Dynace Windows Development System

User Manual
Version 4.02
June 25, 2002

by Blake McBride

This manual was typeset with the TEX typesetting system developed by Donald Knuth.

Short Contents

1 Introduction 1
2 Concepts 9
3 Mechanics 17
4 Library Reference 23
Method Index 215

Table of Contents

1 Introduction 1

1.1 Overview 1
1.2 Benefits 2
1.3 A Short Example 2
1.4 Features 3
1.5 Reasons to use WDS 3
1.6 Installation 4
1.7 Contents 4
1.8 Learning The System 5
1.9 Examples & Tutorial 6
1.10 Manual Organization 7
1.11 Support, Contact & Upgrades 7
1.12 Use, Copyrights & Trademarks 8
1.13 Credits 8

2 Concepts 9

2.1 Windows 9
2.1.1 Main Window 9
2.1.2 Child Window 9
2.1.3 Popup Window 9
2.2 Menus 10
2.3 Dialogs 10
2.3.1 Modal Dialogs 10
2.3.2 Modeless Dialogs 10
2.4 Controls 10
2.4.1 Text Control 10
2.4.2 Numeric Control 10
2.4.3 Date Control 11
2.4.4 Push Button 11
2.4.5 Check Box 11
2.4.6 Radio Button 11
2.4.7 List Box 11
2.4.8 Combo Box 11
2.4.9 Scroll Bar 11
2.5 Help System 11
2.6 Cursors 11
2.6.1 System Cursors 11
2.6.2 External Cursors 12
2.7 Icons 12
2.7.1 System Icons 12
2.7.2 External Icons 12
2.8 Fonts 12

2.8.1 System Fonts ... 12
2.8.2 External Fonts ... 12
2.9 Brushes ... 12
2.9.1 System Brushes ... 12
2.9.2 Stock Brushes ... 13
2.9.3 Solid Brushes ... 13
2.9.4 Hatch Brushes ... 13
2.10 Pens ... 13
2.10.1 Stock Pens ... 13
2.10.2 Custom Pens ... 13
2.11 Common Dialogs ... 13
2.11.1 Color Dialog ... 13
2.11.2 Font Dialog ... 13
2.11.3 File Dialog ... 13
2.11.4 Print Dialog ... 13
2.12 Printing ... 14
2.13 Resources ... 14
2.14 Dynamic Link Libraries (DLL) ... 14
2.15 Win16, Win32, Win32s ... 15
2.16 Message Passing Architecture ... 16

3 Mechanics ... 17
3.1 Build Modes ... 17
3.2 IDE Build Setup ... 17
3.2.1 Microsoft IDE ... 17
3.2.2 Borland IDE ... 17
3.2.3 Symantec IDE ... 18
3.3 DMAKE Build Setup ... 18
3.4 Building Dynace & WDS From Scratch ... 18
3.5 Examples Setup ... 19
3.6 Example Files ... 19
3.7 Building The Examples ... 20
3.8 Debugging With The IDE ... 21
3.9 Building Your Own Application ... 21
3.10 DMAKE ... 21
3.11 DPP ... 21

4 Library Reference ... 23
4.1 A Note To Dynace Language Users ... 23
4.2 Class Hierarchy ... 23
4.3 Application ... 25
4.4 Windows ... 36
4.4.1 Main Window ... 68
4.4.2 Child Windows ... 69
4.4.3 Popup Windows ... 70
4.5 Printing ... 71

4.6 Menus ... 84
4.6.1 External Menus ... 88
4.6.2 Internal Menus ... 90
4.7 Popup Menus ... 92
4.8 Dialogs ... 95
4.8.1 Standard Dialog Method Arguments ... 95
4.8.2 Dialog Methods ... 95
4.8.3 Modal Dialogs ... 113
4.8.4 Modeless Dialogs ... 114
4.9 Controls ... 116
4.9.1 Standard Control Method Arguments ... 116
4.9.2 Control Methods ... 116
4.9.3 Text Control ... 122
4.9.4 Numeric Control ... 128
4.9.5 Date Control ... 134
4.9.6 Push Buttons ... 138
4.9.7 Check Boxes ... 140
4.9.8 Radio Buttons ... 144
4.9.9 List Boxes ... 148
4.9.10 Combo Boxes ... 157
4.9.11 Scroll Bars ... 169
4.9.12 Custom Controls ... 174
4.10 Cursors ... 174
4.10.1 System Cursors ... 175
4.10.2 External Cursors ... 176
4.11 Icons ... 177
4.11.1 System Icons ... 179
4.11.2 External Icons ... 180
4.12 Fonts ... 181
4.12.1 System Fonts ... 184
4.12.2 External Fonts ... 184
4.13 Brushes ... 186
4.13.1 Stock Brushes ... 188
4.13.2 Solid Brushes ... 188
4.13.3 System Brushes ... 189
4.13.4 Hatch Brushes ... 190
4.14 Pens ... 191
4.14.1 Stock Pens ... 193
4.14.2 Custom Pens ... 193
4.15 Help System ... 194
4.16 Common Dialogs ... 197
4.16.1 File Selection Dialog ... 197
4.16.2 Printer Selection and Configuration Dialog ... 202
4.16.3 Color Selection Dialog ... 205
4.16.4 Font Selection Dialog ... 208
4.17 Dynamic Link Libraries ... 212

Method Index **215**

1 Introduction

The Dynace Windows Development System (WDS) is a Dynace class library which enables a C programmer with no knowledge of C++, Dynace, the Windows API or message-driven architecture to write real Windows applications with an absolute minimum learning curve and number of lines of code. In fact, it is possible to become familiar enough with windows, menus, dialogs and controls using WDS to write a Windows application after just one day!

The programmer is able to write the application using familiar C in a familiar procedural fashion. Instead of taking the usual fifty plus lines of code necessary to implement a typical "Hello World" program, with WDS it takes four lines of code! Fully functional menus and dialogs can be implemented with WDS in a handful of lines instead of the hundreds it takes using the Windows API or other available tools.

WDS applications are portable across Windows 3.1 (Win16), Win32s, Windows 95 and Windows NT (Win32).

Dynace (pronounced *dī-ne-sē*), what WDS is based upon, stands for a "DYNAmic C language Extension". It is an object oriented extension to the C or C++ languages. Dynace is written in the C language and designed to be as portable as possible. It solves many problems associated with C++ and adds features previously only available in languages such as CLOS or Smalltalk without their overhead. Dynace is fully documented in another manual.

1.1 Overview

The C language is the most popular and well known among the language choices available for the PC and Unix environments. Therefore, there are many more programmers who know and feel comfortable with the C language than any other. All the popular tools available for software development under Windows 3.1, 95 and Windows NT use the C++ language. C++ is a complex superset of the C language with questionable benefits. There is a significant learning curve associated with going from proficient C ability to C++.

Although the currently available tools for Windows development, which work in association with C++, provide a very high degree of flexibility and power, they are tremendously complex to learn and use effectively. Given the tremendous complexity of both the Windows development tools currently available on the market and the fact that these tools are all based on C++, the time necessary for a normal C programmer to become knowledgeable and proficient in Windows development, including both C++ and the Windows tools, is an absolute minimum of six months. This time may vary to a period exceeding a year.

The currently available tools require an enormous amount of code (lines of program text) in order to achieve the most fundamental functionality. Although the current tools actually generate most of this code, the bottom line is that any real life application will end up having an enormous amount of code. There is a clear relationship between lines of code and a) how maintainable a program is for finding bugs or making enhancements, and b) how difficult it is for new programmers to get up to speed with respect to the new tools

and application. Given the cost of software development and the tremendous backlog, this issue is of paramount importance.

The focus of WDS is to enable a normal C programmer to learn and be able to write and understand a fundamental Windows program in one day. Given just a little more time with WDS (in terms of days), the programmer will be able to write and understand real Windows applications in a minimum amount of time and with a very minimum amount of code. There's no need to learn a new language, such as C++. The programmer may use his existing knowledge in C and just needs to learn a very high level and simple to understand set of tools. Instead of taking hundreds of lines of code to add a new dialog, as required by the existing tools, WDS can accomplish the same task in half a dozen lines. Instead of taking up to one hundred lines of code just to bring up the main application window, WDS just requires four lines!

