(const TOcFormatName* name) const;`

Searches the list and returns the position of *name*. If the *name* object is not in the list, *Find* returns UINT_MAX.

IsEmpty

`int IsEmpty() const;`

Returns 1 if the list currently contains no items and 0 if it contains at least one item.

See also TOcNameList::Count

TOcPart class

ocf/ocpart.h

Base class
TUnknown

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.

See also
TOcDocument class, TOcPartCollection class, TOcRemView class, TUnknown class

Public constructors

Constructors

Form 1 `TOcPart(TOcDocument& document, TOcInitInfo far& initInfo, TRect pos, int id = 0);`
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 `TOcPart(TOcDocument& document, const char far* name);`
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.

Both constructors expect to receive the container's own *TOcDocument* object. This represents the compound document where the new object will be placed.

See also TOcDocument class, TOcInitInfo class, TOcPart::Delete, TRect class

Public member functions

operator ==

`bool operator==(const TOcPart& other);`
Returns **true** if *other* is the same *TOcPart* as **this**. This operator is defined for the use of the *TOcPartCollection* class.

Activate

`bool Activate(bool activate);`
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 deactivates an in-place editing session.

Activate returns **true** if the server is able to execute the command.

See also TOcPart::IsActive, TOcPart::Open

Close

```
bool Close();
```

Disconnects the embedded object from its server. Returns **true** if the server closes successfully.

Delete

```
void Delete();
```

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

Detach

```
int Detach();
```

Separates a part from its document. Call *Detach* before cutting a part to the Clipboard, for example.

DoVerb

```
bool DoVerb(uint whichVerb);
```

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

Draw

```
bool Draw(HDC dc, const TRect& pos, const TRect& clip, TOcAspect aspect = asDefault);
```

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 class

EnumVerbs

```
bool EnumVerbs(const TOcVerb& verb);
```

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.

GetName

LPCOLESTR GetName();

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

GetNameLen

int GetNameLen();

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

GetPos

TPoint GetPos() const;

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 class

GetRect

TRect GetRect() const;

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 class

GetServerName

LPCOLESTR GetServerName(TOcPartName partName);

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

GetSize

TSize GetSize() const;

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 class

IsActive

bool IsActive() const;

Returns **true** if the part is currently active and **false** if it is not.

See also TOcPart::SetActive

IsLink

```
bool IsLink() const;
```

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*.

IsSelected

```
bool IsSelected() const;
```

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

IsVisible

Form 1

```
bool IsVisible() const;
```

Returns **true** if the part is currently visible and **false** if it is hidden.

Form 2

```
bool IsVisible(const TRect& logicalRect) const;
```

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

Load

```
bool Load();
```

Initializes a *TOcPart* object with information read from a storage.

See also TOcPart::Save

Open

```
bool Open(bool open);
```

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

Rename

```
void Rename();
```

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

Save

Form 1 `bool Save(bool sameAsLoaded = true);`

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 `bool Save(IStorage* storage, bool sameAsLoaded, bool remember);`

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

Select

`void Select(bool select);`

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

SetActive

`void SetActive();`

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

SetHost

`bool SetHost(IBContainer far* container);`

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.

SetPos

void SetPos(const TPoint& pos);

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 class

SetSize

void SetSize(const TSize& size);

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::SetSize, TOcPart::UpdateRect, TSize class

SetVisible

void SetVisible(bool visible);

Shows or hides the part, according to the value of *visible*.

See also TOcPart::IsVisible

Show

bool Show(bool show);

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

UpdateRect

void UpdateRect();

Sets the part to a new rectangle when its size or position changes. Called by *SetPos* and *SetRect*.

See also TOcPart::GetRect, TOcPart::SetPos, TOcPart::SetSize

Protected destructor

Destructor

~TOcPart();

Destroys the *TOcPart* object.

TOcPartCollection class

ocf/ocpart.h

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.

See also
TOcPart class, TOcPartCollectionIter class

Public constructor and destructor

Constructor
TOcPartCollection();
Creates an empty collection. Call *Add* to insert parts in the collection.

Destructor
~TOcPartCollection();
Releases all the servers that supply the linked or embedded objects.

Public member functions

Add
int Add(TOcPart* const& part);
Adds a new part to the collection. Returns 1 for success and 0 for failure.

See also TOcPart class

Clear
void Clear();
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.

Count
virtual unsigned Count() const;
Returns the number of parts currently in the collection.

Detach
int Detach(TOcPart* const& part, int del = 0);
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 class

Find
unsigned Find(TOcPart* const& part) const;
Searches for *part* and returns its position in the collection. If *part* is not in the collection, *Find* returns UINT_MAX.

See also TOcPart class

IsEmpty
int IsEmpty() const;
Returns **true** if the collection currently contains no objects and **false** if it does contain at least one object.

Locate

TOcPart* Locate(TPoint& point);

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 class

SelectAll

bool SelectAll(bool select = false);

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.

TOcPartCollectionIter class

ocf/ocpart.h

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.

See also

TOcPart class, TOcPartCollection class

Public constructor

Constructor

TOcPartCollectionIter(const TOcPartCollection& coll);

Constructs an iterator to enumerate the objects contained in the collection *coll*.

See also TOcPartCollection class

Public member functions

operator ++

Form 1 TOcPart* operator++(int);

Returns the current part and then advances the iterator to point to the next part (postincrement).

Form 2 TOcPart* operator++();
Advances the iterator to point to the next part in the list and then returns that part (preincrement).

Current
TOcPart* Current() const;
Returns the part that the iterator currently points to.

operator int()
operator int() const;
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.

Restart
Form 1 void Restart();
Resets the iterator to begin again with the first part in the document.

Form 2 void Restart(unsigned start, unsigned stop);
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 — ocf/ocobject.h

enum TOcPartName
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 — ocf/ocapp.h

Base class
TRegistrar
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*.

See also

clsid registration key, path registration key, progid registration key, TRegistrar class

Public constructor and destructor

Constructor

```
TOcRegistrar(TRegList& regInfo, TComponentFactory callback, string& cmdLine,
  HINSTANCE hInst = _hInstance);
```

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.

cmdLine holds the command line string that invoked the application.

hInst is the application's instance.

Destructor

```
~TOcRegistrar();
```

Destroys objects the registrar uses internally.

See also TComponentFactory typedef, TOleFactory<> class

Public member functions

BOleComponentCreate

```
HRESULT BOleComponentCreate(IUnknown far* far* retIface, IUnknown far* outer, BCID idClass);
```

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 result macros

CreateOcApp

```
void CreateOcApp(uint32 options, TOcApp*& ret);
```

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 TOcApp class, TOcAppMode enum, TOcModule::OcInit

GetAppDescriptor

```
TAppDescriptor& GetAppDescriptor();
```

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.

Protected member functions

CanUnload

```
bool CanUnload();
```

Returns **true** if the application is not currently serving any OLE clients and **false** otherwise.

GetFactory

```
void far* GetFactory(const GUID& clsid, const GUID far& iid);
```

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.

LoadBOle

void LoadBOle();

Loads and initializes the ObjectComponents support library (BOCOLE.DLL). *LoadBOle* throws a *TXObjComp* exception if it cannot find BOCOLE.DLL, or if the installed version is not compatible with the application's version of the library.

TOcRemView class

ocf/ocremvie.h

Base class

TOcView

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.

See also

TOcDocument class, TOcView class

Public constructor

Constructor

TOcRemView(TOcDocument& doc, TRegList* regList = 0, IUnknown* outer = 0);

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 BEGIN_REGISTRATION macro, TAutoObject class, TOcDocument class

Public member functions

Copy

```
virtual bool Copy();
```

Copies the object to the Clipboard. Returns **true** for success.

EvClose

```
virtual void EvClose();
```

The application's remote view window calls this function when it closes. *EvClose* disconnects the view from any parts displayed in it.

GetContainerTitle

```
virtual LPCOLESTR GetContainerTitle();
```

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).

GetInitialRect

```
void GetInitialRect();
```

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

Invalidate

```
void Invalidate(TOcInvalidate invalid);
```

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

IsOpenEditing

```
bool IsOpenEditing() const;
```

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.

Load

bool Load(IStorage* storageI);

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

Rename

virtual void Rename();

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.

Save

bool Save(IStorage* storageI);

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 — ocf/ocview.h

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.

The *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 *EvViewSavePart* and *EvViewLoadPart* methods in *TOleView*.

See also

OC_VIEWxxxx messages, TOleView::EvViewSavePart, TOleView::EvViewLoadPart

Public data members

Release

bool Release;

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.

Storagel

IStorage far* Storagel;

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

ocf/ocview.h

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: *EvViewGetScale*, *EvViewSetScale*, and *SetupDC*.

See also

OC_VIEWxxxx messages, TOleWindow::EvViewGetScale, TOleWindow::EvViewSetScale, TOleWindow::SetupDC, TSize class

Public constructors

Constructors

Form 1 TOcScaleFactor();

Initializes the site and part extents to 1 so the scaling factor is 100%.

Form 2 TOcScaleFactor(const RECT& siteRect, const TSize& partSize);

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 TOcScaleFactor(const BOleScaleFactor far& scaleFactor);
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.

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.

Destructor
~TOcScaleFactor();

See also TOcScaleFactor::PartSize, TOcScaleFactor::SiteSize, TSize class

Public data members

PartSize
TSize PartSize;
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 class

SiteSize
TSize SiteSize;
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 class

Public member functions

operator =
Form 1 TOcScaleFactor& operator =(const BOleScaleFactor far& scaleFactor);
Copies the values in a *BOleScaleFactor* structure. The BOCOLE support library uses this structure internally to carry scaling information.

Form 2 TOcScaleFactor& operator =(const TOcScaleFactor& scaleFactor);
Copies one *TOcScaleFactor* into another.

Both forms of the assignment operator copy the values from one scaling object into another.

See also OC_VIEWxxxx messages

GetScale

uint16 GetScale();

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

GetScaleFactor

void GetScaleFactor(BOleScaleFactor far& scaleFactor) const;

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.

IsZoomed

bool IsZoomed();

Returns **true** if the sizes stored for the part and the site do not match.

SetScale

void SetScale(uint16 percent);

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

ocf/ocobject.h

enum TOcScrollDir

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

EvOcViewScroll event handler, OC_VIEWxxxx messages, TOleView::EvOcViewScroll, TOleWindow::EvOcViewScroll, WM_OCEVENT message

TOcToolbarInfo struct

ocf/ocview.h

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* and *TOleView::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.

See also

TOleView::EvOcViewShowTools, TOleWindow::EvOcViewShowTools

Public data members

HBottomTB

HWND HBottomTB;

Holds a handle to the tool bar that the server wants to place at the bottom of the container's client area.

HFrame

HWND HFrame;

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

HLeftTB

HWND HLeftTB;

Holds a handle to the tool bar that the server wants to place at the left edge of the container's client area.

HRightTB

HWND HRightTB;

Holds a handle to the tool bar that the server wants to place at the right edge of the container's client area.

HTopTB

HWND HTopTB;

Holds a handle to the tool bar that the server wants to place at the top of the container's client area.

Show

bool Show;

Is **true** to ask that the server display its tool bar or **false** to request that the server hide the tool bar.

TOcVerb class

ocf/ocpart.h

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.

See also
TOcPart::EnumVerbs

Public constructor

Constructor
TOcVerb();
Creates an empty verb object.

Public data members

CanDirty
bool CanDirty;
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**.

TypeName
LPCOLESTR TypeName;
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

VerbIndex
uint VerbIndex;
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.

VerbName
LPCOLESTR VerbName;
Points to the name of the verb. This is the string that the container adds to its Edit menu.

TOcView class

ocf/ocview.h

Base class
TUnknown
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.

See also
TOcApp class, TOcDocument class, TOcPart class, TUnknown class

Public constructor

Constructor
TOcView(TOcDocument& doc, TRegList* regList = 0, IUnknown* outer=0);
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 BEGIN_REGISTRATION macro, TAutoObject class, TOcDocument class

Public member functions

ActivatePart

bool ActivatePart(TOcPart* part);

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

BrowseClipboard

bool BrowseClipboard(TOcInitInfo& initInfo);

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, TOcApp::BrowseLinks, TOcInitInfo class, TOcPart class

BrowseLinks

bool BrowseLinks();

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

Copy

bool Copy(TOcPart* part);

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.

EvActivate

void EvActivate(bool activate);

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

EvClose

virtual void EvClose();

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

EvResize

```
void EvResize();
```

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

EvSetFocus

```
bool EvSetFocus(bool set);
```

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

GetActivePart

```
TOcPart* GetActivePart();
```

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

GetOcDocument

```
TOcDocument& GetOcDocument();
```

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 class, TOcView::OcDocument

GetOrigin

```
TPoint GetOrigin() const;
```

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, TPoint class

GetWindowRect

```
TRect GetWindowRect() const;
```

Returns the client rectangle for the view window.

See also TOcView::GetOrigin, TRect class

InvalidatePart

```
void InvalidatePart(const TOcPart* part);
```

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.

Paste

```
bool Paste(bool linking = false);
```

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.

RegisterClipFormats

```
bool RegisterClipFormats(TRegList& regList);
```

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 macros to create *regList*. Also, the REGFORMAT macro places Clipboard format entries in the structure. To register custom Clipboard formats, be sure to call *TOcApp::AddUserFormatName* as well.

RegisterClipFormats is called automatically when the view is constructed.

See also BEGIN_REGISTRATION macro, REGFORMAT macro, TOcApp::AddUserFormatName, TOcView::FormatList

ReleaseObject

```
virtual void ReleaseObject();
```

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

Rename

```
virtual void Rename();
```

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.

ScrollWindow

```
void ScrollWindow(int dx, int dy);
```

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

SetLink

```
void SetLink(bool pasteLink);
```

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 REGFORMAT macro, TOcView::Paste

SetupWindow

void SetupWindow(HWND hWin);

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.

See also OC_VIEWxxxx messages, TOleWindow::SetupWindow, TOcView::Win

Protected destructor

Protected Destructor

~TOcView();

Destroys the view object.

Protected member functions

ForwardEvent

Form 1 uint32 ForwardEvent(int eventId, const void far* param);

Form 2 uint32 ForwardEvent(int eventId, uint32 param = 0);

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

Init

void Init(TRegList* regList);

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 macros and conventionally named *DocReg*).

Init makes this view the document's active view, connects with the BOCOLE support library, and registers supported Clipboard formats.

Shutdown

void Shutdown();

Called by the destructor of derived classes to release helper objects that the view holds internally.

See also TOcView public constructors and destructor

Protected data members

ActivePart

TOcPart* GetActivePart();

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

Extent

TSize Extent;

Holds the current width and height of the container window's client area. Both are measured in device units.

See also TOcView::GetWindowRect

FormatList

TOcFormatList FormatList;

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 class, TOcView::RegisterClipFormats

Link

int Link;

Used internally by the *Paste* method to adjust the priority of link source format.

OcApp

TOcApp& OcApp;

A view stores the application that owns it in this protected data member.

OcDocument

TOcDocument& OcDocument;

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

Origin

TPoint Origin;

Holds the coordinates of the point currently mapped to the upper left corner of the container window's client area.

See Also TOcView::GetOrigin

Win

HWND Win;

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

WinTitle

string WinTitle;

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.

TOcViewPaint struct — ocf/ocview.h

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* and *TOleView::EvOcViewPaint*.

See also

OC_VIEWxxxx messages, TOleView::EvOcViewPaint, TOleWindow::EvOcViewPaint

Public data members

Aspect

TOcAspect Aspect;

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

Clip

TRect* Clip;

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 class

DC

HDC DC;

Contains a handle to the device context where the repainting should occur.

Part
TOcPart* Part;
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 class

Pos
TRect* Pos;
Specifies the upper left corner of the server object that has become invalid and needs repainting.

See also TRect class

TOleAllocator class

ocf/oleutil.h

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 registrar 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.

See also
TOcRegistrar class, TRegistrar class

Public constructors and destructor

Constructor

Form 1 TOleAllocator(IMalloc* mem = 0);
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 TOleAllocator();
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

~TOleAllocator();

Releases the memory allocator (either the default allocator or a custom allocator) and uninitializes the OLE system.

See also TOcRegistrar Class, TRegistrar Class

Public member functions

Alloc

void far* Alloc(unsigned long size);

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_

Free

void Free(void far* block);

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_

Public data member

Mem

IMalloc* Mem;

Points to the active memory allocator object. Unless you have registered a custom memory allocator, *Mem* points to OLE's default allocator.

TRegistrar class

ocf/ocreg.h

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, remaining invisible if invoked for embedding.
- Call *Run* to enter the program's message loop.

See also
TOcRegistrar class

Public constructor and destructor

Constructor
TRegistrar(TRegList& regInfo, TComponentFactory callback, string& cmdLine, HINSTANCE hInst);
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.

Destructor
virtual ~TRegistrar();
Deletes objects the registrar maintains internally.

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.

See also string class, TComponentFactory typedef, TOleFactory class, TRegistrar::IsOptionSet

Public member functions

CanUnload
virtual bool CanUnload();
Returns **true** if the application is not currently serving any OLE clients and **false** otherwise.

CreateAutoApp
TUnknown* CreateAutoApp(TObjectDescriptor app, uint32 options, IUnknown* outer = 0);
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<> class, TComponentFactory typedef, TRegistrar::CreateAutoObject

CreateAutoObject

Form 1 `TUnknown* CreateAutoObject(TObjectDescriptor obj, TServedObject& app);`
app is the automated OLE application object.

obj is the automated C++ object.

Form 2 `TUnknown* CreateAutoObject(const void* obj, const typeinfo& objInfo, const void* app, const typeinfo& appInfo);`
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*.

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.

See also TComponentFactory typedef, TRegistrar::CreateAutoApp, typeid, typeinfo class

GetFactory

`virtual void far* GetFactory(const GUID& clsid, const GUID far& iid);`
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.

GetOptions

`uint32 GetOptions() const;`
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

IsOptionSet

`bool IsOptionSet(uint32 option) const;`
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.

See also TOcApp::IsOptionSet, TOcAppMode enum, TRegistrar::GetOptions, TRegistrar::ProcessCmdLine, TRegistrar::SetOption

ProcessCmdLine

void ProcessCmdLine(string& cmdLine);

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 string class, TRegistrar::IsOptionSet

ReleaseAutoApp

void ReleaseAutoApp(TObjectDescriptor app);

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.

RegisterAppClass

void RegisterAppClass();

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

Run

virtual int Run();

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 run the application. The application enters its message loop.

- Call the factory function to shut down the application.
- Ensure that the application's *TOcApp* connector object is properly released.

See also TOcApp, TRegistrar::RegisterApp, TRegistrar::Shutdown

SetOption

void SetOption(uint32 bit, bool state);

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

Shutdown

virtual void Shutdown(IUnknown* releasedObj, uint32 options);

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

UnregisterAppClass

void UnregisterAppClass();

Announces that the application is no longer available for OLE interactions.

See also TRegistrar::RegisterAppClass

Protected data member

AppDesc

TAppDescriptor& AppDesc;

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.

Protected constructor

Constructor

TRegistrar(TAppDescriptor& appDesc);

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.