1.2 Benefits

There are three main benefits to using WDS over the other available options. The first is that the learning curve associated with Dynace for Windows allows a programmer or programming team to get up-to-speed with respect to Windows programming in an absolute minimum amount of time. The difference in time is days instead of six months or more. This directly translates into saved dollars and increases the success of projects.

The second main benefit is that since application features may be implemented in tens of lines of program code, instead of the hundreds or thousands of lines required by existing tools, applications may be developed in a drastically reduced time frame.

The third main benefit is that since WDS is so easy to learn and requires so few lines of code for application development, applications developed with WDS are much easier to debug, maintain and enhance. Programmer turnover is much less a problem due to the fact that new programmers can get up-to-speed in a minimum amount of time.

With WDS there is never a need for a code generator or wizard since so few lines are needed to create the application. All your code is application specific.

In short, WDS is an invaluable tool for Windows application development during the learning, development, and maintenance phases of application development. As an added bonus, WDS is portable across the Windows 3.1, Windows 95, and Windows NT platforms. Therefore, a programmer can develop once and target the three most popular environments.

1.3 A Short Example

The following complete example illustrates how easy it is to get started with WDS. Implementing this exact functionality using the Windows API would take more then fifty lines of code!

```
#include "generics.h"

int     start()
{
        object  win;

        win = vNew(MainWindow, "My Test Application");

        vPrintf(win, "Hello, World!\n");

        return gProcessMessages(win);
}
```

1.4 Features

This section enumerates some of the key features of The Dynace Windows Development System (WDS).

- Drastically reduces the learning curve associated with Windows development
- Drastically reduced the lines of code necessary to build Windows apps
- Applications are portable between Windows 3.1 (Win16), Win32s, Windows 95 and Windows NT
- Applications are easier to debug, enhance and pass off to new programmers since it is straight C and so few lines
- Full support for main, popup and child windows, menus, modal and modeless dialogs, cursors, icons, fonts, brushes, and pens
- Support for all standard Windows controls as well as several supplied controls
- Easy to use routines for printing reports
- Support for most of the Windows common dialogs
- Compiles with a regular C or C++ compiler – not an interpreter – runs FAST!
- Full support for the Windows help system at all levels of context
- Support for external DLL access
- Access of full source code to WDS
- Applications are royalty free (with the appropriate license)

1.5 Reasons to use WDS

The following lists several reasons to use the Dynace Windows Development System:

1. Drastically reduce the learning curve associated with learning to write for Windows – be able to write real Windows applications in a few days instead of the normal 6 months to a year learning curve.
2. Be able to write Windows apps fast using only a few lines of code – do "Hello World" in 4 lines, menus and dialog in a few lines – write completely functional dialogs in a handful of lines instead of the hundreds it normally takes.

3. Drastically reduce the time necessary to debug, maintain and enhance the app since it's so few lines of code.
4. Write apps which are portable to 16 and 32 bit environments including Windows 3.1 (Win16), Win32s, Windows 95 and Windows NT (Win32).
5. Be much less dependent on a few "expert" programmers since the code is much easier to follow and learn.
6. No need to learn a new language since WDS apps are normal C code.
7. Don't be the victim of a vendor since WDS comes with full source code.
8. Be able to take advantage of the very advanced object oriented capabilities of the Dynace Object Oriented Extension to C, which WDS is built on top of.
9. Your apps will run very fast since they are compiled into optimized machine code by your compiler.
10. Integrates well with existing compiler vendor's IDEs. No need to learn a new set of tools.
11. It's easier to port non-Windows apps to Windows with WDS since WDS encapsulates much of the message-driven architecture associated with Windows programming.
12. WDS applications are royalty free (with the appropriate license).

1.6 Installation

The diskette(s) distributed with Dynace are standard DOS formatted diskette(s). The files on the diskette(s) have been compressed in order to reduce the quantity of diskettes.

See the file `README`, located on the first disk, for installation instructions.

1.7 Contents

Once the system has been installed there will exist a series of directories under the `dynace` directory. The system will contain the following directories:

\DYNACE
: This is the root of the Dynace system.

\DYNACE\LIB
: This is the location of all the Dynace libraries. You may wish to add this directory to the list of paths your linker searches.

\DYNACE\INCLUDE
: This is the location of the include files necessary to compile Dynace applications. You may need to add this directory to the path your compiler uses to search include files.

\DYNACE\BIN
: Executable files necessary for development with the Dynace system. This directory should be added to your normal search path for executable programs.

\DYNACE\EXAMPLES
: Example programs used to learn & demonstrate the Dynace object oriented extension to C.

\DYNACE\WINEXAM
: Example programs used to learn & demonstrate WDS.

\DYNACE\DOCS
: Misc. documentation files.

\DYNACE\MANUAL
: The Dynace and WDS manuals.

\DYNACE\KERNAL
: Complete source to the Dynace kernel.

\DYNACE\CLASS
: Source to all of the Dynace base classes.

\DYNACE\THREADS
: Complete source to the multi-threader, pipes and semaphores.

\DYNACE\GENERICS
: Files necessary to build the system generics files from scratch.

\DYNACE\DPP
: Complete source to the `dpp` utility.

\DYNACE\WINDOWS
: Complete source to the Dynace WDS library.

\DYNACE\UTILS
: Complete source to the utility programs.

The only files that are absolutely necessary for a developer are those located in the `\DYNACE\LIB`, `\DYNACE\INCLUDE` and `\DYNACE\BIN` directories and `\DYNACE\UTILS\STARTUP.MK`.

1.8 Learning The System

This manual contains a description of the WDS concepts, a detailed description of the WDS system and complete reference to all WDS classes. It is designed for a programmer who is proficient in the C language, but has little or no knowledge of Windows programming or Dynace.

In addition to this manual, the accompanying example programs will be needed in order to learn how to use the system. The example programs included with WDS provide a step-by-step tutorial to getting started.

A proficient C programmer with a pre-existing compiler environment should be able to install this package and feel comfortable enough with the system to write a simple Windows application in one day.

The best approach to learning WDS would be to start by reading chapters 1 (Introduction) and 2 (Concepts). Then refer to chapter 3 (Mechanics) while working through the WDS example programs.

If you wish to dive right in and get a feel for the system you may perform the setup procedure described in chapter 3 and then go directly to the example programs.

While working through the examples you may refer to the index (located in the back of the manual) to locate information on all WDS functions.

Although it is not necessary, you may also learn and incorporate the many powerful object oriented features of the Dynace object oriented extension to C, which is what this library is based on. Dynace is fully documented in a separate manual.

1.9 Examples & Tutorial

The WDS system includes a series of examples which serve both as a Quick Start and a Tutorial. Theses examples are located under the `\DYNACE\WINEXAM` directory. The examples located under the `\DYNACE\EXAMPLES` directory are strictly for the Dynace object oriented extension to C and are not needed to use WDS.

Each example is contained in its entirety in an independent directory. This is done in order to illustrate the exact files and steps necessary to create a single application.

The example programs are contained in sub-directories under the *WINEXAM* directory and are named *EXAMnn* (where the *nn* is a two digit number). These numbers are significant in that they describe the correct order that the examples should be followed. Each example depends on knowledge built up in previous examples and is not repeated.

Each example contains a *readme* file which describes information relating to the purpose of the example and build instructions. These files should be read first. Each example also includes makefiles which are associated with the compiler vendor being used. Full source is included.

Each example program contains source (.c) files. There is also a mirror of each source file with a `.txt` file extension. These text files contain duplicates of the source file with extensive documentation in the form of comments. These files should be viewed in order to understand the code in the application.

Complete setup and build instructions are contained in Chapter 3.

The example programs are tutorial in nature and should be considered the main source of information used to get started. The manual augments the examples with introductory material, concepts and a complete reference.