TUnknown class

ocf/oleutil.h

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.

Aggregation is established in the *TComponentFactory* callback function. Each component receives the *IUnknown* pointer to its outer object and returns its own *IUnknown* pointer to be placed in the chain of secondary objects.

See also

TComponentFactory typedef

Public member functions

Aggregate

```
IUnknown& Aggregate(TUnknown& inner);
```

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 should use the return value as its *Outer* pointer, too. To aggregate **this** under an object that is not a *TUnknown*, call *SetOuter* instead.

See also TUnknown::SetOuter, TUnknown::Outer

GetOuter

```
IUnknown* GetOuter();
```

Returns a pointer to the object's outer *IUnknown* interface, the one that belongs to the primary object in a group of aggregated objects.

See also TUnknown::SetOuter, TUnknown::Outer

GetRefCount

unsigned long GetRefCount();

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. The destructor prevents the object from being destroyed if the reference count is not 0.

operator IUnknown&()

operator IUnknown&();

Returns a reference to the object's outer *IUnknown* interface. Does not increment the object's reference count.

operator IUnknown*()

operator IUnknown*();

Returns a pointer to the object's outer *IUnknown* interface. Increments the object's reference count first.

See also TUnknown::operator IUnknown&()

SetOuter

IUnknown* SetOuter(IUnknown* outer);

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::operator IUnknown*()

Protected constructor and destructor

Constructor

TUnknown();

Creates a *TUnknown* object with an initial reference count of 0. Initially the object is not aggregated with any other object.

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.

Destructor

virtual ~TUnknown();

Deletes the object.

See also TUnknown::Aggregate

Protected member functions

QueryObject

virtual HRESULT QueryObject(const GUID far& iid, void far* far* pif);

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* should override this function.

For examples of override functions, look at the source code for classes such as *TOcApp* and *TOcView*.

ThisUnknown

IUnknown& ThisUnknown();

Returns a reference to the *IUnknown* interface for **this**, not to the outer or inner aggregated objects.

Protected data member

Outer

IUnknown* Outer;

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

ocf/autodefs.h

Base class

TXBase

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.

See also

TXBase class (OWL.HLP), TXObjComp class, TXOle class, TXRegistry class

Public constructor

Constructor

TXAuto(TXAuto::TError err);

Constructs an exception object to describe the problem indicated by *err*.

See also TXAuto::TError enum

Public data member

ErrorCode

TError ErrorCode;

Holds the code that identifies the problem this object was constructed to describe.

See also TXAuto::TError enum

Type definition

TError

enum TError

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.

TXObjComp class

ocf/ocdefs.h

Base class

TXBase

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.

See also

TXAuto class, TXBase class, TXOle class, TXRegistry class

Public constructor

Constructor

TXObjComp(TXObjComp::TError err, const char* msg = 0);

Constructs an exception object to describe the problem indicated by *err*. Associates the optional *msg* string with the error.

See also TXObjComp::TError

Public member function

ErrorCode

TError ErrorCode;

Holds the error code that identifies the problem this object was constructed to describe.

See also TXObjComp::TError enum

Type definition

TError

enum TError

The values of the enumeration identify possible errors that can occur inside ObjectComponents.

Constant	Meaning
Application Errors	
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	
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	
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.

TXOle class

ocf/oleutil.h

Base class

TXBase

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.

See also
TXAuto class, TXObjComp class, TXRegistry class

Public constructors and destructor

Constructors

Form 1 `TXOle(const char far* msg, HRESULT stat);`
Creates an OLE exception object. *msg* points to an error message and *stat* holds the return value from an OLE API call.

Form 2 `TXOle(const TXOle& copy);`
Constructs a new OLE exception object by copying the one passed as *copy*.

Usually you do not need to construct an OLE exception object directly. Call *Check* instead.

Destructor

`~TXOle();`
Destroys the *TXOle* object.

See also TXOle::Check

Public member functions

Check

Form 1 `static void Check(HRESULT stat, const char far* msg);`
If *stat* indicates an error, Form 1 throws a *TXOle* exception containing the *msg* error string.

Form 2 `static void Check(HRESULT stat);`
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.

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.

Public data member

Stat

long Stat;

Stat ("status") holds the result code returned from an OLE API.

TXRegistry class

ocf/ocdefs.h

Base class

TXBase

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.

See also

TXAuto class, TXObjComp class, TXOle class

Public constructors

Constructors

Form 1 TXRegistry(const char* msg, const char* key);

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 TXRegistry(const TXRegistry& copy);

The copy constructor constructs a new registry exception object by copying the one passed as *copy*.

Usually you do not need to construct a registry exception directly. Call *Check* instead.

See also TXRegistry::Check

Public member functions

Check

static void Check(long stat, const char* key);

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*."

Key

const char* Key;

Points to the name of the registration key that ObjectComponents was processing when the exception occurred.

typehelp registration key

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, REGDATA macro, typelib registration key

usage registration key

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

verb*n* registration keys

A string naming an action the server can perform with its objects. Containers add the active object's 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

REGDATA macro, verb*n*opt registration keys

verb*n*opt registration keys

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 constants, ocrxxxx verb attribute constants, REGVERBOPT macro, verb*n* registration keys

version registration key

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

WM_OCEVENT message

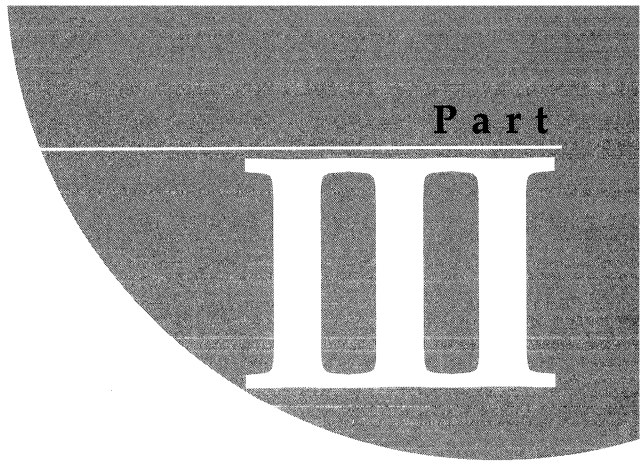
ocf/ocapp.h

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

OC_APPxxxx messages, OC_VIEWxxxx messages



ObjectSupport reference

Chapter

7

Overview of ObjectSupport

This chapter provides an overview of the ObjectSupport classes, libraries, and header files, which provide various services that help you design your ObjectWindows application. These classes include the following groups:

- Mathematical classes such as *TPoint*, *TSize*, and *TRect* that define screen coordinates and properties of rectangles.
- Document template classes that make it easier to design Doc/View applications.
- *TLocaleString*, which localizes OLE registration information required for containers and servers.

The geometric classes support various operations that you might want to perform on points and rectangles. *TPoint* encapsulates a two-dimensional point that represents a screen position. You can use *TPoint* to compare, assign, and manipulate points. *TSize* encapsulates a two-dimensional quantity that represents the displacement of an area or the height and width of a rectangle. You can use *TSize* to compare, assign, and manipulate sizes. *TRect* encapsulates the properties of rectangles with sides parallel to the x- and y-axes. You can use *TRect* to perform a variety of rectangle tests and manipulations, such as inflating, normalizing, and changing the offset dimensions of a rectangle. A parameterized class, *TPointer* holds a pointer to its parameterized type. You can assign a pointer to a *TPointer* object and easily remove the object by assigning 0 to the pointer.

Other ObjectSupport classes such as *TDropInfo* and *TDocTemplate* provide functions that let you manipulate files and documents. *TDropInfo*, which supports file drag and drop operations, makes it easy to determine the number of files dropped, the names of the files, and where they were dropped. The templetized classes, *TDocTemplate* and *TDocTemplateT*, support creating documents and views. You can use these classes to create a document template with a specified file description, file filter pattern, and default file extension.

The ObjectSupport library also includes classes and macros designed to simplify the process of localizing strings and building registration tables. The localization class,

TLocaleString, provides support for ObjectWindows' Doc/View as well as ObjectComponents' OLE-enabled applications. A struct defining a localizable substitute for **char*** strings, *TLocaleString* contains functions that translate and compare strings in a given language. ObjectWindows' registration macros simplify the process of building a registration table for either an automation server or a non-automated application or document.

The ObjectSupport library contains one exception class, *TXBase*, which is the base class for both ObjectWindows' and ObjectComponents' exception-handling classes. Exception classes derived from *TXBase* are designed to handle specific error conditions, such as an out-of-memory error or an attempt to create an invalid window. To handle an exception, you will want to derive a class from any one of the ObjectWindows' classes that describe this particular exception.

The following table lists the files included in the ObjectSupport Library (..\OSL directory).

Table 7.1 Summary of the ObjectSupport library files

File name	Description
geometry.h	Contains descriptions of mathematical classes such as TPoint, TRect, TSize, TResId, TDropInfo, TProcInstance, and TPointer.
doctpl.h	Contains definitions of files TDocTemplate, and TDocTemplateT<D,V>.
defs.h	Contains common definitions, includes windows.h definitions, and deals with BOOL data types.
except.h	Defines class TXBase, the base exception-handling class for ObjectWindows and ObjectComponents classes.
locale.h	Defines TLocaleString class as well as registration macros.

Chapter

8

ObjectSupport library reference

Registration macros

The following macros, defined in locale.h, take care of performing various OLE-related registration procedures. These macros simplify the process of building a registration table (a specific kind of lookup table) for an automation server and for a non-automated application or document associated with either a container or a server. A collection of vital statistics about an object, the registration table provides an external description associated with an object. Some of the information goes into the system registry and is used by OLE. Some of the information is displayed in the File | Open dialog box when the user selects the Insert Object selection.

The registration macros build a *TRegList* structure containing entries of type *TRegItem*, a struct which is defined as follows:

```
struct TRegItem {
  char*  Key;              //Item name
  TLocaleString Value;    //String value for the item
};
```

Using these macros saves you the trouble of having to build the *TRegItem* and *TRegList* structures directly.

Although both servers and containers use the same macros, they pass different kinds of information to the registration structures. Depending on the amount of information you want associated with your application and whether you want to set up a structure with document or application information, use one or more of these registration macros.

Registration macros

Macro	Meaning
BEGIN_REGISTRATION	Begins a registration macro table.
END_REGISTRATION	Ends a registration macro table.
REGDATA	Registers information about the application, for example, class ID, description, document filter, and debugger.
REGDOCFLAG	Registers a series of document flags. Required for a document registration table.
REGFORMAT	Registers a data format.
REGICON	Registers an icon.
REGITEM	Registers a customized format.
REGSTATUS	Registers an aspect status.
REGVERBOPT	Registers an option for a verb.
REGISTRATION_FORMAT_BUFFER	Sets the size of the buffer space needed for expansion.

REGDATA, which is the main macro used in the registration table, passes string data in its arguments. The REGFORMAT, REGDOCFLAG, REGICON, REGSTATUS, and REGVERBOPT macros format numeric values as string values.

See the *ObjectWindows Programmer's Guide* for a list of which item names (referred to as keys) are required in the application and document registration tables. For information about how to use these macros in your OLE-enabled applications, see the sections on "Registering a linking and embedding server" and "Registering the container" in the *ObjectWindows Programmer's Guide*. The sample applications, REGTEST.CPP and STEP15DV.CPP, on your distribution disk, provide examples of registration tables designed for different kinds of applications and documents.

See also END_REGISTRATION Macro, BEGIN_REGISTRATION Macro, TLocaleString

BEGIN_REGISTRATION macro

locale.h

BEGIN_REGISTRATION(regname)

Indicates the beginning of a registration macro table. The macro takes one argument (*regname*), which is the name of the structure to be built. Within the registration table macro, there are several macros that build the registration structure. Depending on the type of application or document, different macros are used. The following example from STEP15.CPP, registers the drawing pad as a server application and builds an *AppReg* structure:

```
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,       "{5E4BD320-8ABC-101B-A23B-CE4E85D07ED2}")
  REGDATA(description,"OWL Drawing Pad Server")
END_REGISTRATION
```

See also END_REGISTRATION macro, REGDATA macro

END_REGISTRATION macro

locale.h

END_REGISTRATION

END_REGISTRATION Indicates the end of a registration macro table. You can insert the registration macros within the BEGIN_REGISTRATION and END_REGISTRATION macros to build a registration structure.

See also BEGIN_REGISTRATION macro

REGDATA macro

locale.h

REGDATA(var,val)

The main registration macro, REGDATA registers information about an application or a document. The macro always takes an item name (for example, *clsid)* and a corresponding string value (for example, 5E4BD320–8ABC–101B–A23B–CE4E85D07ED2). The following example from STEP15.CPP on your distribution disk, passes the class ID and description to build an *AppReg* application registration structure:

```
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,       "{5E4BD320-8ABC-101B-A23B-CE4E85D07ED2}")
  REGDATA(description,"OWL Drawing Pad Server")
END_REGISTRATION
```

For an automation server, the registration structure includes the following REGDATA macros:

```
BEGIN_REGISTRATION(myappreg)
 REGDATA(clsid,       "{01234567-1234-5678-1122-334455667788}")
 REGDATA(progid,      "MySample")
 REGDATA(description,"My Sample 1.0 Application")
 REGDATA(cmdline,     "/automation")
 REGDATA(version,     "1.2")
END_REGISTRATION
```

Each automatable object requires a program ID, a description, and a command-line argument, which is placed on the server's command line. As usual, only one class ID is defined for the application. See the *ObjectWindows Programmer's Guide* for detailed information about how to register an automation server.

For each application registration structure, you may have one or more document registration structures. The following example uses the data from several REGDATA macros to build a *DocReg* registration structure.

```
BEGIN_REGISTRATION(mydocreg)
 REGDATA(description,"My Sample 1.0 Document")
 REGDATA(extension,  "myd") //Do not use a period before the extension.
 REGDATA(directory,  "C:\temp")
 REGDATA(docfilter,  "*.drw;*.drx")
 ...//Insert additional macros here.
 REGDATA(debugger,   "tdw -t") // Sets debugger option
 REGDATA(progid,     "MyDocument") //For servers only
 REGDATA(menuname,   "My Document") //For servers only
```

```
REGDATA(insertable,0) // For servers only
REGDATA(usage,    ocrMultipleUse) // For servers only
REGDATA(verb0,      "&Edit") //For servers only
REGDATA(verb1,      "&Open") //For servers only
REGDATA(verb2,      "&Play") //For servers only
...//Insert additional macros here.
END_REGISTRATION
```

See also BEGIN_REGISTRATION macro, END_REGISTRATION macro

REGITEM macro

locale.h

REGITEM(key,val)

The REGITEM macro lets you write customized entries for the system registry. The following example, from REGTEST.CPP, registers conversion formats.

```
REGITEM("CLSID\\<clsid>\\Conversion\\Readable\\Main", "FormatX,FormatY")
```

The first string is the registry key which has one parameter (*<clsid>*) When the registry information is generated, an actual value is substituted for the template parameters.

See also BEGIN_REGISTRATION macro, END_REGISTRATION macro

REGFORMAT macro

locale.h

REGFORMAT(i,f,a,t,d)

The REGFORMAT macro indicates the data formats the server or container can support and has the following parameters:

i Order of priority for the designated data format with 0 being the highest fidelity rendered.

f The data format, for example, *ocrText*, *ocrTiff*, *ocrDib*, and so on.

a The format used to present the data.

t Medium to use to transfer the data, for example, *ocrMfPict* (METAFILEPICT structure), *ocrGDI* (GDI object such as a bitmap), *ocrIStream* (Stream object in a compound file) and so on.

d Whether or not data is provided as well as received in the designated format. The accepted values are *ocrGet* (imports data in the specified format), *ocrSet* (exports data in the specified format), or *ocrGetSet* (both exports and imports data in the specified format).

For example, STEP15DV.CPP registers the following clipboard formats:

```
REGFORMAT(1, ocrMetafilePict, ocrcontent, ocrMfPict, ocfSet)
```

and generates the string, "format1, 3, 1, 1056,1." Although you could enter the string of numbers, it is much easier to use the enumerated values. See the *ocrxxxx Clipboard* constants for a description of the accepted data formats.

To build a registration structure, use REGFORMAT within the BEGIN_REGISTRATION and END_REGISTRATION macros in a registration macro table. Any formats registered using REGFORMAT are also registered automatically on

the Windows Clipboard. You can register your own formats by inserting a string indicating your own format. For example,

```
REGFORMAT(2, "ANewFormat", ocrContent, ocrIStorage, ocrGetSet)
```

To provide names for your own formats, call *TOleFrame::AddUserFormatName*. This function associates a clipboard data format with the description of the data format as it appears to users in the Help text of the Paste Special dialog box.

See also BEGIN_REGISTRATION macro, ocrxxxx Clipboard constants, ocrxxxx medium constants, TOleFrame::AddUserFormatName

REGSTATUS macro

locale.h

REGSTATUS(a,f)

The REGSTATUS macro indicates the way in which the view (referred to as the "aspect") of an object behaves and has the following arguments:

a The content of the object.

f One of the *ocrxxxx Object Status* enum values indicating the status of the object, for example, *ocrOnlyIconic*, *ocrActivateWhenVisible*, and so on.

The object can be defined as having many different aspects of behavior. For example, an object registered as *ocrActivateWhenVisible* is active whenever it is visible.

Servers that support linked and embedded objects use this macro to register the behavior an object exhibits when it is viewed. This behavior is referred to as the aspect status or simply aspect of the object.

The sample program, REGTEST.CPP, on your distribution disk includes the following REGSTATUS macros:

```
REGSTATUS(all, ocrNoSpecialRendering)
REGSTATUS(icon, ocrOnlyIconic)
```

The first macro registers flags for all aspects of the object while the second macro registers flags for the iconic aspect of the object. (The icon used in the REGSTATUS macro must have been defined and registered.)

See also BEGIN_REGISTRATION macro, REGICON macro, ocrxxxx object status constants

REGVERBOPT macro

locale.h

REGVERBOPT(v,mf,sf)

Registers the actions a server can perform on its objects. The arguments control how the verbs appear on the container's menu. The macro has the following arguments:

v The verb key, for example, *verb1* or *verb2*.

mf A value that describes how a server's verbs appear on the container's menu. This value must be one of the *ocrxxxx Verb Menu* constants, for example, *ocrGrayed*, which makes the verb appear gray on the menu and disables the verb.

sf A value that tells the container how to use the verb. This value must be one of the *ocrxxxx Verb Attribute* constants, for example, *ocrNeverDirties*, which indicates that the verb never modifies the object. These options can be ORed together.

The sample program, REGTEST.CPP, on your distribution disk includes the following REGVERBOPT macro:

```
REGVERBOPT(verb1, ocrGrayed, ocrOnContainerMenu | ocrNeverDirties)
```

These verb options are optional and are only valid if the verb is registered in the document registration table for the server application. To register the verb, use

```
REGDATA(verb1, "&Open")
```

See also BEGIN_REGISTRATION macro, ocrxxxx verb attribute constants, ocrxxxx verb menu constants

REGICON macro

locale.h

REGICON(i)

Registers an icon so that the object is displayed as an icon. The sample program, REGTEST.CPP, on your distribution disk, includes the following REGICON macro:

```
REGICON(1)
```

The macro takes one argument, the index of the default icon to use. This argument indicates which icon is to be retrieved from the resource file when the document is displayed as an icon.

See also BEGIN_REGISTRATION macro

REGDOCFLAGS macro

locale.h

REGDOCFLAGS(i)

Indicates options for the document and defines the characteristics of document templates. The REGDOCFLAGS arguments tell the document manager how to display and manage the documents and views. Although, for backward compatibility, you can still pass this information to the document template using the separate parameters in the constructor, newer programs should use the REGFORMAT, REGDOCFLAGS, and REGDATA macros to create a document template object.

The sample program, REGTEST.CPP, on your distribution disk, includes the following REGDOCFLAGS macro declaration within the document registration structure:

```
BEGIN_REGISTRATION(mytplreg)
 REGDATA(description,"My Sample Draw View")
 REGDATA(filter,     "*.drw;*.drx")
 REGDATA(defaultext, "dvw")
 REGDATA(directory,  0)
 REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt)
END_REGISTRATION
```

The arguments to REGDOCFLAGS define the document's characteristics. In the case of *dtAutoDelete*, for example, the document is deleted when the last view is deleted.

Certain documents must be registered with different options. For example, to register a Doc/View application used with a *TOleDocument* object, you must specify the *dtRegisterExt* and *dtAutoOpen* flags. If this is a Doc/View application, and if the document template is not hidden, you must register the description, filter, and extension.

See also BEGIN_REGISTRATION macro, REGFORMAT macro, REGDATA macro, dt document template constants

REGISTRATION_FORMAT_BUFFER macro — locale.h

REGISTRATION_FORMAT_BUFFER(n);
Allocates space in memory (*n*) for the expansion of the values passed in the registration macros that control the formatting of a document or application. Generally, allow 10 bytes for each value passed in the REGFORMAT macro in addition to the space required for strings passed in the REGSTATUS, REGVERBOPT, REGICON, or REGFORMAT macros. For example, the sample program, STEP15.CPP uses this macro to declare 100 bytes of space:

```
REGISTRATION_FORMAT_BUFFER(100);
```

See also REGFORMAT macro, REGSTATUS macro

TDocTemplate class — doctpl.h

TDocTemplate is an abstract base class that contains document template functionality. This document template class defines several functions that make it easier for you to use documents and their corresponding views. *TDocTemplate* classes create documents and views from resources and handle document naming and browsing. The document manager maintains a list of the current template objects. Each document type requires a separate document template.

Public member functions

ClearFlag
void ClearFlag(long flag);
Clears a document template constant.

See also dt document template constants

Clone

virtual TDocTemplate* Clone(TModule* module, TDocTemplate*& phead=DocTemplateStaticHead)=0;

Makes a copy of a document template.

ConstructDoc

virtual TDocument* ConstructDoc(TDocument* parent = 0) = 0;

A pure virtual function that must be defined in a derived class, *ConstructDoc* creates a document specified by the document template class. Use this function in place of *CreateDoc*.

See also TDocManager::CreateDoc

ConstructView

virtual TView* ConstructView(TDocument& doc = 0) = 0;

A pure virtual function that must be defined in a derived class, *ConstructView* creates the view specified by the document template class.

See also TDocManager::CreateView

CreateDoc

virtual TDocument* CreateDoc(const char far* path, long flags = 0)= 0;

An obsolete pure virtual function that must be defined in a derived class, *CreateDoc* creates a document based on the directory path (*path*) and the specified template and *flags* value. If the *path* is 0 and the new flag (*dtNewDoc*) is not set, the dialog box is displayed. This function is obsolete: use *ConstructDoc* instead.

See also TDocManager::CreateAnyDoc, TDocTemplate::ConstructDoc

CreateView

virtual TView* CreateView(TDocument& doc, long flags) = 0;

A pure virtual function that must be defined in a derived class, *CreateView* creates the view specified by the document template class. This function is obsolete: use *ConstructView* instead.

See also TDocManager::CreateAnyView, TDocTemplate::ConstructView

GetDefaultExt

const char far* GetDefaultExt() const;

Gets the default extension to use if the user has entered the name of a file without any extension. If there is no default extension, *GetDefaultExt* contains 0.

GetDescription

const char far* GetDescription() const;

Gets the template description to put in the file-selection list box or the File | New menu-selection list box.

GetDirectory

const char far* GetDirectory() const;

Gets the directory path to use when searching for matching files. This will get updated if a file is selected and the *dtUpdateDir* flag is set.

See also

GetDocManager
TDocManager* GetDocManager() const;
Points to the document manager.

GetFileFilter
const char far* GetFileFilter() const;
Gets any valid document matching pattern to use when searching for files.

GetFlags
long GetFlags() const;
Gets the document template constants, which indicate how the document is created and opened.

See also dt xxxx document template constants

GetModule
TModule*& GetModule();
Returns a module pointer.

GetNextTemplate
TDocTemplate* GetNextTemplate() const;
Gets the next template in the list of templates.

GetRegList
TRegList& GetRegList() const;
Gets the program's registration table, which contains the program's current program ID, class ID, executable path, as well as other attributes used to construct a *TDocTemplate* object. See the entry for Registration macros in this manual for information about how the registration macros generate registration information.

GetViewName
virtual const char far* GetViewName() = 0;
A pure virtual function that must be defined in a derived class, *GetViewName* gets the name of the view associated with the template.

InitDoc
TDocument* InitDoc(TDocument* doc, const char far* path, long flags);
InitDoc is called only from the subclass so that *CreateDoc* can continue its document processing.

See also TDocTemplate::CreateDoc

InitView
TView* InitView(TView* view);
Called only from the subclass to continue *CreateView* processing.

See also TDocTemplate::CreateView

IsFlagSet
bool IsFlagSet(long flag);
Returns nonzero if the document template flag is set.

See also dt xxxx document template constants

IsMyKindOfDoc

virtual TDocument* IsMyKindOfDoc(TDocument& doc)=0;

A pure virtual function that must be defined in a derived class, *IsMyKindOfDoc* tests if the template belongs to the same class as the document or to a derived class.

See also TDocTemplateT::IsMyKindOfDoc

IsMyKindOfView

virtual TView* IsMyKindOfView(TView& view) = 0;

A pure virtual function that must be defined in a derived class, *IsMyKindofView* tests if the template belongs to the same class as the view or to a derived class.

IsStatic

bool IsStatic();

Returns **true** if the template is statically constructed.

IsVisible

bool IsVisible();

Indicates whether the document can be displayed in the file selection dialog box. A document is visible if *dtHidden* isn't set and *Description* isn't 0.

SelectSave

bool SelectSave(TDocument& doc);

Prompts the user to select a file name for the document. Filters out read-only files.

SetDefaultExt

void SetDefaultExt(const char far*);

Sets the default extension to use if the user has entered the name of a file without any extension. If there is no default extension, *SetDefaultExt* contains 0.

SetDirectory

void SetDirectory(const char far*);
void SetDirectory(const char far*, int len);

Sets the directory path to use when searching for matching files. This will get updated if a file is selected and the *dtUpdateDir* flag is set.

See also TDocTemplate::GetDirectory

SetDocManager

void SetDocManager(TDocManager* dm);

Sets the current document manager to the argument *dm*.

SetFileFilter

void SetFileFilter(const char far*);

Sets the valid document matching pattern to use when searching for files.

SetFlag

void SetFlag(long flag);

Sets the document template constants, which indicate how the document is created and opened.

See also dtxxxx document template constants

SetModule

TModule*& SetModule();
Sets a module pointer.

Protected constructor and destructor

Constructor

TDocTemplate(TRegList& regList, TModule*& module, TDocTemplate*& phead):
Uses the information in the registration table (*regList*) to construct a *TDocTemplate* with the specified file description, file filter pattern, search path for viewing the directory, default file extension, and flags representing the view and creation options from the registration list. Then, adds this template to the document manager's template list. If the document manager is not yet constructed, adds the template to a static list, which the document manager will later add to its template list.

The argument, *module*, specifies the *TModule* of the caller. *phead* specifies the template head for the caller's module. See the Registration macros entry in this manual for information about the registration macros that generate a *TRegList*, which contains the attributes used to create a *TDocTemplate* object.

Destructor

~TDocTemplate();
Destroys a *TDocTemplate* object and frees the data members (*FileFilter, Description, Directory, and DefaultExt*). The Destructor is called only when no views or documents are associated with the template. Instead of calling this Destructor directly, use the *Delete* member function.

See also dtxxxx document template constants

TDocTemplateT<D,V> class

doctpl.h

To register the associated document and view classes, a parameterized subclass, *TDocTemplateT<D,V>*, is used to construct a particular document and view, where *D* represents the document class and *V* represents the view class. The parameterized template classes are created using a macro, which also generates the associated streamable support. The document and view classes are provided through the use of a parameterized subclass. The template class name is used as a **typedef** for the parameterized class. For example,

```
DEFINE_DOC_TEMPLATE_CLASS(TFileDocument, TEditView, MyEditFile)
```

You can instantiate a document template using either a static member or an explicit construction. For example,

```
MyEditFile et1("Edit text files",
      "*.txt","D:\\doc",".TXT"",dtNoAutoView);
new MyEditFile(.....)
```

When a document template is created, the document manager (*TDocManager*) registers the template. When the document template's delete function is called to delete the template, it is no longer visible to the user. However, it remains in memory as long as any documents still use it.

Public constructors

Constructors

Form 1 `TDocTemplateT(const char far* filt, const char far* desc, const char far* dir, const char far* ext, long flags = 0, TModule*& module = ::Module, TDocTemplate*& phead = DocTemplateStaticHead);`

Constructs a *TDocTemplateT* with the specified file description (*desc*), file filter pattern (*filt*), search path for viewing the directory (*dir*), default file extension (*ext*), and flags representing the view and creation options (*flags*). *module*, which is instantiated and exported directly from every executable module, can be used to access the current instance.

Form 2 `TDocTemplateT(TRegList& regList, TModule*& module = ::Module, TDocTemplate*& phead = DocTemplateStaticHead);`

Constructs a *TDocTemplateT* using the registration table to determine the file filter pattern, search path for viewing the directory, default file extension, and flag values. See the entry in this manual for registration macros for more information about how the registration tables are created. *module*, which is instantiated and exported directly from every executable module, can be used to access the current instance.

Public member functions

Clone

`TDocTemplateT* Clone(TModule* module, TDocTemplate*& phead = DocTemplateStaticHead;`

Makes a copy of the *TDocTemplateT* object.

CreateDoc

`D* CreateDoc(const char far* path, long flags = 0);`

CreateDoc creates a document of type *D* based on the directory path (*path*) and *flags* value.

See also TDocTemplate::CreateDoc

CreateView

`TView* CreateView(TDocument& doc, long flags = 0);`

CreateView creates the view specified by the document template class.

See also TDocManager::CreateAnyView

IsMyKindOfDoc

`D* IsMyKindOfDoc(TDocument& doc);`

IsMyKindOfDoc tests to see if the document (*doc*) is the same class as the template's document class or if the document is a derived class. If the template can't use the document, *IsMyKIndOfDoc* returns 0.

See also TDocTemplate::IsMyKindOfDoc

IsMyKindOfView

V* IsMyKindOfView(TView& view);

IsMyKindOfView tests to see if the view (*view*) is the same class as the template's view class or if the view is a derived class. If the template can't use the view, *IsMyKIndOfView* returns 0.

GetViewName

virtual const char far* GetViewName();

GetViewName gets the name of the view associated with the template.

TDropInfo class

geometry.h

TDropInfo is a simple class that supports file-name drag and drop operations using the WM_DROPFILES message. Each *TDropInfo* object has a private handle to the HDROP structure returned by the WM_DOPFILES message.

Public constructor

Constructor

TDropInfo(HDROP handle);

Creates a *TDropInfo* object with *Handle* set to the given *handle*.

Public member functions

DragFinish

void DragFinish();

Releases any memory allocated for the transferring of this *TDropInfo* object's files during drag operations.

DragQueryFile

uint DragQueryFile(uint index, char far* name, uint nameLen)

Retrieves the name of the file and related information for this *i* object. If *index* is set to –1 (0xFFFF), *DragQueryFile* returns the number of dropped files. This is equivalent to calling *DragQueryFileCount*.

If *index* lies between 0 and the total number of dropped files for this object, *DragQueryFile* copies to the *name* buffer (of length *nameLen* bytes) the name of the dropped file that corresponds to *index*, and returns the number of bytes actually copied.

If *name* is 0, *DragQueryFile* returns the required buffer size (in bytes) for the given *index*. This is equivalent to calling *DragQueryFileNameLen*.

See also TDropInfo::DragQueryPoint, TDropInfo::DragQueryFileCount

DragQueryFileCount

uint DragQueryFileCount();

Returns the number of dropped files in this *TDropInfo* object. This call is equivalent to calling `DragQueryFile(-1, 0, 0)`.

See also TDropInfo::DragQueryFile

DragQueryFileNameLen

uint DragQueryFileNameLen(uint index)

Returns the length of the name of the file in this *TDropInfo* object corresponding to the given index. This call is equivalent to calling `DragQueryFile(index, 0, 0)`.

See also TDropInfo::DragQueryFile

DragQueryPoint

bool DragQueryPoint(TPoint& point)

Retrieves the mouse pointer position when this object's files are dropped and copies the coordinates to the given *point* object. *point* refers to the window that received the WM_DROPFILES message. *DragQueryPoint* returns **true** if the drop occurs inside the window's client area, otherwise **false**.

See also TPoint class

HDROP()

operator HDROP();

Typecasting operator that returns *Handle*.

TLangId typedef — locale.h

typedef unsigned short TLangId;

Holds a language ID, a predefined number that represents a base language and dialect. For example, the number 409 represents American English. *TLocaleString* uses the language ID to find the correct translation for strings.

See also TLocaleString

TPoint class — geometry.h

TPoint is a support class, derived from *tagPOINT*. Under Win32, the latter is defined as

```
typedef struct tagPOINT {
   int x;
   int y;
};
```

TPoint encapsulates the notion of a two-dimensional point that usually represents a screen position. *TPoint* inherits two data members, the coordinates *x* and *y*, from *tagPOINT*. Member functions and operators are provided for comparing, assigning, and manipulating points. Overloaded << and >> operators are declared as friends of *TPoint*, allowing chained insertion and extraction of *TPoint* objects with streams.

Public constructors

Constructors

Form 1 TPoint();
The default *TPoint* constructor.

Form 2 TPoint(int _x, int _y);
Creates a *TPoint* object with the given coordinates.

Form 3 TPoint(const POINT& point);
Creates a *TPoint* object with *x = point. x y = point.y.*

Form 4 TPoint(const SIZE& size);
Creates a *TPoint* object with *x = size.cx* and *y = size.cy.*

Form 5 TPoint(uint32 dw);
Creates a *TPoint* object with *x = LOWORD(dw) y = HIWORD(dw)).*

See also TPOINT, TSIZE

Public member functions

Offset

TPoint& Offset(int dx, int dy);
Offsets this point by the given delta arguments. This point is changed to $(x + dx, y + dy)$. Returns a reference to this point.

See also TPoint::OffsetBy, TPoint::operator +=

OffsetBy

TPoint OffsetBy(int dx, int dy) const;
Calculates an offset to this point using the given displacement arguments. Returns the point $(x + dx, y + dy)$. This point is not changed.

See also TPoint::operator +, TPoint::Offset

operator +

TPoint operator +(const TSize& size) const;
Calculates an offset to this point using the given size argument as the displacement. Returns the point $(x + size.cx, y + size.cy)$. This point is not changed.

See also TPoint::OffsetBy, TSize class

operator –

TPoint operator –(const TSize& size) const;
TSize operator –(const TPoint& point) const;
TPoint operator –() const;
The first version calculates a negative offset to this point using the given *size* argument as the displacement. Returns the point $(x - size.cx, y - size.cy)$. This point is not changed.

The second version calculates a distance from this point to the *point* argument. Returns the *TSize* object $(x - point.x, y - point.y)$. This point is not changed.

The third version returns the point $(-x, -y)$. This point is not changed.

See also TPoint::operator +, TSize class

operator ==

bool operator ==(const TPoint& other) const;

Returns true if this point is equal to the *other* point; otherwise returns false.

See also TPoint::operator !=

operator +=

TPoint& operator +=(const TSize& size);

Offsets this point by the given *size* argument. This point is changed to(*x* + *size.cx*, *y* + *size.cy*). Returns a reference to this point.

See also TPoint::Offset, TPoint::operator –=, TSize class

operator –=

TPoint& operator –=(const TSize& size);

Negatively offsets this point by the given *size* argument. This point is changed to (*x* – *size.cx*, *y* – *size.cy*). Returns a reference to this point.

See also TPoint::Offset, TPoint::operator +=, TSize class

operator !=

bool operator !=(const TPoint& other) const;

Returns **false** if this point is equal to the *other* point; otherwise returns **true**.

See also TPoint::operator ==

operator >>

Form 1 ipstream& operator >>(ipstream& is, TPoint& p);

Extracts a *TPoint* object from persistent stream *is*, and copies it to *p*. Returns a reference to the resulting stream, allowing the usual chaining of << operations.

Form 2 istream& operator >>(istream& is, TPoint& p);

Extracts a *TPoint* object from stream *is*, and copies it to *p*. Returns a reference to the resulting stream, allowing the usual chaining of >> operations.

See also TPoint operator <<, class ipstream

operator <<

Form 1 opstream& operator <<(opstream& os, const TPoint& p);

Inserts the given *TPoint* object *p* into persistent stream *os*. Returns a reference to the resulting stream, allowing the usual chaining of >> operations.

Form 2 ostream& operator <<(ostream& os, const TPoint& p);

Formats and inserts the given *TPoint* object *p* into the *ostream os*. The format is "(x,y)". Returns a reference to the resulting stream, allowing the usual chaining of << operations.

See also TPoint operator >>, ostream

TPointer<> class

geometry.h

A small utility class that provides automatic destruction for objects constructed using **new**. *TPointer* is a parameterized class that holds a pointer to its parameterized type and overloads operators to behave like an object pointer. Assigning a pointer to a *TPointer* object eventually causes the object to be deleted when the function exits or when a *TPointer* goes out of scope or when another pointer is assigned to the same object. A *TPointer* object can be instantiated using one of the following equivalent methods:

```
TPointer<SomeClass> p = new SomeClass;
TPointer<SomeClass> p(new SomeClass);
TPointer<SomeClass> p;    p = new SomeClass;
```

To remove the object, assign 0 to the pointer.

Public constructors

Constructors

Form 1 `TPointer() : TPointerBase<T>();`
Default constructor in which *p* is initialized to 0.

Form 2 `TPointer(T* pointer) : TPointerBase<T>(pointer);`
Initialized constructor where *p* is initialized to *pointer*.

Public member functions

operator *

`T& operator *();`
Overloaded type conversion operator that casts its argument to a pointer to the type passed in.

operator T*

`operator T*();`
Overloaded type conversion operator that allows the *TPointer* object to be passed as a function argument or assigned to a variable as if it were a pointer.

operator =

Form 1 `T* operator =(T* src);`
Assignment operator *T** is assigned to *p*.