The following is a list of the enclosed examples:

01 Illustrates the steps necessary to compile and link a bare bones Windows application.

02 Illustrates the use of menus and message boxes.

03 Illustrates the minimum steps necessary for the creation of a modal dialog.

04 Illustrates the addition of text controls to a dialog.

05 Illustrates the addition of numeric and date controls to a dialog.

06 Illustrates the addition of push button controls to a dialog.

07 Illustrates the addition of radio buttons and check boxes to a dialog.

08 Illustrates the addition of a combo box and list box to a dialog.

09 Illustrates the addition of a scroll bar to a dialog.

10 Illustrates a modal dialog with all control types initialized and used.

11 Illustrates the process of printing to the default printer.

12 Illustrates the process of printing to a user selected and configured printer.

13 Illustrates various methods of outputting text to the printer and changing pages.

14 Illustrates printer output utilizing different fonts.

15 Illustrates printer graphics output, output scaling, brushes and pens.

16 Illustrates the use of modeless dialogs.

17 Illustrates the use of a modeless dialog which isn't associated with a parent window.

18 Illustrates the use of the context sensitive help system.

19 Illustrates the creation of Dynace classes in conjunction with a WDS application.

See the file `\DYNACE\WINEXAM\LIST` for the most up-to-date list of example programs.

1.10 Manual Organization

This manual serves as both a user manual and a complete reference manual to the WDS system. The Dynace object oriented extension to C is documented in a separate manual.

Chapter 1 (Introduction) covers background material needed to orient a new user.

Chapter 2 (Concepts) covers fundamental concepts associated with the Windows environment. It does this without introducing very much syntax or other mechanics.

Chapter 3 (Mechanics) documents the exact procedures necessary to setup and use the WDS system. The example programs should be viewed subsequent to setting the system up.

Chapter 4 (Class Reference) provides a detailed reference to all classes and methods associated with the class library included with the WDS system.

The index (located in the back) provides a complete alphabetical listing of all classes, methods, and macros described in chapter 4.

1.11 Support, Contact & Upgrades

We will respond to all questions or comments submitted by registered users (see `REGISTER.DOC`). We will also notify all registered users of bug fixes and enhancements via e–mail.

In addition, it is our hope to hold a public forum concerning Dynace on the internet news group comp.lang.misc. If traffic becomes significant we'd also like to create a comp.lang.dynace news group.

Internet blake@mcbride.name

1.12 Use, Copyrights & Trademarks

1.13 Credits

The Dynace Object Oriented Extension to C, Windows Development System and their associated documentation were written by Blake McBride (blake@mcbride.name).

2 Concepts

The Windows environment has its own lexicon and way of doing things. This chapter is designed to explain that lexicon and give the programmer a certain comfort level with respect to elements of Windows programming.

2.1 Windows

Windows are the rectangular region on your screen where a user interacts with the application. There are many things which may be done with a window. Typically, a user may move, resize or change a window into an icon. Actually, dialogs and controls (discussed further down) are actually little windows located inside bigger windows. Therefore, it is also possible for a user to interact in a very of ways with a specific window.

There are many attributes which may be associated with a window. Each window may have a system menu associated with it, a title bar, a minimize/maximize buttons, various colors, sizes, positions, and border types.

There are three main window types: main, child and popup.

2.1.1 Main Window

There is normally exactly one main window associated with any given application. This is the first window which comes up when an application starts up, the one which acts as the fulcrum for all the facilities of the application, and the one which contains the main menu associated with the application. This is also the window which has the application icon associated with it. This is so when the application is iconized the appropriate icon will displayed.

2.1.2 Child Window

In addition to the main window, there are child windows. A child window is a rectangular region or window which is directly associated with some other (parent) window. Child windows are always restricted to being located inside its parent window. It can't be moved outside its parent. Also, when a parent window gets iconized (minimized into an icon), its child windows follow suit. When a parent window gets moved across the screen, the child windows will follow such that their relative position in their parent remains constant. When a child window is created it is positioned relative to its parent.

2.1.3 Popup Window

In addition to child windows, the system also supports popup windows. A popup window is a rectangular region which acts, to a greater or lesser extent, independently of any other window. A popup window may exist outside of any other window and be moved or resized independent of any other window. Popup windows may or may not have a parent. If a parent window is associated with a popup and the parent gets iconized, the popup will follow suit. If no parent is associated with a popup it can be iconized independent of any other window. Since popup windows are independent of other windows, they are created with positions which are relative to the entire screen.

2.2 Menus

Menus are the words or labels which typically appear at the top of the main window associated with an application. The user is able to select various application functions by selecting the appropriate menu option. Each menu option may also present the user with additional menu options down to a number of levels.

2.3 Dialogs

Once a user selects a particular application function through the application menu, an applications principal method of providing the user with information, or obtaining information from, is through dialogs. A dialog is a rectangular region (or more precisely, a window) which contains various display information and input fields (controls).

There are two fundamental dialog types: model and modeless.

2.3.1 Modal Dialogs

A modal dialog is one which must be completed or canceled prior to accessing any other part of a given application. If the user is presented with a modal dialog and attempts to activate another window within the same application, the system will just beep. This dialog type, however, does not restrict the user from accessing other applications.

Modal dialogs are used most often because the complexity of an application would grow tremendously if the user were allowed to be in multiple parts of the same application at the same time.

2.3.2 Modeless Dialogs

Modeless dialogs allow the user to switch from one part of a particular application to another without completing or canceling the modeless dialog. The user may switch to another part of an application, obtain additional information, and then return to any portion of the modeless dialog to complete it. The user may even iconize the dialog for another time.

Modeless dialogs are more complex and less frequently used than modal dialogs.

2.4 Controls

Controls are the primary method of obtaining information from the user. They are the input fields, push buttons, selection boxes, scroll bars, etc. Although Windows supports a group of standard controls, there is an endless number of possible types of controls. WDS supports all standard Windows controls in addition to providing several additional ones and enhancements to the standard ones.

2.4.1 Text Control

Text controls are used to obtain textual information in the form of character strings. WDS provides a variety of enhancements to the standard Windows text controls.

2.4.2 Numeric Control

Numeric controls are a WDS supplied control used to obtain numeric data from the user.

2.4.3 Date Control

Date controls are a WDS supplied control used to obtain date information from the user.

2.4.4 Push Button

Push buttons are Windows supplied, WDS enhanced, controls which allow the user to evoke an immediate action function. For example, the common "OK" and "CANCEL" buttons are push buttons. The user clicks on the button and some action gets evoked.

2.4.5 Check Box

Check boxes are Windows supplied, WDS enhanced, controls which allow the user to select among non-exclusive yes/no, on/off type options.

2.4.6 Radio Button

Radio buttons are Windows supplied, WDS enhanced, controls which allow the user to select one option among a group of mutually exclusive options.

2.4.7 List Box

List boxes allow the user to select items from a list of options. The list may be fully displayed, scrollable or drop down in nature.

2.4.8 Combo Box

Combo boxes are a combination of the text control and the list box. With combo boxes, a user my select among a list of options or type in a new item.

2.4.9 Scroll Bar

Scroll bars allow the user to visually select a position within some application specific range via the mouse.

2.5 Help System

Windows provides a powerful help system which enables the user to obtain interactive application specific help information via a "Help" menu option or within a specific context via the "F1" key. WDS supports and makes it easy to use all capabilities of the Windows help system.

2.6 Cursors

Cursors are the visual elements which indicate where a users input focus is or where he wishes it to be. Cursors can also be used to indicate something about the input mode or system status (for example the hour glass is used to indicate that the system is busy). Common cursors are the underscore, the arrow, the ibeam, and the hour glass.

2.6.1 System Cursors

System cursors are common cursors which are predefined by the system. These include most of the cursors most applications would need (such as the underscore, the arrow, the ibeam, and the hour glass) and are easy to select and use.

2.6.2 External Cursors

External cursors are cursors which are application specific and created by the programmer. These cursors may look like virtually anything.