Form 2 `T* operator =(const TPointer<T>& src);`
Assignment operator used when *r* is a **const** reference to *TPointer* of *T*. This operator saves converting if another pointer object is used.

operator ->

`T* operator ->();`
Provides access to the pointer.

TProcInstance class

geometry.h

Designed for Win16 applications, *TProcInstance* handles creating and freeing an instance thunk, a piece of code created for use with exported callback functions. (A callback function is a function that exists within a program but is called from outside the program by a Windows library routine, for example, a dialog box function.)

For Win32 applications, *TProcInstance* is non-functional. The address returned from *TProcInstance* can be passed as a parameter to callback functions, window subclassing functions, or Windows dialog box functions.

See the Windows API online Help for more information about *MakeProcInstance*, which creates an instance thunk for the function and *FreeProcInstance*, which frees an instance thunk. For more information about exporting callback functions, see the *Borland C++ Programmer's Guide*.

Public constructor and destructor

Constructor

TProcInstance(FARPROC p);

Makes a *TProcInstance*, passing *p* as the address of the procedure. Under Win16, calls *::MakeProcInstance* to make an instance thunk for *p*. Under Win32, the constructor just saves *p*.

Destructor

~TProcInstance()

Under WIN16 frees the instance thunk.

See also ::MakeProcInstance (Windows API), ::FreeProcInstance (Windows API)

Public member function

FARPROC()

operator FARPROC();

Under WIN16, returns the instance thunk. Under Win32, returns *p* from the constructor.

TRect class

geometry.h

TRect is a mathematical class derived from *tagRect*. The *tagRect* struct is defined as

```
struct tagRECT {
   int left;
   int top;
   int right;
   int bottom;
};
```

TRect encapsulates the properties of rectangles with sides parallel to the x- and y-axes. In ObjectWindows, these rectangles define the boundaries of windows, boxes, and

clipping regions. *TRect* inherits four data members from *tagRect left*, *top*, *right*, and *bottom*. These represent the top left and bottom right (*x*, *y*) coordinates of the rectangle. Note that *x* increases from left to right, and *y* increases from top to bottom.

TRect places no restrictions on the relative positions of top left and bottom right, so it is legal to have *left* > *right* and *top* > *bottom*. However, many manipulations—such as determining width and height, and forming unions and intersections—are simplified by normalizing the *TRect* objects involved. Normalizing a rectangle means interchanging the corner point coordinate values so that *left* < *right* and *top* < *bottom*. Normalization does not alter the physical properties of a rectangle. *myRect.Normalized* creates normalized copy of *myRect* without changing *myRect*, while *myRect.Normalize* changes *myRect* to a normalized format. Both members return the normalized rectangle.

TRect constructors are provided to create rectangles from either four **int**s, two *TPoint* objects, or one *TPoint* and one *TSize* object. In the latter case, the *TPoint* object specifies the top left point (also known as the rectangle's origin) and the *TSize* object supplies the width and height of the rectangle. Member functions perform a variety of rectangle tests and manipulations. Overloaded << and >> operators allow chained insertion and extraction of *TRect* objects with streams.

Public constructors

Constructors

Form 1 `TRect();`
The default constructor.

Form 2 `TRect(const RECT far& rect);`
Copies the given *rect* to this object.

Form 3 `TRect(int _left, int _top, int _right, int _bottom);`
Creates a rectangle with the given values.

Form 4 `TRect(const TPoint& upLeft, const TPoint& loRight);`
Creates a rectangle with the given top left and bottom right points.

Form 5 `TRect(const TPoint& origin, const TSize& extent);`
Creates a rectangle with its origin (top left) at *origin*, width at *extent.cx*, height at *extent.cy*.

See also TPoint class, TSize class

Public member functions

Area

`long Area() const;`
Returns the area of this rectangle.

See also TRect::Size

BottomLeft

`TPoint BottomLeft() const;`
Returns the *TPoint* object representing the bottom left corner of this rectangle.

See also TRect::TopLeft, TRect::TopRight, TRect::BottomRight, TPoint

BottomRight

```
const TPoint& BottomRight() const;
TPoint& BottomRight();
```

Returns the *TPoint* object representing the bottom right corner of this rectangle.

See also TRect::TopRight, TRect::BottomLeft, TRect::TopLeft, TPoint class

BottomRight

```
const TPoint& BottomRight() const;
TPoint& BottomRight();
```

Returns the *TPoint* object representing the bottom right corner of this rectangle.

See also TRect::TopRight, TRect::BottomLeft, TRect::TopLeft, TPoint class

Contains

Form 1

```
bool Contains(const TPoint& point) const;
```

Returns **true** if the given *point* lies within this rectangle; otherwise, it returns **false**. If *point* is on the left vertical or on the top horizontal borders of the rectangle, *Contains* also returns true, but if *point* is on the right vertical or bottom horizontal borders, *Contains* returns false.

Form 2

```
bool Contains(const TRect& other) const;
```

Returns **true** if the other rectangle lies on or within this rectangle; otherwise, it returns **false**.

See also TRect::Touches, TPoint class, TRect class

Height

```
int Height() const;
```

Returns the height of this rectangle (*bottom* – *top*).

See also TRect::Width

Inflate

```
TRect& Inflate(int dx, int dy);
TRect& Inflate(const TSize& delta);
```

Inflates a rectangle inflated by the given delta arguments. In the first version, the top left corner of the returned rectangle is (*left* – *dx*, *top* – *dy*), while its bottom right corner is (*right* + *dx*, *bottom* + *dy*). In the second version the new corners are (*left* – *size.cx*, *top* – *size.cy*) and (*right* + *size.cx*, *bottom* + *size.cy*).

See also TRect class, TSize class

InflatedBy

```
TRect InflatedBy(int dx, int dy) const;
TRect InflatedBy(const TSize& size) const;
```

Returns a rectangle inflated by the given delta arguments. In the first version, the top left corner of the returned rectangle is (*left* – *dx*, *top* – *dy*), while its bottom right corner is (*right* + *dx*, *bottom* + *dy*). In the second version the new corners are (*left* – *size.cx*, *top* – *size.cy*) and (*right* + *size.cx*, *bottom* + *size.cy*). The calling rectangle object is unchanged.

See also TRect::OffsetBy, TRect class, TSize class

IsEmpty

bool IsEmpty() const;

Returns **true** if *left* >= *right* or *top* >= *bottom*; otherwise, returns **false**.

See also TRect::SetEmpty, TRect::IsNull

IsNull

bool IsNull() const;

Returns **true** if left, *right*, *top*, and *bottom* are all 0; otherwise, returns **false**.

See also TRect::IsEmpty, TRect::SetEmpty

Normalize

TRect& Normalize();

Normalizes this rectangle by switching the *left* and *right* data member values if *left* > *right*, and switching the *top* and *bottom* data member values if *top* > *bottom*. *Normalize* returns the normalized rectangle. A valid but nonnormal rectangle might have *left* > *right* and/or *top* > *bottom*. In such cases, many manipulations (such as determining width and height) become unnecessarily complicated. Normalizing a rectangle means interchanging the corner point values so that *left* < *right* and *top* < *bottom*. The physical properties of a rectangle are unchanged by this process.

See also TRect::Normalized, TRect class

Normalized

TRect Normalized() const;

Returns a normalized rectangle with the top left corner at (Min(*left*, *right*), Min(*top*, *bottom*)) and the bottom right corner at (Max(*left*, *right*), Max(*top*, *bottom*)). The calling rectangle object is unchanged. A valid but nonnormal rectangle might have *left* > *right* and/or *top* > *bottom*. In such cases, many manipulations (such as determining width and height) become unnecessarily complicated. Normalizing a rectangle means interchanging the corner point values so that *left* < *right* and *top* < *bottom*. The physical properties of a rectangle are unchanged by this process.

Note that many calculations assume a normalized rectangle. Some Windows API functions behave erratically if an inside-out *Rect* is passed.

See also TRect::Normalize, TRect

Offset

TRect& Offset(int dx, int dy);

Changes this rectangle so its corners are offset by the given delta values. The revised rectangle has a top left corner at (*left* + *dx*, *top* + *dy*) and a right bottom corner at (*right* + *dx*, *bottom* + *dy*). The revised rectangle is returned.

See also TRect::operator +, TRect::operator +=, TRect::OffsetBy

OffsetBy

TRect OffsetBy(int dx, int dy) const;

Returns a rectangle with the corners offset by the given delta values. The returned rectangle has a top left corner at (*left* + *dx*, *top* + *dy*) and a right bottom corner at (*right* + *dx*, *bottom* + *dy*).

See also TRect::operator +

operator +

TRect operator +(const TSize& size) const;

Returns a rectangle offset positively by the delta values given size. The returned rectangle has a top left corner at (*left + size.x*, *top + size.y*) and a right bottom corner at (*right + size.x*, *bottom + size.y*). The calling rectangle object is unchanged.

See also TRect::OffsetBy, TSize class

operator –

TRect operator –(const TSize& size) const;

Returns a rectangle offset negatively by the delta values given size. The returned rectangle has a top left corner at (*left – size.cx*, *top – size.cy*) and a right bottom corner at (*right – size.cx*, *bottom – size.cy*). The calling rectangle object is unchanged.

See also TRect::OffsetBy, TSize class

operator &

TRect operator &(const TRect& other) const;

Returns the intersection of this rectangle and the *other* rectangle. The calling rectangle object is unchanged. Returns a NULL rectangle if the two don't intersect.

See also TRect::operator |, TRect::operator &=

operator |

TRect operator |(const TRect& other) const;

Returns the union of this rectangle and the *other* rectangle. The calling rectangle object is unchanged.

See also TRect::operator &, TRect::operator |=

operator ==

bool operator ==(const TRect& other) const;

Returns **true** if this rectangle has identical corner coordinates to the *other* rectangle; otherwise, returns **false**.

See also TRect::operator !=

operator !=

bool operator !=(const TRect& other) const;

Returns **false** if this rectangle has identical corner coordinates to the *other* rectangle; otherwise, returns **true**.

See also TRect::operator ==

operator +=

TRect& operator +=(const TSize& delta);

Changes this rectangle so its corners are offset by the given delta values, *delta.x* and *delta.y*. The revised rectangle has a top left corner at (*left + delta.x*, *top + delta.y*) and a right bottom corner at (*right + delta.x*, *bottom + delta.y*). The revised rectangle is returned.

See also TRect::operator +, TRect::OffsetBy, TRect::Offset

operator –=

TRect& operator –=(const TSize& delta);

Changes this rectangle so its corners are offset negatively by the given delta values, *delta.x* and *delta.y*. The revised rectangle has a top left corner at (*left – delta.x, top – delta.y*) and a right bottom corner at (*right – delta.x, bottom – delta.y*). The revised rectangle is returned.

See also TRect::operator –, TRect::operator +=, TRect::OffsetBy, TRect::Offset

operator &=

```
TRect& operator &=(const TRect& other);
```

Changes this rectangle to its intersection with the *other* rectangle. This rectangle object is returned. Returns a NULL rectangle if there is no intersection.

See also TRect::operator &, TRect::operator |=

operator |=

```
TRect& operator |=(const TRect& other);
```

Changes this rectangle to its union with the *other* rectangle. This rectangle object is returned.

See also TRect::operator |, TRect::operator &=

operator >>

```
ipstream& _BIDSFUNC operator >>(ipstream& is, TRect& r);
```

Extracts a *TRect* object from *is*, the given input stream, and copies it to *r*. Returns a reference to the resulting stream, allowing the usual chaining of >> operations.

See also TRect operator <<

operator <<

Form 1

```
opstream& _BIDSFUNC operator <<(opstream& os, const TRect& r);
```

Inserts the given *TRect* object, *r*, into the *opstream*, *os*. Returns a reference to the resulting stream, allowing the usual chaining of << operations.

Form 2

```
ostream& _BIDSFUNC operator <<(ostream& os, const TRect& r);
```

Formats and inserts the given *TRect* object, *r*, into the *ostream*, *os*. The format is (*r.left, r.top*)(*r.right, r.bottom*). Returns a reference to the resulting stream and allows the usual chaining of << operations.

See also TRect operator >>

operator TPoint*()

```
operator const TPoint*() const;
operator TPoint*()
```

Type conversion operators converting the pointer to this rectangle to type pointer to *TPoint*.

See also TPoint class

Set

```
void Set(int _left, int _top, int _right, int _bottom);
```

Repositions and resizes this rectangle to the given values.

SetEmpty

```
void SetEmpty();
```

Empties this rectangle by setting *left*, *top*, *right*, and *bottom* to 0.

SetNull

```
void SetNull();
```

Sets the *left*, *top*, *right*, and *bottom* of the rectangle to 0.

Size

```
TSize Size() const;
```

Returns a *TSize* object representing the width and height of this rectangle.

See also TSize class

TopLeft

```
const TPoint& TopLeft() const;
TPoint& TopLeft();
```

Returns the *TPoint* object representing the top left corner of this rectangle.

See also TRect::TopRight, TRect::BottomLeft, TRect::BottomRight, TPoint class

TopRight

```
TPoint TopRight() const;
```

Returns the *TPoint* object representing the top right corner of this rectangle.

See also TRect::TopLeft, TRect::BottomLeft, TRect::BottomRight

Touches

```
bool Touches(const TRect& other) const;
```

Returns **true** if the *other* rectangle shares any interior points with this rectangle; otherwise, returns **false**.

See also TRect::Contains

Width

```
int Width() const;
```

Returns the width of this rectangle (*right* – *left*).

See also TRect::Height

TResId class

geometry.h

A simple support class, *TResId* creates a resource ID object from either an integer or an actual string identifier. For example, *TResId* encapsulates the use of *LPSTR* (**char_far***) as a resource identifier. This resource identifier can be passed to various ObjectWindows classes. To handle these two different types of resource identifiers, *TResId* defines a conversion operator and provides two constructors that convert and use these native data types. One constructor accepts a 16-bit integer and the other accepts a character string.

Public constructors

Constructors

Form 1 TResId();
The default *TResId* constructor.

Form 2 TResId(int resNum);
Creates a *TResId* object with the given *resNum*.

Form 3 TResId(const char far* resString);
Creates a *TResId* object with the given *resString*.

Public member functions

char far*
operator char far*();
Typecasting operator that converts *Id* (a *TResId* private data member) to type **char far*** so that instances of *TResId* can be used in places where **char far*** data types are expected.

IsString
bool IsString() const;
Returns **true** if this resource ID was created from a string; otherwise, returns **false**.

Friend functions

operator >>
friend ipstream& operator >>(ipstream& is, TResId& id);
Extracts a *TResId* object from *is* (the given input stream), and copies it to *id*. Returns a reference to the resulting stream, allowing the usual chaining of >> operations.

See also TResId friend operator <<, ipstream

operator <<

Form 1 friend opstream& operator <<(opstream& os, const TResId& id);
Inserts the given *TResId* object (*id*) into the *opstream* (*os*). Returns a reference to the resulting stream, allowing the usual chaining of << operations.

Form 2 friend ostream& operator <<(ostream& os, const TResId& id);
Formats and inserts the given *TResId* object (*id*) into the *ostream* (*os*). Returns a reference to the resulting stream, allowing the usual chaining of << operations.

See also TResId friend operator >>, ostream, opstream

TSize class

geometry.h

TSize is a mathematical class derived from the structure *tagSIZE*.

The *tagSIZE* struct is defined as

```
struct tagSIZE {
```

```
    int cx;
    int cy;
};
```

TSize encapsulates the notion of a two-dimensional quantity that usually represents a displacement or the height and width of a rectangle. *TSize* inherits the two data members *cx* and *cy* from *tagSIZE*. Member functions and operators are provided for comparing, assigning, and manipulating sizes. Overloaded << and >> operators allow chained insertion and extraction of *TSize* objects with streams.

Public constructors

Constructors

Form 1 TSize();
The default *TSize* constructor.

Form 2 TSize(int dx, int dy);
Creates a *TSize* object with *cx* = *dx* and *cy* = *dy*.

Form 3 TSize(const POINT& point);
Creates a *TSize* object with *cx* = *point.x* and cy = *point.y*.

Form 4 TSize(const SIZE& size);
Creates a *TSize* object with *cx* = *size.cx* and *cy* = *size.cy*.

Form 5 TSize(uint32 dw);
Creates a *TSize* object with *cx* = *LOWORD(dw)* and *cy* = *HIWORD(dw))*.

See also TPoint class, Size struct

Public member functions

Magnitude

int Magnitude() const;
Returns the length of the diagonal of the rectangle represented by this object. The value returned is an **int** approximation to the square root of (*cx* + *cy*).

operator +

TSize operator +(const TSize& size) const;
Calculates an offset to this *TSize* object using the given *size* argument as the displacement. Returns the object (*cx* + *size.cx*, *cy* + *size.cy*). This *TSize* object is not changed.

See also TSize::operator –

operator –

Form 1 TSize operator –(const TSize& size) const;
The first version calculates a negative offset to this *TSize* object using the given *size* argument as the displacement. Returns the point (*cx* – *size.cx*, *cy* – *size.cy*). This object is not changed.

Form 2 TSize operator –() const;

The second version returns the *TSize* object (*–cx, –cy*). This object is not changed.

See also TSize::operator +

operator ==

bool operator ==(const TSize& other) const;

Returns **true** if this size object is equal to the *other TSize* object; otherwise returns **false**.

See also TSize::operator !=

operator !=

bool operator !=(const TSize& other) const;

Returns **false** if this size object is equal to the *other TSize* object; otherwise returns **true**.

See also TSize::operator ==

operator +=

TSize& operator +=(const TSize& size);

Offsets this *TSize* object by the given *size* argument. This *TSize* object is changed to (*cx + size.cx, cy + size.cy*). Returns a reference to this object.

See also TSize::operator –=

operator –=

TSize& operator –=(const TSize& size);

Negatively offsets this *TSize* object by the given *size* argument. This object is changed to (*cx – size.cx, cy – size.cy*). Returns a reference to this object.

See also TSize::operator +=

operator >>

ipstream& operator >>(ipstream& is, TSize& s);

Extracts a *TSize* object from *is*, the given input stream, and copies it to *s*. Returns a reference to the resulting stream, allowing the usual chaining of >> operations.

See also TSize operator <<, ipstream

operator <<

Form 1 opstream& operator <<(opstream& os, const TSize& s);

Inserts the given *TSize* object (*s*) into the *opstream* (*os*). Returns a reference to the resulting stream, allowing the usual chaining of << operations.

Form 2 ostream& operator <<(ostream& os, const TSize& s);

Formats and inserts the given *TSize* object (*s*) into the *ostream* (*os*). The format is "(*cx x cy*)". Returns a reference to the resulting stream, allowing the usual chaining of << operations.

See also TSize operator >>, opstream, ostream

TXBase class

except.h

Derived from *xmsg*, *TXBase* is the base class for ObjectWindows and ObjectComponents exception-handling classes. The ObjectWindows classes that handle specific kinds of

exceptions, for example out-of-memory or invalid window exceptions are derived from *TXOwl*, which is in turn derived from *TXBase*. The ObjectComponents classes, *TXOle* and *TXAuto*, are derived directly from *TXBase*.