2.7 Icons

Icons are those little picture boxes which are left when an application is minimized. It is normally something the user can identify the application with in order to reactive it.

2.7.1 System Icons

System icons are those generic icons which are predefined by Windows. These system defined icons are easy to select and use, but are of a bit less value since they have little to do with any specific application.

2.7.2 External Icons

External icons are application specific and programmer created icons. They must be designed by the programmer prior to use, but are very flexible, easy to design, and help the user identify the application.

2.8 Fonts

Fonts are the types (such as Helvetica or times) and size (i.e. 10pt) of the characters which appear within an application. There are literally thousands to choose from. Fortunately, however, the system defaults are often sufficient.

2.8.1 System Fonts

The system fonts are those fonts which the user has configured his system to use on a global basis. Use of these fonts makes an application have a familiar look and feel when compared to the rest of the apps on the users machine.

2.8.2 External Fonts

External fonts are programmer selectable fonts and may be any of the many fonts available to the Windows system.

2.9 Brushes

Brushes specify the color and pattern which appears in a window background, in text and graphics.

2.9.1 System Brushes

System brushes are those colors which were selected by the user as global to their Windows environment. The programmer may select, for example, the color the user chose for background windows. The new brush will then have colors the same colors that the user is used to seeing in other applications on their system. System brushes are used most often.

2.9.2 Stock Brushes

Stock brushes allow the programmer to select from a number of common, Windows defined, brushes.

2.9.3 Solid Brushes

Solid brushes are arbitrary color solid brushes which may be selected by the programmer.

2.9.4 Hatch Brushes

Hatch brushes enable the programmer to select from any color and any of a number of Windows defined hatch patterns.

2.10 Pens

Pens are used, principally in graphics, for drawing lines or providing the outline of a figure. The attributes associated with a pen includes color of the line, width of the line, and its style. Common styles include solid, dashed, doted, etc.

2.10.1 Stock Pens

Stock pens allows a programmer to select among a few, common, Windows defined pens.

2.10.2 Custom Pens

Using a custom pen allows the programmer to specify the exact attributes associated with a particular pen.

2.11 Common Dialogs

Windows provides a few common dialogs in order to obtain frequently needed information in a uniform format. Use of these dialogs when appropriate eases the programming normally associated with dialog creation and provides the user with a familiar interface.

2.11.1 Color Dialog

The color dialog provides a convenient method of requesting detailed color selection information from the user.

2.11.2 Font Dialog

The font dialog provides a convenient method of requesting detailed font selection information from the user.

2.11.3 File Dialog

The file dialog provides a convenient method of allowing the user to view and select files.

2.11.4 Print Dialog

The print dialog provides a convenient method of selecting and configuring the printer.

2.12 Printing

WDS provides many convenient and easy to use methods of selecting and configuring printers as well creating reports with a variety of fonts and graphics.

2.13 Resources

Windows programs may use application specific icons, dialogs, menus and cursors. These items are referred to as resources. Each resource has a name and may be accessed by the application program. Resources are created with a graphically oriented tool called a *resource editor*.

Once the needed resources are created, the resource editor saves them to a file called the resource file. The contents of the resource file are then compiled into a binary form using a *resource compiler*. Finally, the compiled resource file is combined with the executable file at link time.

Resource editors and compilers are included and documented along with your C compiler package. The resource editor associated with Microsoft's Visual C++ is called *App Studio*. WATCOM calls theirs the *WATCOM Resource Editor* and Borland calls theirs *Resource Workshop*.

2.14 Dynamic Link Libraries (DLL)

Normally, when a program is linked, all of the library code which is accessed gets copied to the executable being created. Therefore, when the program is run, all the necessary routines are readily available in the executable file. This is called static linking.

When using DLLs, at link time the linker copies very small stub functions instead of the real routines into the executable file. These stub functions, when called by the application code, automatically access the real routines at run time which exist in external DLL files. This is called dynamic linking because it happens dynamically at run time.

The actual code which uses statically or dynamically linked routines is exactly the same. In other words, the process of writing code doesn't change. The only difference is the link commands used and the fact that dynamic linking requires that the DLLs be available at run time.

There are three advantages to using DLLs. The first is that if you have several apps which use the same library routines, you can save disk space by only requiring one copy of the routines on disk (in the DLL instead of a copy in each executable file).

The second is that since Windows knows that both apps are sharing the exact same code (because they use the same DLL), if the user activates both apps, Windows only needs to load one copy of the routines in memory and share the same code for both apps. This saves memory space.

The third advantage of using DLLs is that if you have several apps which use the same routine, and a bug is discovered in that routine, only the DLL needs to be updated. None of the apps will have to be re-linked. The next time any of the apps are run, it will load the new, and repaired, routine.

There are also a couple of drawbacks associated with the use of DLLs. First, instead of just using an executable file (as is the case with statically linked executables) you must make sure to install the appropriate DLLs. This complicates the installation process.

The second, and more troublesome, problem associated with DLLs has to do with versions. Lets say you have some app which uses some common DLL. Everything is compiled and linked and runs well. Several months go by and you start working on a new project. While developing the new app you decide to use some routine in your common DLL which used by the first app. While using the routine located in the DLL you discover a bug or anomaly in the routine, you therefore decide to correct it. Now the second app works fine. A month later you try to use the first app again and it now has all sorts of bugs! How could this happen? It always worked before and you haven't changed it.

What happened is that the first app depended on a particular anomaly associated with the original version of the routine located in the DLL. When the anomaly was corrected, it mysteriously broke the first app. This scenario is more common and troublesome than it may appear!

2.15 Win16, Win32, Win32s

Microsoft has two independent but similar models in which an application may run. Each model has an associated application program interface (API) or library which it links with and uses in order to interface with the OS. The two APIs are called Win16 and Win32.

Win16 is a 16 bit interface designed to run on Windows 3.1. All common Windows 3.1 apps are created using this API. This API has the advantage of being portable to the most environments. Although it will only support 16 bit applications, these apps will run on Windows 3.1, Windows '95 and Windows NT.

Win32 on the other hand is a full featured 32 bit API. The syntax of this API is very similar but not exactly the same as Win16. The main differences have to do with the fact that in a 32 bit environment `int`'s are 32 bits instead of 16. Win32 and Win16 are similar enough, however, that with a little planning, it is not difficult to create apps which are compile time portable between Win16 and Win32. Windows NT is the primary host for Win32 applications, however, these 32 bit applications may run on 16 bit Windows 3.1 as follows.

In order to bridge the gulf between Win16 and Win32 and allow 32 bit applications to run under Windows 3.1, Microsoft created Win32s. Win32s is a large subset of Win32 which supports the same syntax as Win32, supports 32 bit apps, and will run on Windows 3.1 and NT. In order to run on 16 bit Windows 3.1 it is necessary to install some 32 bit extensions to Windows 3.1. These extensions come with the C compiler packages and are freely redistributable. Win32s apps will run in true 32 bit mode under Windows 3.1 or Windows NT using the same executable.

It is important to note that there is no difference between creating a Win32 or Win32s application. Both use the same compiler, compiler flags, linker, linker flags and libraries. The exact same executable is built. There is no way to specifically tell the system which

type of application you wish. The *only* difference between the two is which specific APIs your application calls.

If your application only calls those APIs which are common to Win32 and Win32s then your application will run as a Win32 application under NT and a Win32s under Windows 3.1. However, if your application makes any API calls which are specific to Win32 then those calls will not work under Windows 3.1.

WDS supports Win16 and Win32. In addition, WDS only uses those APIs which are available under Win32s, therefore, WDS applications will run fine under Win32s.

2.16 Message Passing Architecture

Most character based applications are pretty simple in the sense that they (in one form or another) basically get a key, perform some processing, get the next key, perform some more processing, etc. In a graphical environment (such as Windows) this model doesn't work. At any point (like in the middle of entering data in a field) the user may decide to move or resize a window and the application has to be able to handle a myriad of possible input requests (messages) at any point. In order to accommodate this fact, Windows programming uses a different programming model called *Message Driven Architecture* (MDA).

MDA is tremendously more complex and requires much more code to handle than the old get-key model. The main benefit of the Dynace Windows Development System is that WDS encapsulates and hides most of the aspects of MDA. With WDS you can code in a simple, familiar way and produce full featured Windows applications.

3 Mechanics

This chapter, in concert with the WDS examples, describes the tools and actual steps necessary to create WDS application programs. The best method of learning is to go through the example programs located under the `\DYNACE\WINEXAM` directory in the order in which they appear.

3.1 Build Modes

There are two principal modes of building WDS applications. You can either use the Integrated Development Environment (IDE) which comes with your compiler or you can use the included DMAKE command line make utility along with the other command line facilities of your compiler.

Use of the IDE is recommended since it provides a convenient environment for editing, building, debugging and executing your application. However, the DMAKE method is required both for building the WDS libraries from scratch (if you have the source edition) or to build WDS applications which contain new Dynace classes. This is required because the build procedure necessary for Dynace classes is too complex for the IDEs.

Note, however, that creating new Dynace classes is not required in order to build full featured WDS applications.

In addition, only the DMAKE mode is supported when initially building the Dynace and WDS systems from scratch.

3.2 IDE Build Setup

Prior to building any of the examples using the IDE, a few configuration options within the IDE need to be set.

3.2.1 Microsoft IDE

Configure the correct directories for include files and libraries. This can be done via the tools / options / directories menu item.

Configure the IDE such that it will look under the `\DYNACE\INCLUDE` directory (in addition to whatever is already set) for include files.

Additionally, configure the IDE such that it will look under the `\DYNACE\LIB` directory (in addition to whatever is already set) for library files.

3.2.2 Borland IDE

Configure the correct directories for include files and libraries. Since the Borland IDE associates these directories directly with each project, this information will have to be adjusted separately for each of the two `.IDE` files which come with Dynace. These files are under the `\DYNACE\WINEXAM\SETUP` directory and are called `WIN16.IDE` and `WIN32.IDE`. This can be done via the options / project / directories menu item after loading one of the `IDE` files. You must then save the change by using the project / close project menu option. Repeat this procedure for both `IDE` files.

Configure the IDE such that it will look under the `\DYNACE\INCLUDE` directory (in addition to whatever is already set) for include files.

Additionally, configure the IDE such that it will look under the `\DYNACE\LIB` directory (in addition to whatever is already set) for library files.

3.2.3 Symantec IDE

Symantec compiler versions 7.2 and prior do not work due to bugs in their compiler. A patched 7.2 compiler did work but was a little flaky. Version 7.21 seemed to work fine but was not extensively tested.

3.3 DMAKE Build Setup

This section lists environment variables which must be set if the DMAKE build mode is used. If the IDE build mode is used see the IDE Build Setup section for setup details.

The `\DYANCE\BIN` directory should be part of the search path used by your system for executable programs (the `PATH` environment variable).

The following environment variables should be set as follows (modify as appropriate):

```
set MAKESTARTUP=c:\dynace\utils\startup.mk
set TMPDIR=c:\tmp
set DOS4G=quiet
set DOS16M=:4M
```

The `DOS4G` and `DOS16M` environment variables are only necessary when running the 32 bit DOS version of `dpp.exe` which has been compiled with the WATCOM compiler. Win32 or other versions of `dpp` do not require these variables.

Although not required by the example programs, Microsoft users may find it convenient to add `\DYNACE\INCLUDE` and `\DYNACE\LIB` to your compiler's include and library search paths respectively.

Borland users will need to adjust the `BORLAND_HOME` path setting in the `B16.DM` and `B32.DM` makefiles located in the `\DYNACE\WINEXAM\SETUP` and Dynace source code directories. In addition, the Borland resource compiler requires that the `INCLUDE` environment variable be set to where the Borland include files are located, for example:

```
set INCLUDE=d:\bc45\include
```

3.4 Building Dynace & WDS From Scratch

The procedure used to build Dynace & WDS from scratch is fully described in `\DYNACE\DOCS\BUILD.txt`. See that file for build instructions.

New users will want to read all the files in the `docs` directory.

3.5 Examples Setup

All examples are buildable under all supported environments and build procedures, including IDE or DMAKE builds, as well as 16 bit Windows or 32 bit Win32 (NT, Win32s, Windows 95). The only difference is the build procedures or associated makefiles.

In order to avoid having all supported build procedures cluttering up the example directories, the example directories are shipped without any build procedures installed. All the build procedures are located in the `\DYNACE\WINEXAM\SETUP` directory along with batch files used to install the specific build procedures you need.

The following table lists the commands available in order to configure the example programs. Note that each command must be executed from the `\DYNACE\WINEXAM\SETUP` directory unless otherwise indicated. There is no harm in configuring the examples for more than one configuration at the same time.

M16IDE IDE build with Microsoft Visual C 16 bit

M16DM DMAKE build with Microsoft Visual C 16 bit

M32IDE IDE build with Microsoft Visual C 32 bit

M32DM DMAKE build with Microsoft Visual C 32 bit

B16IDE IDE build with Borland C 16 bit

B16DM DMAKE build with Borland C 16 bit

B32IDE IDE build with Borland C 32 bit

B32DM DMAKE build with Borland C 32 bit

S32IDE IDE build with Symantec C 32 bit

S32DM DMAKE build with Symantec C 32 bit

CLEANALL
: Used to delete all files produced by builds in all examples

REALCLN
: Used to delete all files produced by builds and all build configuration files (basically to go back to an as-shipped state)

CLEAN Used to delete all files produced by a build for a single example (execute from the particular example's directory)

3.6 Example Files

This section documents the files which are contained in each example program.

README This file describes the object of the example. It should be read first.

MAIN.C This file is the complete source code to the example program.

MAIN.TXT
: This is a fully commented version of MAIN.C. It describes all aspects of the current example which are unique.

MAIN.RC This is the resource script describing the resources used by the application. It is created and edited via the indigenous resource editor.

RESOURCE.H
This is another file created by the resource editor. It is used by the application code to associate resources defined in MAIN.RC to macro names.

ALGOCORP.ICO
An icon picturing the Algorithms Corporation logo.

MAIN.DEF
Link definition file. Only needed for Windows 3.1 applications.

In addition to the above, the following build/system specific make or project files are used.

MAIN.MAK
Project file for Microsoft Visual C 16 bit

M16.DM DMAKE makefile for Microsoft Visual C 16 bit

WIN32.MAK
Project file for Microsoft Visual C 32 bit

M32.DM DMAKE makefile for Microsoft Visual C 32 bit

WIN16.IDE
Project file for Borland C 16 bit

B16.DM DMAKE makefile for Borland C 16 bit

WIN32.IDE
Project file for Borland C 32 bit

B32.DM DMAKE makefile for Borland C 32 bit

S32.PRJ & S32.OPN
Project files for Symantec C 32 bit (requires version 7.21 or later)

S32.DM DMAKE makefile for Symantec C 32 bit (requires version 7.21 or later)

Once an application is built, the only file needed to run it is `MAIN.EXE`.

3.7 Building The Examples

In order to build an example with an IDE, simply open the appropriate project file and select build.

In order to build with the DMAKE command line utility execute the following command.

```
dmake -f MAKEFILE.DM
```

where `MAKEFILE.DM` is one of the DMAKE makefiles listed above.

Alternatively, you can use the following command to create a debug version of the example:

```
dmake -f MAKEFILE.DM  DEBUG=1
```

3.8 Debugging With The IDE

Due to the fact that the WDS is the first thing which Windows sees when it executes an application and the fact that WDS is not shipped with debugging information, the following procedure will make it easier to debug your application code.

First compile your application with debugging information. Then when you wish to debug the application, set a break point at "start". You can then debug your application by telling the debugger to run to the first break point. You will then see the initial function of your application and may debug as usual from that point.

If, however, you purchased the WDS source code, you may compile it with full debugging information and debug through the WDS code as well as your application specific code.

3.9 Building Your Own Application

The best method of building your own application would be to start with one of the example programs and proceed from there. This way all the compiler, linker and other options will be preset.