TXBase contains the functions, *Clone* and *Throw*, which are overridden in all derived classes, as well as two constructors. The constructors increment, *InstanceCount*, *TXBase's* public data member, and the destructor decrements *InstanceCount*.

See the *Borland C++ Library Reference* for a description of *xmsg*, *TXBase's* parent class. See the *ObjectWindows Programmer's Guide* for information about how to use *TXBase* in your applications.

See also
TXOwl

Public constructors and destructor

Constructors
Form 1 `TXBase(const string& msg);`
Calls the *xmsg* class's constructor that takes a string parameter and initializes *xmsg* with the value of the string parameter.

Form 2 `TXBase(const TXBase& src);`
Creates a copy of the *TXBase* object passed in the *TXBase* parameter.

Destructor
`virtual ~TXBase;`
Destroys the *TXBase* object and decrements the *InstanceCount* data member

See also TXOwl public constructors and destructor

Public data member

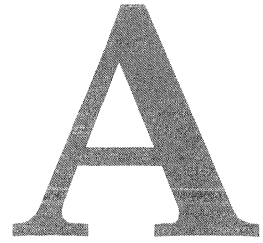
InstanceCount
`static int InstanceCount;`
Counts the number of *TXBase* and *TXBase*-derived objects existing in a single application.

Public member functions

Clone
`virtual TXBase* Clone();`
Makes a copy of the exception object.

Throw
`virtual void Throw();`
Throws the exception object.

Appendix



Windows API encapsulated functions

This appendix includes several tables listing encapsulated Windows API functions that take an HWND as the first argument. The tables are organized in the following manner:

Most of the encapsulated functions are implemented as inline functions that pass the HWindow member variable as the HWND argument. The remaining arguments are passed without changing their prototype.

The other functions are static functions that return information regarding windows. These functions don't actually use HWindow as the argument because the HWND is

either implied or the function returns the HWND. An HWND that serves as a handle to an window can be converted to a TWindow* by using *GetWindowPtr()*.

Note that in the scope of *TWindow* or derived class, all direct calls to the corresponding Windows versions of these functions must be globally scoped, as in *::SendMessage()*.

The functions in Table A.1 allow a *TWindow* to be used as a HWND in Windows API calls.

Table A.1 Encapsulated inline HWND functions

Name	Function declaration
HWND	operator HWND () const
IsWindow	BOOL IsWindow () const

The functions in Table A.2 handle Window messages.

Table A.2 Encapsulated Window messages

Name	Function declaration
EnableWindow	BOOL EnableWindow (BOOL enable)
GetCapture	static HWND GetCapture ()
GetFocus	static HWND GetFocus ()
IsWindowEnabled	BOOL IsWindowEnabled () const
PostMessage	BOOL PostMessage (UINT msg, WPARAM wParam = 0, LPARAM lParam = 0) const
ReleaseCapture	static void ReleaseCapture ()
SendDlgItemMessage	LRESULT SendDlgItemMessage (int childId, UINT msg, WPARAM wParam = 0, LPARAM lParam = 0) const
SendMessage	LRESULT SendMessage (UINT msg, WPARAM wParam = 0, LPARAM lParam = 0) const
SetCapture	HWND SetCapture ()
SetFocus	HWND SetFocus ()

Table A.3 lists functions that adjust window coordinates and sizes.

Table A.3 Window coordinates and dimensions

Name	Function declaration
AdjustWindowRect	static void AdjustWindowRect (TRect& rect, DWORD style, BOOL menu)
AdjustWindowRectEx	static void AdjustWindowRectEx (TRect& rect, DWORD style, BOOL menu, DWORD exStyle)
ChildWindowFromPoint	HWND ChildWindowFromPoint (const TPoint& point)
ClientToScreen	void ClientToScreen (TPoint& point) const
GetClientRect	TRect GetClientRect ()
GetClientRect	void GetClientRect (TRect& rect)
GetWindowRect	void GetWindowRect (TRect& rect)
MapWindowPoints	void MapWindowPoints (HWND hWndTo, TPoint *points, int count) const
ScreenToClient	void ScreenToClient (TPoint& point) const
WindowFromPoint	static HWND WindowFromPoint (const TPoint& point)

Table A.4 lists functions that encapsulate window properties and style attributes.

Table A.4 Window properties

Name	Function declaration
EnumProps	int EnumProps (PROPENUMPROC proc)
GetClassLong	long GetClassLong (int index) const
GetClassName	long GetClassName (char far* className, int maxCount) const
GetClassWord	WORD GetClassWord (int index) const
GetProp	HANDLE GetProp (const char far* str) const
GetProp	HANDLE GetProp (WORD atom) const
GetWindowLong	long GetWindowLong (int index) const
GetWindowWord	WORD GetWindowWord (int index) const
RemoveProp	HANDLE RemoveProp (const char far* str) const
RemoveProp	HANDLE RemoveProp (WORD atom) const
SetClassLong	long SetClassLong (int index, long newLong)
SetClassWord	WORD SetClassWord (int index, WORD newWord)
SetProp	BOOL SetProp (const char far* str, HANDLE data) const
SetProp	BOOL SetProp (WORD atom, HANDLE data) const
SetWindowLong	long SetWindowLong (int index, long newLong)
SetWindowWord	WORD SetWindowWord (int index, WORD newWord)

Table A.5 lists functions that encapsulate window placement and display properties.

Table A.5 Window placement

Name	Function declaration
GetWindowPlacement	BOOL GetWindowPlacement (WINDOWPLACEMENT *place) const
GetWindowText	int GetWindowText (char far* str, int maxCount) const
GetWindowTextLength	int GetWindowTextLength () const
IsIconic	BOOL IsIconic () const
IsWindowVisible	BOOL IsWindowVisible () const
IsZoomed	BOOL IsZoomed () const
MoveWindow	void MoveWindow (const TRect& rect, BOOL repaint = FALSE)
MoveWindow	void MoveWindow (int x, int y, int w, int h, BOOL repaint = FALSE)
SetWindowPlacement	BOOL SetWindowPlacement (const WINDOWPLACEMENT *place)
SetWindowText	void SetWindowText (const char far* str)
ShowOwnedPopups	void ShowOwnedPopups (BOOL show)
ShowWindow	BOOL ShowWindow (int cmdShow)

The functions in Table A.6 control window positions and sibling relationships.

Table A.6 Window relationships

Name	Function declaration
BringWindowToTop	void BringWindowToTop ()
GetActiveWindow	static HWND GetActiveWindow ()

Table A.6 Window relationships (continued)

Name	Function declaration
GetDesktopWindow	static HWND GetDesktopWindow ()
GetLastActivePopup	HWND GetLastActivePopup () const
GetNextWindow	HWND GetNextWindow (UINT dirFlag) const
GetSysModalWindow	static HWND GetSysModalWindow ()
GetTopWindow	HWND GetTopWindow () const
SetActiveWindow	HWND SetActiveWindow ()
SetSysModalWindow	HWND SetSysModalWindow ()
SetWindowPos	void SetWindowPos (HWND hWndInsertAfter, const TRect& rect, UINT flags)
SetWindowPos	void SetWindowPos (HWND hWndInsertAfter, int x, int y, int w, int h, UINT flags)

The encapsulated functions in Table A.7 control window painting, invalidating, validating, and updating.

Table A.7 Window painting functions

Name	Function declaration
FlashWindow	BOOL FlashWindow (BOOL invert)
GetUpdateRect	BOOL GetUpdateRect (TRect& rect, BOOL erase = TRUE) const
Invalidate	void Invalidate (BOOL erase = TRUE)
InvalidateRect	void InvalidateRect (const TRect& rect, BOOL erase = TRUE)
InvalidateRgn	void InvalidateRgn (HRGN hRgn, BOOL erase = TRUE)
LockWindowUpdate	BOOL LockWindowUpdate ()
RedrawWindow	BOOL RedrawWindow (TRect *update, HRGN hUpdateRgnh, UINT redrawFlags = RDW_INVALIDATE | RDW_UPDATENOW | RDW_ERASE)
UpdateWindow	void UpdateWindow ()
Validate	void Validate ()
ValidateRect	void ValidateRect (const TRect& rect)
ValidateRgn	void ValidateRgn (HRGN hRgn)

The functions in Table A.8 control window scrolling and scroll bars.

Table A.8 Window scrolling functions

Name	Function declaration
GetScrollPos	int GetScrollPos (int bar)
GetScrollRange	void GetScrollRange (int bar, int &minPos, int &maxPos) const
ScrollWindow	void ScrollWindow (int dx, int dy, const TRect *scroll = 0, const TRect *clip = 0)
ScrollWindowEx	void ScrollWindowEx (int dx, int dy, const TRect *scroll = 0, const TRect *clip = 0, HRGN hUpdateRgn = 0, TRect *update = 0, UINT flags = 0)
SetScrollPos	int SetScrollPos (int bar, int pos, BOOL redraw = TRUE)
SetScrollRange	void SetScrollRange (int bar, in minPos, int maxPos, BOOL redraw = TRUE)
ShowScrollBar	void ShowScrollBar (int bar, BOOL show = TRUE)

The functions in Table A.9 control parent and child windows using command IDs.

Table A.9 Child window ID functions

Name	Function declaration
CheckDlgButton	void CheckDlgButton (int buttonId, UINT check)
CheckRadioButton	void CheckRadioButton (int firstButtonId, int lastButtonId, int checkButtonId)
GetDlgCtrlID	int GetDlgCtrlID () const
GetDlgItem	HWND GetDlgItem (int childId) const
GetDlgItemInt	UINT GetDlgItemInt (int childId, BOOL * translated, BOOL isSigned) const
GetDlgItemText	int GetDlgItemText (int childId, char far* text, int max) const
GetNextDlgGroupItem	HWND GetNextDlgGroupItem (HWND hWndCtrl, BOOL previous = FALSE) const
GetNextDlgTabItem	HWND GetNextDlgTabItem (HWND hWndCtrl, BOOL previous = FALSE) const
GetParent	HWND GetParent () const
IsChild	BOOL IsChild (HWND) const
IsDlgButtonChecked	UINT IsDlgButtonChecked (int buttonId) const
SetDlgItemInt	void SetDlgItemInt (int childId, UINT value, BOOL is Signed = TRUE) const
SetDlgItemText	void SetDlgItemText (int childId, const char far* text) const

The functions in Table A.10 control menus and menu bar operations.

Table A.10 Menu and menu bar functions

Name	Function declaration
DrawMenuBar	void DrawMenuBar ()
GetMenu	HMENU GetMenu ()
GetSystemMenu	HMENU GetSystemMenu (BOOL revert = FALSE)
HiliteMenuItem	BOOL HiliteMenuItem (HMENU, UINT idItem, UINT hilite)
SetMenu	BOOL SetMenu (HMENU)

The functions in Table A.11 controls Clipboard operations.

Table A.11 Clipboard functions

Name	Function declaration
&OpenClipboard	TClipBoard &OpenClipboard ()

The functions in Table A.12 control timer operations.

Table A.12 Timer functions

Name	Function declaration
KillTimer	BOOL KillTimer (UINT timerId)
SetTimer	BOOL SetTimer (UINT timerId, UINT timeout, TIMERPROC proc = 0)

The functions in Table A.13 control caret and cursor operations.

Table A.13 Caret and cursor functions

Name	Function declaration
CreateCaret	void CreateCaret (HBITMAP)
CreateCaret	void CreateCaret (int shade, int width, int height)
DestroyCaret	static void DestroyCaret ()
GetCaretBlinkTime	static UINT GetCaretBlinkTime ()
GetCaretPos	static void GetCaretPos (TPoint& pos)
GetCursorPos	static void GetCursorPos (TPoint& pos)
HideCaret	void HideCaret ()
SetCaretBlinkTime	static void SetCaretBlinkTime (WORD milliSecs)
SetCaretPos	static void SetCaretPos (const TPoint& pos)
SetCaretPos	static void SetCaretPos (int x, int y)

The functions in Table A.14 control the operations of hot keys.

Table A.14 Hot key functions

Name	Function declaration
RegisterHotKey	BOOL RegisterHotKey (int idHotKey, UINT modifiers, UINT virtKey)[1]
UnregisterHotKey	BOOL UnregisterHotKey (int idHotKey)[1]

1. WIN32 API only

The functions in Table A.15 control miscellaneous operations such as accessing WinHelp.

Table A.15 Help and task functions

Name	Function declaration
DragAcceptFiles	void DragAcceptFiles (BOOL accept)
GetWindowTask	HANDLE GetWindowTask () const[1]
GetWindowTask	HTASK GetWindowTask () const[2]
MessageBox	int MessageBox (const char far* text, const char far* caption = 0, UINT type = MB_OK)
WinHelp	BOOL WinHelp (const char far* helpFile, UINT command, DWORD data)

1. WIN32 API only
2. WIN16 API only

Appendix

B

Windows API structs

This appendix lists several Windows API structs that ObjectWindows uses.

ABC struct

windows.h

```
typedef struct _ABC {
  int abcA;
  UINT abcB;
  int abcC;
} ABC;
```

The ABC structure contains the width of a character in a TrueType® font.

Member	Description
abcA	"A" character spacing. "A" spacing is the distance to add to the current position before drawing the character glyph.
abcB	"B" character spacing. "B" spacing is the width of the drawn portion of the character glyph.
abcC	"C" character spacing. "C" spacing is the distance to add to the current position to provide white space to the right of the character glyph.

The total width of a character is the sum of *abcA*, *abcB*, and *abcC*. Either the *abcA* or *abcC* can be negative to indicate underhangs or overhangs.

BITMAP struct

windows.h

16-bit version:

```
typedef struct tagBITMAP {
  int bmType;
  int bmWidth;
  int bmHeight;
```

```
  int bmWidthBytes;
  BYTE bmPlanes;
  BYTE bmBitsPixel;
  void FAR* bmBits;
} BITMAP;
```

32-bit version:

```
typedef struct tagBITMAP {
  LONG bmType;
  LONG bmWidth;
  LONG bmHeight;
  LONG bmWidthBytes;
  WORD bmPlanes;
  WORD bmBitsPixel;
  LPVOID bmBits;
} BITMAP;
```

BITMAP defines the height, width, color format, and bit values of a logical bitmap.

Member	Description
bmType	Bitmap type. For logical bitmaps, this member must be zero.
bmWidth	Bitmap width in pixels. Must be greater than zero.
bmHeight	Bitmap height in raster lines. Must be greater than zero.
bmWidthBytes	Number of bytes per raster line. Must be an even number because the graphics device interface (GDI) assumes that the bit values of a bitmap form an array of integer (two-byte) values.
bmPlanes	Number of bitmap color planes.
bmBitsPixel	Number of adjacent color bits on each plane needed to define a pixel.
bmBits	Points to the location of the bitmap bit values. Must be a long pointer to an array of one-byte values.

Currently used bitmap formats are monochrome and color. Monochrome bitmaps use a one-bit, one-plane format. Each scan is a multiple of 16 bits or 32 bits. A monochrome bitmap of height *n* is organized as follows:

```
Scan 0
Scan 1
  ...
Scan n-2
Scan n-1
```

Monochrome device pixels are black or white. If a bitmap bit is 1, the corresponding pixel is turned on (white). If a bitmap is 0, the corresponding pixel is turned off (black).

BITMAPCOREHEADER struct

windows.h

```
typedef struct tagBITMAPCOREHEADER {
 DWORD bcSize;
 WORD bcWidth;
 WORD bcHeight;
```

```
 WORD bcPlanes;
 WORD bcBitCount;
} BITMAPCOREHEADER;
```

Contains device-independent bitmap (DIB) dimension and color-format information.

Member	Description
bcSize	Number of bytes required by the structure.
bcWidth	Width of the bitmap, in pixels.
bcHeight	Height of the bitmap, in pixels.
bcPlanes	Number of planes for the target device. Must be 1.
bcBitCount	Number of bits per pixel. Must be 1, 4, 8, or 24.

This structure is combined with a color table in the *BITMAPCOREINFO* structure to provide a complete definition of the dimensions and colors of a DIB.

See also BITMAPCOREINFO struct, RGBTRIPLE struct

BITMAPCOREINFO struct

windows.h

```
typedef struct _BITMAPCOREINFO {
 BITMAPCOREHEADER bmciHeader;
 RGBTRIPLE bmciColors[1];
} BITMAPCOREINFO;
```

Defines the dimensions and color information for a device-independent bitmap (DIB).

Member	Description
bmciHeader	Structure containing DIB color and dimension information.
bmciColors	Structure defining bitmap colors.

A DIB consists of two parts: a *BITMAPCOREINFO* structure describing the dimensions and colors of the bitmap, and an array of bytes defining the pixels of the bitmap. The bits in the array are packed together, but each scan line must be padded with zeroes to end on a LONG boundary. The origin of the bitmap is the lower left corner.

The *bcBitCount* member of the *BITMAPCOREHEADER* structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap. This member can be one of the following values:

Value	Description
1	The bitmap is monochrome, and the *bmciColors* member contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the *bmciColors* table; if the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the *bmciColors* member contains up to 16 entries. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.

Value	Description
8	The bitmap has a maximum of 256 colors, and the *bmciColors* member contains up to 256 entries. In this case, each byte in the array represents a single pixel.
24	The bitmap has a maximum of 2 colors, and the *bmciColors* member is NULL. Each 3-byte triplet in the bitmap array represents the relative intensities of red, green, and blue, respectively, for a pixel.

The colors in the *bmciColors* table should appear in order of importance.

Alternatively, for functions that use DIBs, the *bmciColors* member can be an array of 16-bit unsigned integers that specify indexes into the currently realized logical palette, instead of explicit RGB values. In this case, an application using the bitmap must call the DIB functions (*CreateDIBitmap*, *CreateDIBPatternBrush*, and *CreateDIBSection*) with the iUsage parameter set to DIB_PAL_COLORS.

Note The *bmciColors* member should not contain palette indexes if the bitmap is to be stored in a file or transferred to another application. Unless the application has exclusive use and control of the bitmap, the bitmap color table should contain explicit RGB values.

BITMAPINFO struct

windows.h

16-bit version:

```
typedef struct tagBITMAPINFO {
 BITMAPINFOHEADER bmiHeader;
 RGBQUAD bmiColors[1];
} BITMAPINFO;
```

Defines the dimensions and color information for a device-independent bitmap (DIB).

Member	Description
bmiHeader	Structure containing DIB dimension and color-format information.
bmiColors	Structures that defines bitmap colors.

A DIB consists of a *BITMAPINFO* structure, which describes the dimensions and colors of the bitmap, and an array of bytes defining the bitmap pixels. The array bits are packed together, but each scan line must be zero-padded to end on a LONG boundary. Segment boundaries can appear anywhere in the bitmap. The bitmap origin is the lower-left corner.

The *biBitCount* member of the *BITMAPINFOHEADER* structure determines the number of bits which define each pixel and the maximum number of colors in the bitmap. This member may be set to any of the following values:

Value	Description
1	The bitmap is monochrome, and the *bmciColors* member must contain two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the *bmciColors* table. If the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the *bmciColors* member contains 16 entries. Each pixel in the bitmap is represented by a four-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	The bitmap has a maximum of 256 colors, and the *bmciColors* member contains 256 entries. In this case, each byte in the array represents a single pixel.
16	The bitmap has a maximum of 2^16 colors. The *biCompression* member of the *BITMAPINFOHEADER* must be BI_BITFIELDS. The *bmiColors* member contains 3 DWORD color masks which specify the red, green, and blue components, respectively, of each pixel. Bits set in the DWORD mask must be contiguous and should not overlap the bits of another mask. All the bits in the pixel do not have to be used. Each WORD in the array represents a single pixel.
24	The bitmap has a maximum of 2 colors. The *bmciColors* member is NULL, and each 3-byte sequence in the bitmap array represents the relative intensities of red, green, and blue, respectively, of a pixel.
32	The bitmap has a maximum of 2 colors. The *biCompression* member of the *BITMAPINFOHEADER* must be BI_BITFIELDS. The bmiColors member contains 3 DWORD color masks which specify the red, green, and blue components, respectively, of each pixel. Bits set in the DWORD mask must be contiguous and should not overlap the bits of another mask. All the bits in the pixel do not have to be used. Each DWORD in the array represents a single pixel.

The *biClrUsed* member of the *BITMAPINFOHEADER* structure specifies the number of color indexes in the color table actually used by the bitmap. If the *biClrUsed* member is set to zero, the bitmap uses the maximum number of colors corresponding to the value of the *biBitCount* member. The colors in the *bmiColors* table should appear in order of importance.

Alternatively, for functions that use DIBs, the *bmiColors* member can be an array of 16-bit unsigned integers that specify an index into the currently realized logical palette instead of explicit RGB values. In this case, an application using the bitmap must call DIB functions (*CreateDIBitmap*, *CreateDIBPatternBrush*, and *CreateDIBSection*) with the *wUsage* parameter set to DIB_PAL_COLORS.

Note The *bmiColors* member should not contain palette indexes if the bitmap is to be stored in a file or transferred to another application. Unless the application uses the bitmap exclusively and under its complete control, the bitmap color table should contain explicit RGB values.

BITMAPINFOHEADER struct

windows.h

```
typedef struct tagBITMAPINFOHEADER {
 DWORD biSize;
 LONG biWidth;
 LONG biHeight;
```

```
WORD biPlanes;
WORD biBitCount;
DWORD biCompression;
DWORD biSizeImage;
LONG biXPelsPerMeter;
LONG biYPelsPerMeter;
DWORD biClrUsed;
DWORD biClrImportant;
} BITMAPINFOHEADER;
```

Contains dimension and color-format information for device-independent bitmaps (DIB).

Member	Description
biSize	Number of bytes required by this structure.
biWidth	Width of the bitmap, in pixels.
biHeight	Height of the bitmap, in pixels. If *biHeight* is negative, the bitmap origin is the upper-left corner and the height is the absolute value of *biHeight*.
biPlanes	Specifies the number of planes for the target device. This member must be set to 1.
biBitCount	Specifies the number of bits per pixel. This value must be 1, 4, 8, 16 (16-bit only), 24, or 32 (32-bit only).
biCompression	Specifies the type of compression for a compressed bitmap. It can be one of the following values:
	Value — **Meaning**
	BI_RGB — Uncompressed.
	BI_RLE8 — Specifies a run-length encoded format for bitmaps with 8 bits per pixel.
	BI_RLE4 — Specifies a run-length encoded format for bitmaps with 4 bits per pixel.
	BI_BITFIELDS — Specifies the bitmap is not compressed and the color table consists of 3 DWORD color masks which specify the red, green, and blue components, respectively, of each pixel. This is valid when used with 16 and 32 bits per pixel bitmaps.
biSizeImage	Size of the image, in bytes. Can be 0 if the bitmap is in the BI_RGB format.
biXPelsPerMeter	Specifies the horizontal resolution, in pixels per meter, of the target device for the bitmap. Use this value to select a bitmap from a resource group that best matches the characteristics of the current device.
biYPelsPerMeter	Specifies the vertical resolution, in pixels per meter, of the target device.

Member	Description
biClrUsed	Size of the image, in bytes. Can be 0 if the bitmap is in the BI_RGB format.
	Specifies the horizontal resolution, in pixels per meter, of the target device for the bitmap. Use this value to select a bitmap from a resource group that best matches the characteristics of the current device.
	Specifies the vertical resolution, in pixels per meter, of the target device.
	Specifies the number of color indexes in the color table actually used by the bitmap. If this value is 0, the bitmap uses the maximum number of colors corresponding to the value of the biBitCount member.
	For 16-bit applications, if *biClrUsed* is nozero, it specifies the actual number of colors that the graphics engine or device driver will access if the *biBitCount* member is less than 24. If *biBitCount* is set to 24, *biClrUsed* specifies the size of the reference color table used to optimize performance of color palettes.
	For 32-bit applications, if *biClrUsed* is nonzero and the *biBitCount* member is less than 16, the *biClrUsed* member specifies the actual number of colors the graphics engine or device driver accesses. If *biBitCount* is 16 or greater, then *biClrUsed* member specifies the size of the color table used to optimize performance of Windows color palettes. For *biBitCount* equal to 16 or 32 the optimal color palette starts immediately following the 3 DWORD masks.
	If the bitmap is a packed bitmap (that is, a bitmap in which the bitmap array immediately follows the *BITMAPINFO* header and which is referenced by a single pointer), the *biClrUsed* member must be set to zero or to the actual size of the color table.
biClrImportant	Specifies the number of color indexes that are considered important for displaying the bitmap. If this value is zero, all colors are important.

COLORREF typedef

windows.h

```
typedef DWORD COLORREF;
```

A 32-bit value used to specify an RGB color. The *COLORREF* value has the following hexadecimal form:

```
0x00bbggrr
```

The low-order byte (*rr*contains a value for the relative intensity of red; the second byte (*gg*contains a value for green; and the third byte (*bb*contains a value for blue. The fourth byte byte must be zero. The RGB macro can be used to set these values:

```
RGB(red, green, blue)
```

Each color parameter can range from 0x0 to 0xFF.

COMPAREITEMSTRUCT struct

windows.h

```
typedef struct tagCOMPAREITEMSTRUCT {
 UINT  CtlType;
 UINT  CtlID;
 HWND  hwndItem;
 UINT  itemID1;
 DWORD itemData1;
 UINT  itemID2;
```

```
 DWORD itemData2;
} COMPAREITEMSTRUCT;
```

Supplies the identifiers and application-supplied data for two items in a sorted owner-drawn list box or combo box. The table below describes the members.

Member	Description
CtlType	Either ODT_LISTBOX for an owner-drawn list box, or ODT_COMBOBOX for an owner-drawn combo box.
CtlID	List box or combo box identifier.
hwndItem	Control identifier.
itemID1	Index of the first item in the list box or combo box being compared.
itemData1	Application-supplied data for the first item being compared.
itemID2	Index of the second item in the list box or combo box being compared.
itemData2	Application-supplied data for the second item being compared.

DELETEITEMSTRUCT struct

windows.h

```
typedef struct tagDELETEITEMSTRUCT {
 UINT CtlType;
 UINT CtlID;
 UINT itemID;
 HWND hwndItem;
 UINT itemData;
} DELETEITEMSTRUCT;
```

Describes a deleted owner-drawn list-box or combo-box item. The table below describes the members.

Member	Description
CtlType	Either ODT_LISTBOX for an owner-drawn list box, or ODT_COMBOBOX for an owner-drawn combo box.
CtlID	List box or combo box identifier.
itemID	Index of the item in the list box or combo box being removed.
hwndItem	Control indentifier.
itemData	Application-defined item data.

DEVMODE struct

windows.h

16-bit version:

```
typedef struct tagDEVMODE {
 char  dmDeviceName[CCHDEVICENAME];
 UINT  dmSpecVersion;
 UINT  dmDriverVersion;
 UINT  dmSize;
 UINT  dmDriverExtra;
 DWORD dmFields;
```

```
 int  dmOrientation;
 int  dmPaperSize;
 int  dmPaperLength;
 int  dmPaperWidth;
 int  dmScale;
 int  dmCopies;
 int  dmDefaultSource;
 int  dmPrintQuality;
 int  dmColor;
 int  dmDuplex;
 int  dmYResolution;
 int  dmTTOption;
} DEVMODE;
```

32-bit version:

```
typedef struct _devicemode {
 TCHAR  dmDeviceName[32];
 WORD   dmSpecVersion;
 WORD   dmDriverVersion;
 WORD   dmSize;
 WORD   dmDriverExtra;
 DWORD  dmFields;
 short  dmOrientation;
 short  dmPaperSize;
 short  dmPaperLength;
 short  dmPaperWidth;
 short  dmScale;
 short  dmCopies;
 short  dmDefaultSource;
 short  dmPrintQuality;
 short  dmColor;
 short  dmDuplex;
 short  dmYResolution;
 short  dmTTOption;
 short  dmCollate;
 TCHAR  dmFormName[32];
 WORD   dmUnusedPadding;
 USHORT dmBitsPerPel;
 DWORD  dmPelsWidth;
 DWORD  dmPelsHeight;
 DWORD  dmDisplayFlags;
 DWORD  dmDisplayFrequency;
} DEVMODE;
```

Contains printer-driver initialization and environment data. The following table describes each member.

Member	Description
dmDeviceName	Name of the device the driver supports—for example, PCL/Laserjet: in the case of the Hewlett-Packard Laserjet.
dmSpecVersion	Version number of this structure.
dmDriverVersion	Printer driver version number.
dmSize	Size of this structure, in bytes,excluding *dmDriverData*. Use to find the length of the structure without having to account for different versions.
dmDriverExtra	Size of the optional *dmDriverData* member, in bytes. If an application does not use device-specific information, it should set this member to zero.
dmFields	Contains flags that indicate which of the remaining members in this structure have been initialized. It can be any combination of the following values: DM_ORIENTATION — DM_DEFAULTSOURCE DM_PAPERSIZE — DM_PRINTQUALITY DM_PAPERLENGTH — DM_COLOR DM_PAPERWIDTH — DM_DUPLEX DM_SCALE — DM_YRESOLUTION DM_COPIES — DM_TTOPTION
dmOrientation	Paper orientation. Can be DMORIENT_PORTRAIT or DMORIENT_LANDSCAPE.
dmPaperSize	Paper size. Can be set to zero if the length and width of the paper are specified by *dmPaperLength* and *dmPaperWidth*. Otherwise, set to one of the following values: **Value** — **Meaning** DMPAPER_FIRST — DMPAPER_LETTER DMPAPER_LETTER — Letter, 8 1/2 x 11 in. DMPAPER_LETTERSMALL — Letter Small, 8 1/2 x 11 in. DMPAPER_TABLOID — Tabloid, 11 x 17 in. DMPAPER_LEDGER — Ledger, 17 x 11 in. DMPAPER_LEGAL — Legal, 8 1/2 x 14 in. DMPAPER_STATEMENT — Statement, 5 1/2 x 8 1/2 in. DMPAPER_EXECUTIVE — Executive, 7 1/2 x 10 1/2 in. DMPAPER_A3 — A3, 297 x 420 mm DMPAPER_A4 — A4, 210 x 297 mm DMPAPER_A4SMALL — A4 Small, 210 x 297 mm DMPAPER_A5 — A5, 148 x 210 mm DMPAPER_B4 — B4, 250 x 354 mm DMPAPER_B5 — B5, 182 x 257 mm DMPAPER_FOLIO — Folio, 8 1/2 x 13 in. DMPAPER_QUARTO — Quarto, 215 x 275 mm DMPAPER_10X14 — 10 x 14 in. DMPAPER_11X17 — 11 x 17 in. DMPAPER_NOTE — Note, 8 1/2 x 11 in. DMPAPER_ENV_9 — Envelope #9, 3 7/8 x 8 7/8 in. DMPAPER_ENV_10 — Envelope #10, 4 1/8 x 9 1/2 in.

Member	Description	
	DMPAPER_ENV_11	Envelope #11, 4 1/2 x 10 3/8 in.
	DMPAPER_ENV_12	Envelope #12, 4 1/2 x 11 in.
	DMPAPER_ENV_14	Envelope #14, 5 x 11 1/2 in.
	DMPAPER_CSHEET	C size sheet
	DMPAPER_DSHEET	D size sheet
	DMPAPER_ESHEET	E size sheet
	DMPAPER_ENV_DL	Envelope DL, 110 x 220 mm
	DMPAPER_ENV_C3	Envelope C3, 324 x 458 mm
	DMPAPER_ENV_C4	Envelope C4, 229 x 324 mm
	DMPAPER_ENV_C5	Envelope C5, 162 x 229 mm
	DMPAPER_ENV_C6	Envelope C6, 114 x 162 mm
	DMPAPER_ENV_C65	Envelope C65, 114 x 229 mm
	DMPAPER_ENV_B4	Envelope B4, 250 x 353 mm
	DMPAPER_ENV_B5	Envelope B5, 176 x 250 mm
	DMPAPER_ENV_B6	Envelope B6, 176 x 125 mm
	DMPAPER_ENV_ITALY	Envelope, 110 x 230 mm
	DMPAPER_ENV_MONARCH	Envelope Monarch, 3 7/8 x 7 1/2 in.
	DMPAPER_ENV_PERSONAL	Envelope, 3 5/8 x 6 1/2 in.
	DMPAPER_FANFOLD_US	U.S. Standard Fanfold, 14 7/8 x 11 in.
	DMPAPER_FANFOLD_STD_GERMAN	German Standard Fanfold, 8 1/2 x 12 in.
	DMPAPER_FANFOLD_LGL_GERMAN	German Legal Fanfold, 8 1/2 x 13 in.
	DMPAPER_LAST	German Legal Fanfold, 8 1/2 x 13 in.
	DMPAPER_USER	User-defined
dmPaperLength	Paper length, in tenths of a millimeter. Overrides *dmPaperSize*.	
dmPaperWidth	Paper width, in tenths of a millimeter. Overrides *dmPaperSize*.	
dmScale	Print output scale factor. Apparent page size is scaled from the physical page size by dmScale/100.	
dmCopies	Specifies the number of copies printed if the device supports multiple-page copies.	
dmDefaultSource	Default paper feed bin. This member can be one of the following values:	
	DMBIN_AUTO	DMBIN_LOWER
	DMBIN_CASSETTE	DMBIN_MANUAL
	DMBIN_ENVELOPE	DMBIN_MIDDLE
	DMBIN_ENVMANUAL	DMBIN_ONLYONE
	DMBIN_FIRST	DMBIN_SMALLFMT
	DMBIN_LARGECAPACITY	DMBIN_TRACTOR
	DMBIN_LARGEFMT	DMBIN_UPPER
	DMBIN_LAST	
mPrintQuality	Printer resolution. Can be one of the following values:	
	DMRES_HIGH	
	DMRES_MEDIUM	
	DMRES_LOW	

Member	Description	
	DMRES_DRAFT	
	If another (positive) value is given, it specifies dots per inch (DPI) and is device-dependent. If the printer initializes *dmYResolution*, then *dmPrintQuality* specifies the x-resolution of the printer in DPI.	
dmColor	Color or monochrome output. Can be DMCOLOR_COLOR, or DMCOLOR_MONOCHROME.	
dmDuplex	Duplex (double-sided) printing for printers capable of duplex printing. This member can be one of the following values:	
	DMDUP_SIMPLEX	
	DMDUP_HORIZONTAL	
	DMDUP_VERTICAL	
dmYResolution	Y-resolution of the printer in DPI. If the printer initializes *dmYResolution*, *dmPrintQuality* specifies the x-resolution of the printer, in DPI.	
dmTTOption	TrueType-font printing. Can be one of the following values:	
	Value	**Meaning**
	DMTT_BITMAP	Print TrueType fonts as graphics. The default for dot-matrix printers.
	DMTT_DOWNLOAD	Download TrueType fonts as soft fonts. Default for Hewlett-Packard printers that use Printer Control Language (PCL).
	DMTT_SUBDEV	Substitute device fonts for TrueType fonts. Default for PostScript printers.
dmUnusedPadding	(32-bit only) Used to align the structure on a DWORD boundary.	
dmCollate	Collation when printing multiple copies. Can be be one of the following values:	
	DMCOLLATE_TRUE	Collate when printing multiple copies.
	DMCOLLATE_FALSE	Do NOT collate when printing multiple copies.
dmFormName	(32-bit only) Specifies the name of the form to use; for example, "Letter" or "Legal".	
dmBitsPerPel	(32-bit only) Color resolution of the display device, in bits per pixel.	
dmPelsWidth	(32-bit only) Width of the visible device surface, in pixels.	
dmPelsHeight	(32-bit only) Height of the visible device surface, in pixels.	
dmDisplayFlags	(32-bit only) Device display mode. Can be one of the following values:	
	Value	**Meaning**
	DM_GRAYSCALE	Non-color display. If this flag is not set, color is assumed.
	DM_INTERLACED	Interlaced display mode. If the flag is not set, non-interlaced is assumed.
dmDisplayFrequency	(32-bit only) Specifies the display device frequency, in hertz.	

DRAWITEMSTRUCT struct

windows.h

```
typedef struct tagDRAWITEMSTRUCT {
 UINT CtlType;
 UINT CtlID;
 UINT itemID;
 UINT itemAction;
```

```
UINT itemState;
HWND hwndItem;
HDC hDC;
RECT rcItem;
DWORD itemData;
} DRAWITEMSTRUCT;
```

Provides information the owner window must have to determine how to paint an owner-drawn control or menu item. The owner window of the owner-drawn control or menu item receives a pointer to this structure as the *lParam* parameter of the WM_DRAWITEM message.

Member	Description
CtlType	Specifies the control type. This member can be one of the following values: ODT_BUTTON — Owner-drawn button ODT_COMBOBOX — Owner-drawn combo box ODT_LISTBOX — Owner-drawn list box ODT_MENU — Owner-drawn menu item
CtlID	Specifies the identifier of the combo box, list box, or button. This member is not used for a menu item.
itemID	Specifies the menu item identifier for a menu item or the index of the item in a list box or combo box. For an empty list box or combo box, this member can be –1. This allows the application to draw only the focus rectangle at the coordinates specified by the *rcItem* member even though there are no items in the control. This indicates to the user whether the list box or combo box has the focus. How the bits are set in the *itemAction* member determines whether the rectangle is to be drawn as though the list box or combo box has the focus.
itemAction	Specifies the drawing action required. This member can be one or more of the following values: **Value** — **Meaning** ODA_DRAWENTIRE — The entire control needs to be drawn. ODA_FOCUS — The control has lost or gained the keyboard focus. The *itemState* member should be checked to determine whether the control has the focus. ODA_SELECT — The selection status has changed. The *itemState* member should be checked to determine the new selection state.
itemState	Specifies the visual state of the item *after* the current drawing action takes place. This member can be a combination of the following values: **Value** — **Meaning** ODS_CHECKED — The menu item is to be checked. This bit is used only in a menu. ODS_DISABLED — The item is to be drawn as disabled. ODS_FOCUS — The item has the keyboard focus. ODS_GRAYED — The item is to be grayed. This bit is used only in a menu. ODS_SELECTED — The menu item's status is selected.
hwndItem	Identifies the control for combo boxes, list boxes, and buttons. For menus, this member identifies the menu containing the item.
hDC	Identifies a device context (DC); this DC must be used when performing drawing operations on the control.