If you plan to create your own Dynace classes along with your WDS application, it would be best for you to start with the WDS example which illustrates the creation of a Dynace class along with the WDS application. This example has the appropriate make file logic to handle custom classes along with a WDS application.

3.10 DMAKE

DMAKE is a very powerful, portable and enhanced make utility developed by Dennis Vadura. It is freely distributable and not owned by Algorithms Corporation or Blake McBride. Complete documentation for this utility is contained in `\DYNACE\DOCS\DMAKE.MAN`

3.11 DPP

`DPP` is the pre-processor used by Dynace to convert class definition files into C source files. It is also used to generate generic files and perform generic/method argument checking. This utility is only needed when creating new classes and is only used by more advanced WDS users who wish to create their own Dynace classes.

This utility is fully documented in the Dynace language manual.

4 Library Reference

This chapter gives a detailed description of each WDS class. It is organized by facility so that in order to look something up, you would first go to the section which deals with the facility you are interested in. And then the functions are in alphabetical order.

There is a naming convention used with Dynace generics. All generics start with either a lower case "g", "v" or "m", and are always followed by an upper case letter. The ones which start with "g" are normal generics and may be treated like normal C functions. The ones which begin with "v" use the variable argument facilities of C and you should, therefore, take a bit extra care when using them since there is no compile time argument checking being done with these functions.

Since all generics start with either "g", "v" or "m" and in order to avoid the difficulty associated with grouping all the generics under three letters, the first letter is dropped for indexing or heading purposes. Therefore if you are looking up a generic, it will always appear in the index or header with its first letter missing. The syntax description and example code, however, will show the entire name.

There is one thing about Dynace which you must be aware of, however. Dynace generic functions (which is what most of these functions are) allow the same function to be called in many circumstances. These generic functions dynamically dispatch to the appropriate procedures based on the type of the first argument to the generic function (or "generic" for short). So even though you are calling the same function name, entirely different functionality may occur based on the type of the first argument. This fact is seldom, if ever, a problem since you know whether you are dealing with an icon or a dialog, for example, at any given point. Just be sure to look up the generic in the appropriate section.

4.1 A Note To Dynace Language Users

In an effort to hide unnecessary details associated with class vs. instance methods from WDS users who don't care about the differences, these differences are not discussed or grouped as they are in the Dynace manual. You can always distinguish between the two by what the method takes as its first argument. If it's a class, then you have a class method, otherwise it's an instance method.

If you are a WDS user with interest in the Dynace object oriented extension to C, it is fully documented in its associated manual.

4.2 Class Hierarchy

This Dynace Windows Development System contains the following class hierarchy:

Dynace Windows Development System Class Hierarchy

(Object)

- Application
- (Stream)
  - Window
    - MainWindow
    - PopupWindow
      - DisplayWindow
      - StatusWindow
      - SplashWindow
    - ChildWindow
      - ToolBar
      - StatusBar
    - ButtonWindow
    - Control
      - SpinControl
        - TextControl
        - NumericControl
        - DateControl
      - PushButton
      - CheckBox
      - RadioButton
      - ListBox
        - DirListBox
      - ComboBox
      - ScrollBar
      - StaticControl
      - CustomControl
  - Printer
- Menu
  - ExternalMenu
  - InternalMenu
- PopupMenu
- Dialog
  - ModalDialog
  - ModelessDialog
- HelpSystem
- Cursor
  - SystemCursor
  - ExternalCursor
- TaskList
- Task
  - DialogTask
- Icon
  - SystemIcon
  - ExternalIcon
- Font
  - SystemFont
  - ExternalFont
- Brush
  - StockBrush
  - SolidBrush
    - SystemBrush
  - HatchBrush
- Pen
  - StockPen
  - CustomPen
- CommonDialog
  - FileDialog
  - PrintDialog
  - ColorDialog
  - FontDialog
- DynamicLibrary
- Database (ODBC)
- Statement
- StatementInfo
- TableInfo
- ColumnInfo
- TableListbox
- VirtualListbox
- ComClient (COM/DCOM/OLE)
- ComInstance
- ComInterface
- ComServer
- OLEDispatch

4.3 Application

The `Application` class is the class used to control application wide defaults such as fonts, cursors, brushes and so on. It is also the class which gets executed by Windows first and executes your application specific `start` function.

There are no instance methods associated with the `Application` class. All access to this class is through class methods via the global class object `Application`.

`CmdLine::Application` [CmdLine]

```
cl = gCmdLine(Application);

char    *cl;    /*  command line   */
```

This method is used to gain access to command line which was used when the application was launched.

Example:

```
int     cl;

cl = gCmdLine(Application);
```

See also: `Instance`, `Show`, `PrevInstance`

`Error::Application` [Error]

```
gError(obj, msg);

object  obj;    /*  any object     */
char    *msg;   /*  error message  */
```

This method is used in a severe error condition to issue an error message to the user and terminate the application. This method should be avoided if possible.

Note that this method is actually associated with the Dynace `Object` class which is why the first argument may be any Dynace object.

Example:

```
gError(Application, "Error message");
```

See also: `QuitApplication`

`GetBackBrush::Application` [GetBackBrush]

```
bo = gGetBackBrush(Application);

object  bo;     /*  brush object  */
```

This method is used to obtain a copy of the default background brush object associated with the application. The returned object will be an instance of one of the subclasses of the `Brush` class.

The object returned must, either explicitly or implicitly, be disposed when it is no longer needed. This is normally done automatically by WDS when it is associated with a window.

See the `Brush` class and its subclasses for further details.

Example:

```
object  bo;

bo = gGetBackBrush(Application);
```

See also: `GetTextBrush, SetBackBrush`

`GetCursor::Application` [GetCursor]

```
co = gGetCursor(Application);

object  co;     /*  cursor object  */
```

This method is used to obtain a copy of the default cursor object associated with the application. The returned object will be an instance of one of the subclasses of the `Cursor` class.

The object returned must, either explicitly or implicitly, be disposed when it is no longer needed. This is normally done automatically by WDS when it is associated with a window.

See the `Cursor` class and its subclasses for further details.

Example:

```
object  co;

co = gGetCursor(Application);
```

See also: `SetCursor`

GetFont::Application [GetFont]

```
fo = gGetFont(Application);

object  fo;     /*  font object  */
```

This method is used to obtain a copy of the default font object associated with the application. The returned object will be an instance of one of the subclasses of the `Font` class.

The object returned must, either explicitly or implicitly, be disposed when it is no longer needed. This is normally done automatically by WDS when it is associated with a window.

See the `Font` class and its subclasses for further details.

Example:

```
object  fo;

fo = gGetFont(Application);
```

See also: `SetFont`

GetIcon::Application [GetIcon]

```
io = gGetIcon(Application);

object  io;     /*  icon object  */
```

This method is used to obtain a copy of the default icon object associated with the application. The returned object will be an instance of one of the subclasses of the `Icon` class.

The object returned must, either explicitly or implicitly, be disposed when it is no longer needed. This is normally done automatically by WDS when it is associated with a window.

See the `Icon` class and its subclasses for further details.

Example:

```
object  io;

io = gGetIcon(Application);
```

See also: `SetIcon`

`GetName::Application` [GetName]

```
nm = gGetName(Application);

char    *nm;    /*  application name  */
```

This method is used to obtain the global name associated with the application.

Example:

```
char    *nm;

nm = gGetName(Application);
```

See also: `SetName`

`GetTextBrush::Application` [GetTextBrush]

```
bo = gGetTextBrush(Application);

object  bo;     /*  brush object  */
```

This method is used to obtain a copy of the default text brush object associated with the application. The returned object will be an instance of one of the subclasses of the `Brush` class.

The object returned must, either explicitly or implicitly, be disposed when it is no longer needed. This is normally done automatically by WDS when it is associated with a window.

See the `Brush` class and its subclasses for further details.