Member	Description
rcItem	Specifies a rectangle that defines the boundaries of the control to be drawn. This rectangle is in the DC specified by the *hDC* member. Anything the owner window draws in the DC for combo boxes, list boxes, and buttons is automatically clipped, except for menu items. When drawing menu items, the owner window must not draw outside the boundaries of the rectangle defined by the *rcItem* member.
itemData	Specifies the application-defined 32-bit value associated with the menu item. For a control, this parameter specifies the value last assigned to the list box or combo box by the LB_SETITEMDATA or CB_SETITEMDATA message. If the list box or combo box has the LBS_HASSTRINGS or CBS_HASSTRINGS style, this value is initially zero. Otherwise, this value is initially the value that was passed to the list box or combo box in the *lParam* parameter of one of these messages: CB_ADDSTRING, CB_INSERTSTRING, LB_ADDSTRING, LB_INSERTSTRING.

FINDREPLACE struct

windows.h

```
typedef struct {
 DWORD        lStructSize;
 HWND         hwndOwner;
 HINSTANCE    hInstance;
 DWORD        Flags;
 LPSTR        lpstrFindWhat;
 LPSTR        lpstrReplaceWith;
 WORD         wFindWhatLen;
 WORD         wReplaceWithLen;
 DWORD        lCustData;
 LPFRHOOKPROC lpfnHook;
 LPCSTR       lpTemplateName;
} FINDREPLACE;
```

Contains initialization information to initialize the system-defined find-and-replace dialog boxes. The table below describes the members.

Member	Description	
lStructSize	Length, in bytes, of the structure.	
hwndOwner	Window that owns the dialog box. Can be any valid window handle, but must not be NULL.	
hInstance	Data block that contains a dialog box template specified by *lpstrTemplateName*. Used only if the *Flags* member is set to FR_ENABLETEMPLATE; otherwise ignored.	
Flags	Dialog box initialization flags. Can be a combination of the following values:	
	Value	**Meaning**
	FR_DIALOGTERM	Indicates that the dialog box is closing.
	FR_DOWN	If this flag is set, the search direction is down; if the flag is clear, the search direction is up.
	FR_ENABLEHOOK	Enables the hook function specified in the *lpfnHook* member of this structure. Used to initialize the dialog box only.

Member	Description	
	FR_ENABLETEMPLATE	Create the dialog box by using the dialog box template identified by the *hInstance* and *lpTemplateName* members. This flag is used to initialize the dialog box only.
	FR_ENABLETEMPLATEHANDLE	Indicates that the *hInstance* member identifies a data block that contains a preloaded dialog box template. The *lpTemplateName* member is ignored if this flag is specified.
	FR_FINDNEXT	The application should search for the next occurrence of the string specified by the *lpstrFindWhat* member.
	FR_HIDEUPDOWN	Hides the Direction check box and the Up and Down controls.
	FR_HIDEMATCHCASE	Hides the Match Case check box.
	FR_HIDEWHOLEWORD	Hides the Match Whole Word Only check box.
	FR_MATCHCASE	Case-sensitive searches.
	FR_NOMATCHCASE	Disables the Match Case check box.
	FR_NOUPDOWN	Disables the direction radio buttons.
	FR_NOWHOLEWORD	Disables the Whole Word check box.
	FR_REPLACE	Replace the current occurrence of the string specified by lpstrFindWhat with the string specified by lpstrReplaceWith.
	FR_REPLACEALL	Replace all occurrences of the string specified by *lpstrFindWhat* member with the string specified by *lpstrReplaceWith*.
	FR_SHOWHELP	Show the Help button. The *hwndOwner* member must not be NULL if this option is specified.
	FR_WHOLEWORD	Checks the Whole Word check box. Only whole words matching the search string will be considered.
lpstrFindWhat	String to search for. If there is a string specified when the dialog box starts, the dialog box initializes the Find What: text control with this string. If the FR_FINDNEXT flag is set when the dialog box is opened, the application should search for an occurrence of this string by using the FR_DOWN, FR_WHOLEWORD, and FR_MATCHCASE flags to further define the direction and type of search. The application must allocate a buffer for the string, which should be at least 80 characters long.	
lpstrReplaceWith	Replacement string for replace operations.	
wFindWhatLen	Length, in bytes, of the buffer pointed to by *lpstrFindWhat*.	
wReplaceWithLen	Length, in bytes, of the buffer pointed to by lpstrReplaceWith.	
lCustData	Application-defined data the passed to the hook function identified *lpfnHook*.	
lpfnHook	Points to a hook function that processes messages intended for the dialog box. An application must specify the FR_ENABLEHOOK flag in the *Flags* member to enable the function; otherwise, the system ignores this structure member. Should return FALSE to pass a message on to the standard dialog procedure, or TRUE to discard the message.	
lpstrTemplateName	Points to a null-terminated string that names the dialog box template resource to be substituted for the standard dialog box template.	

GLYPHMETRICS struct

windows.h

```
typedef struct _GLYPHMETRICS {
 UINT gmBlackBoxX;
 UINT gmBlackBoxY;
 POINT gmptGlyphOrigin;
 short gmCellIncX;
 short gmCellIncY;
} GLYPHMETRICS;
```

Contains information about the placement and orientation of a glyph in a character cell, specified in logical units.

Member	Description
gmBlackBoxX	Width of the smallest rectangle that completely encloses the glyph.
gmBlackBoxY	Height of the smallest rectangle that completely encloses the glyph.
gmptGlyphOrigin	The x- and y-coordinates of the upper left corner of the smallest rectangle that completely encloses the glyph.
gmCellIncX	The horizontal distance from the origin of the current character cell to the origin of the next character cell.
gmCellIncY	The vertical distance from the origin of the current character cell to the origin of the next character cell.

HANDLETABLE struct

windows.h

```
typedef struct tagHANDLETABLE {
 HGDIOBJ objectHandle[1];
} HANDLETABLE;
```

An array of handles that identify a graphics device interface (GDI) object.

ICONINFO struct

windows.h

```
typedef struct _ICONINFO {
  BOOL   fIcon;
  DWORD  xHotspot;
  DWORD  yHotspot;
  HBITMAP hbmMask;
  HBITMAP hbmColor;
} ICONINFO;
```

The *ICONINFO* structure contains information about an icon or a cursor. The following table describes the members.

Member	Description
fIcon	Set to TRUE if this specifies an icon; FALSE if it specifies a cursor.
xHotspot	Specifies the x-coordinate of a cursor's hot spot. For icons the hot spot is always in the center, so this member is ignored.

Member	Description
yHotspot	Specifies the y-coordinate of the cursor's hot spot. For icons the hot spot is always in the center, so this member is ignored.
hbmMask	Specifies the icon bitmask bitmap. For black and white icons the upper half of *hbmMask* is the icon AND bitmask, the lower half is the icon XOR bitmask, and the height should be an even multiple of two. For color icons this mask defines the AND bitmask of the icon only.
hbmColor	Identifies the icon color bitmap. This member can be optional if this structure defines a black and white icon. After the AND bitmask of *hbmMask* is applied to the destination, the color bitmap is applied (using XOR) to the destination.

KERNINGPAIR struct

windows.h

```
typedef struct tagKERNINGPAIR {
   WORD wFirst;
   WORD wSecond;
   int  iKernAmount;
} KERNINGPAIR;
```

Defines a kerning pair. The following table describes the members.

Member	Description
wFirst	Character code for the first character in the kerning pair.
wSecond	Character code for the second character in the kerning pair.
iKernAmount	Amount this pair will be kerned if they appear side by side in the same font and size. This value is usually negative, and is given in logical units (depending on mapping mode).

LOGBRUSH struct

windows.h

```
typedef struct tagLOGBRUSH {
 UINT lbStyle;
 COLORREF lbColor;
 int lbHatch;
} LOGBRUSH;
```

Defines the style, color, and pattern of a physical brush. The following table describes the members:

Member	Description	
lbStyle	Specifies the brush style. This member can be one of the following values:	
	Value	**Meaning**
	BS_DIBPATTERN	Pattern brush defined by a device-independent bitmap (DIB) specification.
	BS_DIBPATTERNPT	A pattern brush defined by a device-independent bitmap (DIB) specification (32-bit only).
	BS_HATCHED	Hatched brush.
	BS_HOLLOW	Hollow brush.
	BS_PATTERN	Pattern brush defined by a memory bitmap.

Member	Description	
	BS_NULL	Equivalent to BS_HOLLOW.
	BS_SOLID	Solid brush.
lbColor	Specifies the color to draw the brush. If the *lbStyle* member is the BS_HOLLOW or BS_PATTERN value, *lbColor* is ignored.	
	If *lpStyle* is the BS_DIBPATTERN or BS_DIBPATTERNPT value, the low-order word of *lbColor* specifies whether the *bmiColors* members of the *BITMAPINFO* structure contain explicit RGB values or indexes into the currently realized logical palette. The *lbColor* member must be one of the following values:	
	Value	**Meaning**
	DIB_PAL_COLORS	Color table consists of an array of 16-bit indexes into the currently realized logical palette.
	DIB_RGB_COLORS	Color table contains literal RGB values.
lbHatch	Specifies a hatch style. The meaning depends on the brush style. If the *lbStyle* member is the BS_DIBPATTERN style, the *lbHatch* member contains a handle to a packed DIB. If the *lbStyle* member is the BS_DIBPATTERNPT style, the *lbHatch* member contains a pointer to a packed DIB.	
	If the *lbStyle* member is the BS_HATCHED style, the *lbHatch* member specifies the orientation of the lines used to create the hatch. This member can be one of the following values:	
	Value	**Meaning**
	HS_BDIAGONAL	45-degree upward hatch (left to right)
	HS_CROSS	Horizontal and vertical cross-hatch
	HS_DIAGCROSS	45-degree cross-hatch
	HS_FDIAGONAL	45-degree downward hatch (left to right)
	HS_HORIZONTAL	Horizontal hatch
	HS_VERTICAL	Vertical hatch
	If the *lbStyle*member is the BS_PATTERN style, *lbHatch* must be a handle to the bitmap that defines the pattern. If the *lbStyle* member is the BS_SOLID or the BS_HOLLOW style, *lbHatch* is ignored.	

See also BITMAPINFO struct

LOGFONT struct

windows.h

```
typedef struct tagLOGFONT {
 int   lfHeight;
 int   lfWidth;
 int   lfEscapement;
 int   lfOrientation;
 int   lfWeight;
 BYTE  lfItalic;
 BYTE  lfUnderline;
 BYTE  lfStrikeOut;
 BYTE  lfCharSet;
 BYTE  lfOutPrecision;
 BYTE  lfClipPrecision;
 BYTE  lfQuality;
 BYTE  lfPitchAndFamily;
```

```
 BYTE  lfFaceName[LF_FACESIZE];
} LOGFONT;
```

32 bit version:

```
typedef struct tagLOGFONT { /* lf */
 LONG lfHeight;
 LONG lfWidth;
 LONG lfEscapement;
 LONG lfOrientation;
 LONG lfWeight;
 BYTE lfItalic;
 BYTE lfUnderline;
 BYTE lfStrikeOut;
 BYTE lfCharSet;
 BYTE lfOutPrecision;
 BYTE lfClipPrecision;
 BYTE lfQuality;
 BYTE lfPitchAndFamily;
 CHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

Defines font attributes. When creating a logical font, applications can use the default settings for most of the members. Unless *lfHeight* and *lfFaceName* are given specific values, the created logical font will be device dependent. The following table describes the members.

Member	Description
lfHeight	(16-bit applications) Font height, in logical units. If greater than zero, it specifies the font cell height. If less than zero, it specifies the character height of the font (applications that specify font height in points typically use a negative number for this member). If 0, the font mapper uses a default height. The font mapper chooses the largest physical font not exceeding the requested size,or the smallest font if the fonts exceed the requested size. The absolute value of the *lfHeight* member must not exceed 16,384 after it is converted to device units. (32-bit applictions) Font height, in logical units. The font height can be specified in one of three ways. If *lfHeight* is greater than zero, it is transformed into device units and matched against the cell height of the available fonts. If it is zero, a reasonable default size is used. If it is less than zero, it is transformed into device units and the absolute value is matched against the character height of the available fonts. For all height comparisons, the font mapper looks for the largest font that does not exceed the requested size; if there is no such font, it looks for the smallest font available. This mapping occurs when the font is used for the first time.
lfWidth	Average font character width, in logical units, of font characters in logical units. If 0, the font mapper chooses a default width.
lfEscapement	Angle between the base line of a character and the x-axis, in tenths of degrees.
lfOrientation	Angle of character base line, tenths of degrees and relative to the bottom of the page.
lfWeight	Font weight. Can be one of the following values: FW_DONTCARE FW_SEMIBOLD FW_THIN FW_DEMIBOLD FW_EXTRALIGHT FW_BOLD FW_ULTRALIGHT FW_BOLD FW_LIGHT FW_ULTRABOLD

Member	Description
	FW_NORMAL FW_BLACK FW_REGULAR FW_HEAVY FW_MEDIUM If the value is 0, a default weight is used.
lfItalic	Italic font if TRUE.
lfUnderline	Underlined font if TRUE.
lfStrikeOut	Strikeout font if TRUE.
lfCharSet	Font character set. Can be one of the following values: ANSI_CHARSET DEFAULT_CHARSET (16 bit) SYMBOL_CHARSET SHIFTJIS_CHARSET (16 bit) OEM_CHARSET UNICODE_CHARSET (32 bit)
lfOutPrecision	Desired output precision. Specifies how closely the output must match the height, width, character orientation, escapement, and pitch of the requested font. Can be one of the following values: OUT_CHARACTER_PRECIS OUT_STRING_PRECIS OUT_DEFAULT_PRECIS OUT_STROKE_PRECIS OUT_DEVICE_PRECIS OUT_TT_PRECIS OUT_RASTER_PRECIS OUT_TT_ONLY_PRECIS
lfClipPrecision	Desired clipping precision. Specifies how to clip characters partially outside the clipping region. Can be any one of the following values: CLIP_CHARACTER_PRECIS CLIP_MASK CLIP_DEFAULT_PRECIS CLIP_STROKE_PRECIS CLIP_EMBEDDED CLIP_TT_ALWAYS CLIP_LH_ANGLES To use an embedded read-only font, applications must specify the CLIP_EMBEDDED value. To achieve consistent rotation of device, TrueType, and vector fonts, an application can use the OR operator to combine the CLIP_LH_ANGLES value with any of the other *lfClipPrecision* values. If the CLIP_LH_ANGLES bit is set, the rotation for all fonts is dependent on whether the orientation of the coordinate system is left-handed or right-handed. If CLIP_LH_ANGLES is not set, device fonts always rotate counter-clockwise, but the rotation of other fonts is dependent on the orientation of the coordinate system. (For more information about the orientation of coordinate systems, see the description of the *lfEscapement* member.)
lfQuality	Output font quality. Can be one of the following values: **Value** **Meaning** DEFAULT_QUALITY Font appearance does not matter. DRAFT_QUALITY Font appearance is less important than when the PROOF_QUALITY value is used. PROOF_QUALITY Font character quality is more important than matching logical-font attributes.
lfPitchAndFamily	Font family and pitch. The two low-order bits specify the font pitch and can be one of the following values: DEFAULT_PITCH

Member	Description
	FIXED_PITCH VARIABLE_PITCH The four high-order bits of the member specify the font family and can be one of the following values:

Value	Meaning
FF_DECORATIVE	Novelty fonts.
FF_DONTCARE	Don't care or don't know.
FF_MODERN	Fonts with constant stroke width, with or without serifs (e.g. Pica and Courier).
FF_ROMAN	Fonts with variable stroke width and with serifs (e.g. Times New Roman).
FF_SCRIPT	Fonts designed to look like handwriting (e.g. Script and Cursive).
FF_SWISS	Fonts with variable stroke width and without serifs (e.g. Sans Serif).

Use the Boolean OR operator to match a pitch constant with a family constant.

lfFaceName	Specifies the typeface name of the font. The length of this string must not exceed LF_FACESIZE – 1.

LOGPALETTE struct

windows.h

```
typedef struct tagLOGPALETTE {
 WORD palVersion;
 WORD palNumEntries;
 PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
```

Defines a logical color palette. Colors in the palette-entry table should appear in order of importance because entries earlier in the logical palette are more likely to be placed in the system palette. The following table describes the members:

Member	Description
palVersion	Specifies the Windows version number for the structure (currently 0x300).
palNumEntries	Specifies the number of entries in the logical color palette.
palPalEntry	Specifies an array of *PALETTEENTRY* structures that define the color and usage of each entry in the logical palette.

LOGPEN struct

windows.h

```
typedef struct tagLOGPEN {
 UINT lopnStyle;
 POINT lopnWidth;
 COLORREF lopnColor;
} LOGPEN;
```

Defines the style, width, and color of a pen.

Member	Description	
lopnStyle	Specifies the pen type. This member can be one of the following values:	
	Value	**Meaning**
	PS_SOLID	Solid pen
	PS_DASH	Dashed pen (pen width = 1)
	PS_DOT	Dotted pen (pen width = 1)
	PS_DASHDOT	Alternating dashes and dots (pen width = 1)
	PS_DASHDOTDOT	Alternating dashes and double dots (pen width = 1)
	PS_NULL	Invisible pen
	PS_INSIDEFRAME	Creates a pen that draws a line inside the frame of closed shapes produced by graphics device interface (GDI) output functions that specify a bounding rectangle. When this style is used with GDI output functions.
lopnWidth	Specifies the pen width, in logical units. If the *lopnWidth* member is zero, the pen is one pixel wide on raster devices regardless of the current mapping mode.	
lopnColor	Specifies the pen color.	

MDICREATESTRUCT struct

```
typedef struct tagMDICREATESTRUCT {
 LPCSTR   szClass;
 LPCSTR   szTitle;
 HANDLE hOwner;
 int      x;
 int      y;
 int      cx;
 int      cy;
 DWORD    style;
 LPARAM   lParam;
} MDICREATESTRUCT;
```

Contains class, title, location, owner, and size of a Multiple Document Interface (MDI) child window. The table below describes the members.

Member	Description
szClass	Points to a null-terminated string specifying the registered MDI child window name.
szTitle	Points to a null-terminated string specifying the registered MDI child window title.
hOwner	Instance handle for the application creating the MDI child window.
x	Initial position of the left side of the MDI child window. If set to CW_USEDEFAULT, the MDI child window is assigned a horizontal position.
y	Initial position of the MDI child window top edge. If set to CW_USEDEFAULT, the MDI child window is assigned a default vertical position.
cx	Initial width of the MDI child window. If set to CW_USEDEFAULT, the MDI child window is assigned a default width.
cy	Initial height of the MDI child window. If set to CW_USEDEFAULT, the MDI child window is assigned a default height.

Member	Description
style	Specifies additional styles for the MDI child window. If the MDI client window was created with the MDIS_ALLCHILDSTYLES window style, this member can be any combination of the window styles and creation attributes, including those listed in the following table:

Value	Meaning
WS_MINIMIZE	Created in a minimized state.
WS_MAXIMIZE	Created in a maximized state.
WS_HSCROLL	Created with a horizontal scroll bar.
WS_VSCROLL	Created with a vertical scroll bar.

lParam	An application-defined 32-bit value.

METARECORD struct

windows.h

```
typedef struct tagMETARECORD {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[1];
} METARECORD;
```

Contains a metafile record. The table below describes the members.

Member	Description
rdSize	Size of the record, in words.
rdFunction	Function number.
rdParm	Function parameters in reverse of the order they are passed to the function.

MEASUREITEMSTRUCT struct

windows.h

```
typedef struct tagMEASUREITEMSTRUCT {
 UINT CtlType;
 UINT CtlID;
 UINT itemID;
 UINT itemWidth;
 UINT itemHeight;
 DWORD itemData;
} MEASUREITEMSTRUCT;
```

Contains dimensions of an owner-drawn control. The table below describes the members.

Member	Description
CtlType	Specifies the control type and can be one of the following values:

ODT_BUTTON	Owner-drawn button
ODT_COMBOBOX	Owner-drawn combo box
ODT_LISTBOX	Owner-drawn list box
ODT_MENU	Owner-drawn menu

Member	Description
CtlID	Identifier of the combo box, list box, or button (not used for menus).
itemID	Menu item identifier for a menu item, or the list box item identifier for a variable-height list or combo box (not used for a fixed-height list box, combo box, or for a button).
itemWidth	Width of a menu item, in pixels.
itemHeight	Height of an individual item in a list box or a menu, in pixels.
itemData	Application-defined 32-bit value associated with a menu item. For a control, this parameter specifies the value last assigned to the list box or combo box by the LB_SETITEMDATA or CB_SETITEMDATA message. If the list box or combo box has the LB_HASSTRINGS or CB_HASSTRINGS style, this value is initially zero. Otherwise, this value is initially the value passed to the list box or combo box in the *lParam* parameter of one of the following messages: CB_ADDSTRING CB_INSERTSTRING LB_ADDSTRING LB_INSERTSTRING

MSG struct

windows.h

```
typedef struct tagMSG {
 HWND hwnd;
 UINT message;
 WPARAM wParam;
 LPARAM lParam;
 DWORD time;
 POINT pt;
} MSG;
```

The *MSG* structure contains the following message information from a message queue:

Member	Description
hwnd	Handle for the window whose window procedure receives the message.
message	Message number.
wParam	Additional message information. The meaning depends on the value of *message*.
lParam	Additional message information. The meaning depends on the value of *message*.
time	The time the message was posted.
pt	The cursor position when the message was posted, in screen coordinates.

OUTLINETEXTMETRIC struct

windows.h

```
typedef struct _OUTLINETEXTMETRIC {
 UINT   otmSize;
 TEXTMETRIC otmTextMetrics;
 BYTE   otmFiller;
 PANOSE otmPanoseNumber;
 UINT   otmfsSelection;
```

```
UINT   otmfsType;
int    otmsCharSlopeRise;
int    otmsCharSlopeRun;
int    otmItalicAngle;
UINT   otmEMSquare;
int    otmAscent;
int    otmDescent;
UINT   otmLineGap;
UINT   otmsCapEmHeight;
UINT   otmsXHeight;
RECT   otmrcFontBox;
int    otmMacAscent;
int    otmMacDescent;
UINT   otmMacLineGap;
UINT   otmusMinimumPPEM;
POINT  otmptSubscriptSize;
POINT  otmptSubscriptOffset;
POINT  otmptSuperscriptSize;
POINT  otmptSuperscriptOffset;
UINT   otmsStrikeoutSize;
int    otmsStrikeoutPosition;
int    otmsUnderscoreSize;
int    otmsUnderscorePosition;
PSTR   otmpFamilyName;
PSTR   otmpFaceName;
PSTR   otmpStyleName;
PSTR   otmpFullName;
} OUTLINETEXTMETRIC;
```

Contains metrics describing a TrueType font. The sizes returned in are given in logical units (depending on the specified display context's current mapping mode). The following table describes the members.

Member	Description
otmSize	Size of this structure, in bytes
otmTextMetrics	*TEXTMETRIC* structure containing additional font information
otmFiller	Value that causes this structure to be byte aligned
otmPanoseNumber	The *PANOSE* number for this font
otmfsSelection	Nature of the font pattern. Can be a combination of the following bits: **Bit** **Meaning** 0 Italic 1 Underscore 2 Negative 3 Outline 4 Strikeout 5 Bold
otmfsType	Specifies whether the font is licensed. Licensed fonts must not be modified or exchanged. If bit 1 is set, the font may not be embedded in a document. If bit 1 is clear, the font can be embedded. If bit 2 is set, the embedding is read-only.

Member	Description
otmsCharSlopeRise	Slope of the cursor. This value is 1 if the slope is vertical. Applications can use this value and the value of the *otmsCharSlopeRun* member to create an italic cursor that has the same slope as the main italic angle (specified by the *otmItalicAngle* member).
otmsCharSlopeRun	Slope of the cursor. This value is zero if the slope is vertical. Applications can use this value and the value of the *otmsCharSlopeRise* member to create an italic cursor that has the same slope as the main italic angle (specified by the *otmItalicAngle* member).
otmItalicAngle	Main italic angle of the font, in counterclockwise degrees from vertical. Regular (roman) fonts have a value of zero. Italic fonts typically have a negative italic angle (that is, they lean to the right).
otmEMSquare	Number of logical units defining the x- or y-dimension of the em square for this font. (The number of units in the x- and y-directions are always the same for an em square).
otmAscent	Maximum distance characters in this font extend above the base line. This is the typographic ascent for the font.
otmDescent	Maximum distance characters in this font extend below the base line. This is the typographic descent for the font.
otmLineGap	Typographic line spacing.
otmsCapEmHeight	Not supported.
otmsXHeight	Not supported.
otmrcFontBox	Bounding box for the font.
otmMacAscent	Maximum distance characters in this font extend above the base line for the Macintosh® computer.
otmMacDescent	Maximum distance characters in this font extend below the base line for the Macintosh computer.
otmMacLineGap	Line-spacing information for the Macintosh computer.
otmusMinimumPPEM	Specifies the smallest recommended size for this font in pixels per em-square.
otmptSubscriptSize	Recommended horizontal and vertical size for subscripts in this font.
otmptSubscriptOffset	Recommended horizontal and vertical offset for subscripts in this font. The subscript offset is measured from the character origin to the origin of the subscript character.
otmptSuperscriptSize	Recommended horizontal and vertical size for superscripts in this font.
otmptSuperscriptOffset	Recommended horizontal and vertical offset for superscripts in this font. The superscript offset is measured from the character base line to the base line of the superscript character.
otmsStrikeoutSize	Width of the strikeout stroke for this font. Typically, this is the width of the em-dash for the font.
otmsStrikeoutPosition	Position of the strikeout stroke relative to the base line for this font. Positive values are above the base line and negative values are below.
otmsUnderscoreSize	Thickness of the underscore character for this font.
otmsUnderscorePosition	Position of the underscore character for this font.
otmpFamilyName	Offset from the beginning of the structure to a string specifying the family name for the font.
otmpFaceName	Offset from the beginning of the structure to a string specifying the typeface name for the font. (This typeface name corresponds to the name specified in the *LOGFONT* structure.)

Member	Description
otmpStyleName	Offset from the beginning of the structure to a string specifying the style name for the font.
otmpFullName	Offset from the beginning of the structure to a string specifying the full name for the font. This name is unique for the font and often contains a version number or other identifying information.

RGBQUAD struct

windows.h

```
typedef struct tagRGBQUAD {
 BYTE rgbBlue;
 BYTE rgbGreen;
 BYTE rgbRed;
 BYTE rgbReserved;
} RGBQUAD;
```

Describes a color consisting of relative intensities of red, green, and blue. The following table describes the members.

Member	Description
rgbBlue	Intensity of blue
rgbGreen	Intensity of green
rgbRed	Intensity of red
rgbReserved	Must be zero

RGBTRIPLE struct

windows.h

```
typedef struct tagRGBTRIPLE {
 BYTE rgbtBlue;
 BYTE rgbtGreen;
 BYTE rgbtRed;
} RGBTRIPLE;
```

Describes a color consisting of relative intensities of red, green, and blue.

Member	Description
rgbtBlue	Blue intensity
rgbtGreen	Green intensity
rgbtRed	Red intensity

TEXTMETRIC struct

windows.h

The 16-bit version formatted and packed differently than the 32-bit version:

```
typedef struct tagTEXTMETRIC {
 int tmHeight;
 int tmAscent;
```

```
  int tmDescent;
  int tmInternalLeading;
  int tmExternalLeading;
  int tmAveCharWidth;
  int tmMaxCharWidth;
  int tmWeight;
  BYTE tmItalic;
  BYTE tmUnderlined;
  BYTE tmStruckOut;
  BYTE tmFirstChar;
  BYTE tmLastChar;
  BYTE tmDefaultChar;
  BYTE tmBreakChar;
  BYTE tmPitchAndFamily;
  BYTE tmCharSet;
  int tmOverhang;
  int tmDigitizedAspectX;
  int tmDigitizedAspectY;
} TEXTMETRIC;
```

32-bit version:

```
typedef struct tagTEXTMETRIC {
  LONG tmHeight;
  LONG tmAscent;
  LONG tmDescent;
  LONG tmInternalLeading;
  LONG tmExternalLeading;
  LONG tmAveCharWidth;
  LONG tmMaxCharWidth;
  LONG tmWeight;
  LONG tmOverhang;
  LONG tmDigitizedAspectX;
  LONG tmDigitizedAspectY;
  BYTE tmFirstChar;
  BYTE tmLastChar;
  BYTE tmDefaultChar;
  BYTE tmBreakChar;
  BYTE tmItalic;
  BYTE tmUnderlined;
  BYTE tmStruckOut;
  BYTE tmPitchAndFamily;
  BYTE tmCharSet;
} TEXTMETRIC;
```

Contains basic information about a physical font. All sizes are in logical units (mapping mode dependent). The table below describes the members.

Member	Description
tmHeight	Height of character cells (*tmAscent* + *tmDescent*).
tmAscent	Ascent of character cells (in units above base line).

Member	Description
tmDescent	Descent of character cells (in units below base line).
tmInternalLeading	Amount of leading (space) inside the bounds set by *tmHeight*. Diacritical characters occur in this area.
tmExternalLeading	Amount of extra leading (space) added between rows. Sometimes set to zero.
tmAveCharWidth	Average width of characters in the font. Does not include overhang.
tmMaxCharWidth	Width of the widest character in the font.
tmWeight	Font weight. Can be one of the following values: FW_DONTCARE FW_SEMIBOLD FW_THIN FW_DEMIBOLD FW_EXTRALIGHT FW_BOLD FW_ULTRALIGHT FW_EXTRABOLD FW_LIGHT FW_ULTRABOLD FW_NORMAL FW_BLACK FW_REGULAR FW_HEAVY FW_MEDIUM
tmItalic	Italic font if not 0
tmUnderlined	Underlined font if not 0
tmStruckOut	Struckout font if not 0
tmFirstChar	Value of first character defined in font
tmLastChar	Value of last character defined in font
tmDefaultChar	Value of non-font substitution character
tmBreakChar	Word-break character value for text justification
tmPitchAndFamily	Pitch and family of the selected font. The four low-order bits identify the type of font, as follows: **Value** **Meaning** TMPF_FIXED_PITCH Fixed-pitch font. TMPF_VECTOR Vector or TrueType font. TMPF_TRUETYPE TrueType font. TMPF_DEVICE Device font. The four high-order bits of this member designate the font family. The *tmPitchAndFamily* member can be combined with the hexadecimal value 0xF0 by using the bitwise AND operator and can then be compared with the font family names for an identical match. The following font families are defined: **Value** **Meaning** FF_DECORATIVE Novelty fonts FF_DONTCARE Don't care or don't know FF_MODERN Fonts with constant stroke width, with or without serifs FF_ROMAN Fonts with variable stroke width and with serifs FF_SCRIPT Fonts designed to look like handwriting FF_SWISS Fonts with variable stroke width and without serifs
tmCharSet	Specifies the character set of the font. The following values are defined: ANSI_CHARSET DEFAULT_CHARSET SYMBOL_CHARSET

Member	Description
	SHIFTJIS_CHARSET
	OEM_CHARSET
tmOverhang	Extra width that is added to some synthesized fonts
tmDigitizedAspectX	Horizontal aspect of the device for which the font was designed. The ratio of *tmDigitizedAspectX* and *tmDigitizedAspectY* is the aspect ratio of the device for which the font was designed.
tmDigitizedAspectY	Specifies the vertical aspect of the device for which the font was designed. The ratio of *tmDigitizedAspectX* and *tmDigitizedAspectY* is the aspect ratio of the device for which the font was designed.

WNDCLASS struct

windows.h

```
typedef struct tagWNDCLASS {
 UINT    style;
 WNDPROC lpfnWndProc;
 int     cbClsExtra;
 int     cbWndExtra;
 HANDLE  hInstance;
 HICON   hIcon;
 HCURSOR hCursor;
 HBRUSH  hbrBackground;
 LPCTSTR lpszMenuName;
 LPCTSTR lpszClassName;
} WNDCLASS;
```

Contains *TWindow* class registration attributes, described in the following table:

Member	Description	
style	Class style. Can be any combination of the following values:	
	Value	**Meaning**
	CS_BYTEALIGNCLIENT	Aligns the window's client area on the byte boundary (in the x direction).
	CS_BYTEALIGNWINDOW	Aligns a window on a byte boundary (in the x direction) to enhance performance during operations that involve moving or sizing the window. This style affects the width of the window and its horizontal position on the display.
	CS_CLASSDC	Allocates a device context (DC) to be shared by all windows in the class.
	CS_DBLCLKS	Sends double-click messages to the window procedure while the cursor is within a window belonging to the class.
	CS_GLOBALCLASS	Allows an application to create a window of the class regardless of the value of the *hInstance* parameter used at window creation.
lpfnWndProc	Points to the window procedure.	
cbClsExtra	Number of extra bytes to allocate following the window-class structure. The operating system initializes the bytes to zero.	
cbWndExtra	Number of extra bytes to allocate following the window instance. Initializedt to 0.	

Member	Description
hInstance	Identifies the instance that the window procedure of this class is within.
hIcon	Identifies the class icon. Must be a handle of an icon resource. If NULL, an application must draw an icon whenever the user minimizes the application's window.
hCursor	Identifies the class cursor. Must be a handle of a cursor resource. If NULL, an application must set the cursor shape whenever the mouse moves into the application's window.
hbrBackground	Identifies the class background brush. Can be a handle to the physical brush to be used for painting the background, or it can be a color value. A color value must be one of the standard system colors listed below, plus 1. If a color value is given, you must convert it to one of the following HBRUSH types: COLOR_ACTIVEBORDER COLOR_HIGHLIGHTTEXT COLOR_ACTIVECAPTION COLOR_INACTIVEBORDER COLOR_APPWORKSPACE COLOR_INACTIVECAPTION COLOR_BACKGROUND COLOR_MENU COLOR_BTNFACE COLOR_MENUTEXT COLOR_BTNSHADOW COLOR_SCROLLBAR COLOR_BTNTEXT COLOR_WINDOW COLOR_CAPTIONTEXT COLOR_WINDOWFRAME COLOR_GRAYTEXT COLOR_WINDOWTEXT COLOR_HIGHLIGHT An application should not delete these brushes, because a class may be used by multiple instances of an application. When this member is NULL, an application must paint its own background whenever it is requested to paint in its client area.
lpszMenuName	Points to a null-terminated character string that specifies the resource name of the class menu, as the name appears in the resource file. If NULL, windows belonging to this class have no default menu.
lpszClassName	Points to a null-terminated string or is an atom that specifies the window class name.

PAINTSTRUCT struct

windows.h

16-bit version:

```
typedef struct tagPAINTSTRUCT {
 HDC  hdc;
 BOOL fErase;
 RECT rcPaint;
 BOOL fRestore;
 BOOL fIncUpdate;
 BYTE rgbReserved[16];
} PAINTSTRUCT;
```

32-bit version:

```
typedef struct tagPAINTSTRUCT {
 HDC  hdc;
 BOOL fErase;
 RECT rcPaint;
 BOOL fRestore;
```

```
 BOOL fIncUpdate;
 BYTE rgbReserved[32];
} PAINTSTRUCT;
```

Contains information needed by an application to paint the client area of a window.

Member	Description
hdc	The display device context used for painting.
fErase	If non-zero erase the background.
rcPaint	Specifies the upper-left and lower-right corners of the rectangle to be painted.
fRestore	Reserved.
fIncUpdate	Reserved.
rgbReserved	Reserved.

PALETTEENTRY struct

windows.h

```
typedef struct tagPALETTEENTRY {
 BYTE peRed;
 BYTE peGreen;
 BYTE peBlue;
 BYTE peFlags;
} PALETTEENTRY;
```

Specifies the color and usage of an entry in a logical color palette (see *LOGPALETTE* struct). The following table describes the settings:

Member	Description	
peRed	Specifies a red intensity value for the palette entry.	
peGreen	Specifies a green intensity value for the palette entry.	
peBlue	Specifies a blue intensity value for the palette entry.	
peFlags	Specifies how the palette entry is to be used. The *peFlags* member may be set to NULL or one of these values:	
	Value	**Meaning**
	PC_EXPLICIT	The low-order word of the logical palette entry designates a hardware palette index. This flag allows the application to show the contents of the display device palette.
	PC_NOCOLLAPSE	The color is placed in an unused system palette entry instead of being matched to an existing system palette color.
	PC_RESERVED	Specifies that the logical palette entry be used for palette animation, preventing other windows from matching colors to the palette entry since the color frequently changes. If an unused system-palette entry is available, this color is placed in that entry. Otherwise, the color is available for animation.

XFORM struct

windows.h

```
typedef struct  tagXFORM {
 FLOAT eM11;
```

```
 FLOAT eM12;
 FLOAT eM21;
 FLOAT eM22;
 FLOAT eDx;
 FLOAT eDy;
} XFORM;
```

Specifies a world-space to page-space transformation. The following table describes the members.

Member	Description	
eM11	Specifies the following:	
	Value	**Meaning**
	Scaling	Horizontal scaling component
	Rotation	Cosine of rotation angle
	Reflection	Horizontal component
eM12	Specifies the following:	
	Value	**Meaning**
	Shear	Horizontal proportionality constant
	Rotation	Sine of the rotation angle
eM21	Specifies the following:	
	Value	**Meaning**
	Shear	Vertical proportionality constant
	Rotation	Negative sine of the rotation angle
eM22	Specifies the following:	
	Value	**Meaning**
	Scaling	Vertical scaling component
	Rotation	Cosine of rotation angle
	Reflection	Vertical reflection component
eDx	Specifies the horizontal translation component.	
eDy	Specifies the vertical translation component.	

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

0

P

Q

R

S

T

U

V

W

X

Y

Borland

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Brazil, Canada, Chile, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # BCP1245WW21775 • BOR 7773