Example:

```
object  bo;

bo = gGetTextBrush(Application);
```

See also: `GetBackBrush, SetTextBrush`

`GetScalingMode::Application` [GetScalingMode]

```
m = gGetScalingMode(Application);

int     m;      /*  scaling mode   */
```

This method is used to get the current scaling mode in affect. See `SetScalingMode` for further details.

Example:

```
int     mode;

mode = gGetScalingMode(Application);
```

See also: `SetScalingMode`, `ScaleToPixels`, `ScaleToCurrentMode`

`GetSize::Application` [GetSize]

```
r = gGetSize(Application, vert, horz);

int    *vert;   /*  vertical size    */
int    *horz;   /*  horizontal size  */
object  r;      /*  Application      */
```

This method is used to get the total size of the user's screen. `vert` and `horz` are in increments dictated by the mode selected by `SetScalingMode`.

Example:

```
int     y, x;

gGetSize(Application, &y, &x);
```

See also: `SetScalingMode`, `GetSize::Window`

`Instance::Application` [Instance]

```
ins = gInstance(Application);

HINSTANCE  ins;    /*  instance handle   */
```

This method is used to gain access to the Windows instance handle associated with the application. This handle is mainly used internally by Windows and WDS, and should not normally be needed by a WDS programmer.

Example:

```
HINSTANCE  h;

h = gInstance(Application);
```

See also: `PrevInstance, CmdLine, Show`

`PrevInstance::Application` [PrevInstance]

```
ins = gPrevInstance(Application);

HINSTANCE  ins;    /*  instance handle   */
```

This method is used to gain access to the Windows instance handle associated with the previous instance of this application, should more than one be running. This handle is mainly used internally by Windows and WDS, and should not normally be needed by a WDS programmer.

This value will always be `NULL` under Windows NT.

Example:

```
HINSTANCE  h;

h = gPrevInstance(Application);
```

See also: `Instance, CmdLine, Show`

`QuitApplication::Application` [QuitApplication]

```
r = gQuitApplication(Application, ret);

int     ret;    /*  app return value  */
object  r;      /*  Application       */
```

This method is used to terminate an application. The value passed will be used as the return value of the application.

Example:

```
gQuitApplication(Application, 0);
```

See also: `Error`

ScaleToCurrentMode::Application [ScaleToCurrentMode]

```
r = gScaleToCurrentMode(Application, y, x, fnt);

int     *y;     /*  row position         */
int     *x;     /*  column position      */
object  fnt;    /*  current font object  */
object  r;      /*  Application          */
```

This method is used to convert coordinates from pixels to the current scaling mode. It is mainly used internally by WDS in order to convert Window's standard pixel positions into the current scaling mode.

On entry `x` and `y` point to values which are coordinates in terms of pixels. After this method returns, their value will be changed to be in terms of the current scaling mode (see `SetScalingMode`).

`fnt` is a font object which is used only if the current scaling mode is relative to a font. If so, the font it will be related to will be the one passed.

Example:

```
int     y, x;
object  fnt;

y = some position;
x = some position;
fnt = some font object;
gScaleToCurrentMode(Application, &y, &x, fnt);
```

See also: `SetScalingMode`, `GetScalingMode`, `ScaleToPixels`

ScaleToPixels::Application [ScaleToPixels]

```
r = gScaleToPixels(Application, y, x, fnt);

int     *y;     /*  row position         */
int     *x;     /*  column position      */
object  fnt;    /*  current font object  */
object  r;      /*  Application          */
```

This method is used to convert coordinates from the current scaling mode to pixels. It is mainly used internally by WDS in order to convert your coordinates into standard pixel positions.

On entry `x` and `y` point to values which are coordinates in terms of the current scaling mode (see `SetScalingMode`). After this method returns, their value will be changed to be in terms of pixels.

`fnt` is a font object which is used only if the current scaling mode is relative to a font. If so, the font it will be related to will be the one passed.

Example:

```
int     y, x;
object  fnt;

y = some position;
x = some position;
fnt = some font object;
gScaleToPixels(Application, &y, &x, fnt);
```

See also: `SetScalingMode, GetScalingMode, ScaleToCurrentMode`

`SetBackBrush::Application` [SetBackBrush]

```
r = gSetBackBrush(Application, bo);

object  bo;     /*  brush object         */
object  r;      /*  brush object passed  */
```

This method is used to set the application wide default background brush. This is the brush used to color everything except the text that appears on windows and dialogs. `bo` must be an instance of one of the subclasses of `Brush`. All windows and dialogs will use the application wide default brush which is in effect when they are created unless a specific brush object is specified for a particular window.

When a new default brush object is set, any previous default object will be disposed. If no default is set, WDS uses the system brush identified as `COLOR_WINDOW`.

See the `Brush` class and its subclasses for further details.

Example:

```
gSetBackBrush(Application, vNew(SystemBrush, COLOR_WINDOW));
```

See also: `GetBackBrush, SetTextBrush`

SetCursor::Application [SetCursor]

```
r = gSetCursor(Application, co);

object  co;     /*  cursor object         */
object  r;      /*  cursor object passed  */
```

This method is used to set the application wide default cursor. `co` must be an instance of one of the subclasses of `Cursor`. All windows will use the application wide default cursor which is in effect when they are created unless a specific cursor object is specified for a particular window.

When a new default cursor object is set, any previous default object will be disposed. If no default is set, WDS uses the system cursor identified as `IDC_ARROW`.

See the `Cursor` class and its subclasses for further details.

Example:

```
gSetCursor(Application, gLoadSys(SystemCursor, IDC_ARROW));
```

See also: `GetCursor`, `LoadSys::SystemCursor`, `Load::ExternalCursor`

SetFont::Application [SetFont]

```
r = gSetFont(Application, fo);

object  fo;     /*  font object         */
object  r;      /*  font object passed  */
```

This method is used to set the application wide default font. `fo` must be an instance of one of the subclasses of `Font`. All windows will use the application wide default font which is in effect when they are created unless a specific font object is specified for a particular window.

When a new default font object is set, any previous default object will be disposed. If no default is set, WDS uses the system font identified as `SYSTEM_FONT`.

See the `Font` class and its subclasses for further details.

Example:

```
gSetFont(Application, vNew(SystemFont, SYSTEM_FONT));
```

See also: `GetFont`, `Load::SystemFont`, `New::ExternalFont`

SetIcon::Application [SetIcon]

```
r = gSetIcon(Application, io);

object  io;     /*  icon object         */
object  r;      /*  icon object passed  */
```

This method is used to set the application wide default icon. An icon associated to a window is the one which is displayed when the window is iconized. `io` must be an instance of one of the subclasses of `Icon`. All windows will use the application wide default icon which is in effect when they are created unless a specific icon object is specified for a particular window.

When a new default icon object is set, any previous default object will be disposed. If no default is set, WDS uses the system icon identified as `IDI_APPLICATION`.

See the `Icon` class and its subclasses for further details.

Example:

```
gSetIcon(Application, gLoadSys(SystemIcon, IDI_APPLICATION));
```

See also: `GetIcon`, `LoadSys::SystemIcon`, `Load::ExternalIcon`

SetName::Application [SetName]

```
gSetName(Application, nm);

char    *nm;    /*  application name  */
```

This method is used to give the application a globally accessible name. This name is accessible via `GetName`. There is no other use made of this information.

Example:

```
gSetName(Application, "My App");
```

See also: `GetName`

SetScalingMode::Application [SetScalingMode]

```
r = gSetScalingMode(Application, m);

int     m;      /*  scaling mode   */
int     r;      /*  previous mode  */
```

This method is used to set the scaling mode used by all other WDS methods which take coordinate positions. Valid modes are as follows:

`SM_1_PER_CHAR`
: This mode causes each position to be in increments determined by the size of the current font. For example row 7 would mean 7 times the height of the current font. Similar to line positions.

`SM_10_PER_SYSCHAR`
: This mode causes each position to be in increments determined by one tenth the size of the system font declared globally to the user's Windows environment. For example row 70 would mean 7 times the height of the Windows global system font. This allows positioning relative to a scaling factor determined by the user.

`SM_PIXELS`
: This mode performs no conversion. The application is able to use pixel coordinated directly.

The default value is `SM_1_PER_CHAR`.

The value returned is the mode which was previously set.

Example:

```
gSetScalingMode(Application, SM_PIXELS);
```

See also: `GetScalingMode`, `ScaleToPixels`, `ScaleToCurrentMode`

`SetTextBrush::Application` [SetTextBrush]

```
r = gSetTextBrush(Application, bo);

object  bo;     /*  brush object         */
object  r;      /*  brush object passed  */
```

This method is used to set the application wide default text brush. This is the brush used by all text output (or foreground) to windows or dialogs. `bo` must be an instance of one of the subclasses of `Brush`. All windows and dialogs will use the application wide default brush which is in effect when they are created unless a specific brush object is specified for a particular window.

When a new default brush object is set, any previous default object will be disposed. If no default is set, WDS uses the system brush identified as `COLOR_WINDOWTEXT`.

See the `Brush` class and its subclasses for further details.

Example:

```
gSetTextBrush(Application, vNew(SystemBrush, COLOR_WINDOWTEXT));
```

See also: `GetTextBrush`, `SetBackBrush`

`Show::Application` [Show]

```
sv = gShow(Application);

int     sv;     /*  show value   */
```

This method is used to gain access to the show value supplied by Windows when an application starts. It is normally used to determine how an initial application's main window should be shown. The available options are Windows macros which begin with `SW_` and are fully documented by the Windows documentation under the `WinMain` function.

Example:

```
int     sv;

sv = gShow(Application);
```

See also: `Instance`, `CmdLine`, `PrevInstance`

4.4 Windows

This section documents the `Window` class which contains all the functionality which is common to Main, Child, and Popup windows. See chapter 2 of this manual for a description of the different window types.

`AddHandlerAfter::Window` [AddHandlerAfter]

```
r = gAddHandlerAfter(wind, msg, func);

object   wind;     /*  a window object  */
unsigned msg;      /*  message          */
long    (*func)(); /*  function pointer */
object  r;         /*  the window obj   */
```

This method is used to associate function `func` with Windows window message `msg` for window `wind`. Whenever window `wind` receives message `msg`, `func` will be called.

`wind` is the window object who's messages you wish to process. `msg` is the particular message you wish to trap. These messages are fully documented in the Windows documentation in the Messages section. They normally begin with `WM_`.

`func` is the function which gets called whenever the specified message gets received and takes the following form:

```
long    func(object     wind,
             HWND       hwnd,
             UINT       mMsg,
             WPARAM     wParam,
             LPARAM     lParam)
{
        .
        .
        .
        return 0L;  /* or whatever is appropriate  */
}
```

Where `wind` is the window being sent the message. The remaining arguments and return value is fully documented in the Windows documentation under the `WindowProc` function and the Windows Messages documentation.

WDS keeps a list of functions associated with each message associated with each window. When a particular message is received the appropriate list of handler functions gets executed sequentially. `AddHandlerAfter` appends the new function to the end of this list, and `AddHandlerBefore` adds the new function to the beginning of the list.

WDS may also, and optionally, execute the Windows default procedure associated with a given message either before or after the user added list of functions. This behavior may be controlled via `DefaultProcessingMode`.

Windows will only see the return value of the last message handler executed including, if applicable, the default.

Example:

```
int     hSize, vSize;

static  long    process_wm_size(object  wind,
                                HWND    hwnd,
                                UINT    mMsg,
                                WPARAM  wParam,
                                LPARAM  lParam)
{
        hSize = LOWORD(lParam);
        vSize = HIWORD(lParam);
        return 0L;
}

        .
        .
        gAddHandlerAfter(wind, (unsigned) WM_SIZE, process_wm_size);
        .
        .
```

See also: `DefaultProcessingMode, AddHandlerBefore`

`AddHandlerBefore::Window` [AddHandlerBefore]

```
r = gAddHandlerBefore(wind, msg, func);

object   wind;     /*  a window object  */
unsigned msg;      /*  message          */
long    (*func)(); /*  function pointer */
object  r;         /*  the window obj   */
```

This function is fully documented under `AddHandlerAfter`.

See also: `AddHandlerAfter`

`Associate::Window` [Associate]

```
r = mAssociate(wind, itm, fun);

object  wind;   /*  a window object     */
int     itm;    /*  menu item           */
long    (*fun)();  /*  function         */
object   r;        /*  the menu         */
```

This method is used to associate an application specific function (`fun`) with a menu item (`itm`) associated with the current menu attached to window `wind`. Once this is done, if the user selects the menu option identified by `itm`, then the function `fun` will be executed.

`itm` is a programmer defined macro which identifies one particular choice among those available within the menu which is currently attached to window `wind`. This macro is defined while the programmer defines the entire menu using the resource editor.

`fun` is the function which will be executed when the user selects menu option `itm` and has the following form:

```
long    fun(object wind, unsigned id)
{
        .
        .
        .
        return 0L;
}
```

The function executed (`fun`) is passed the window object and the specific menu id which the user selected and returns a long. The return value is documented in the

Windows documentation under the message named `WM_COMMAND`. It should normally be `0L`.

Example:

```
static  long    file_message(object wind)
{
        gMessage(wind, "File_Message");
        return 0L;
}

        .
        .
        mLoadMenu(win, IDR_MENU1);
        mAssociate(win, ID_FILE_MESSAGE, file_message);
```

See also: `LoadMenu`, `MenuItemMode`

`AutoDispose::Window` [AutoDispose]

```
r = gAutoDispose(wind, mode);

object   wind;     /*  a window object    */
int      mode;     /*  auto dispose mode  */
int      r;        /*  previous value     */
```

This method is used to enable or disable the auto dispose mechanism associated with a window. This feature, when enabled, causes WDS to automatically dispose of the WDS object associated with a window whenever the user closes the window (`wind`). Normally, when this feature is disabled, if the user closes a window, WDS removes the window from the display, but the WDS object remains intact until the program manually disposes of the window object (via `Dispose`). This feature is most commonly used with asynchronous, popup windows.

Note that it is invalid to attempt to use a WDS object subsequent to it being disposed, and each unneeded WDS object must be deleted.

`mode` is set to `1` to enable the feature and `0` to disable it. The value returned is the prior mode.

Example:

```
gAutoDispose(wind, 1);
```

See also: `Dispose`

`AutoShow::Window` [AutoShow]

```
r = gAutoShow(wind, mode);

object   wind;     /*  a window object    */
int      mode;     /*  auto show mode     */
int      r;        /*  previous value     */
```

This method is used to enable or disable the auto show mechanism associated with a window. This feature, when enabled, causes WDS to automatically display (execute `Show`) on a window which is written to. This enables the programmer to create a window which will automatically popup the first time the program displays text to it.

`mode` is set to `1` to enable the feature and `0` to disable it. The value returned is the prior mode.

Example:

```
gAutoShow(wind, 1);
```

See also: `Show`

`BackBrush::Window` [BackBrush]

```
r = gBackBrush(wind, brsh);

object  wind;   /*  a window object   */
object  brsh;   /*  brush object      */
object  r;      /*  wind              */
```

This method is used to determine what brush object is used for the background of window `wind` and performs the same function as the `Use` method when used with a `Brush` object. Any previously associated brush object will be disposed. This brush object will also be automatically disposed when the window is disposed.

The window passed is returned.

Example:

```
object  myWind;

gBackBrush(myWind, vNew(SolidBrush, 0, 0, 255));
```

See also: `TextBrush`, `Use`, `SetTextBrush::Application` and the `Brush` classes

`DefaultProcessingMode::Window` [DefaultProcessingMode]

```
r = gDefaultProcessingMode(wind, msg, mode);

object   wind;     /*  a window object          */
unsigned msg;      /*  message                  */
int      mode;     /*  default processing mode  */
object   r;        /*  the window obj           */
```

This method is used to determine when or if the Windows default message procedure is processed for a given message (`msg`) associated with a particular window (`wind`).

WDS allows a programmer to specify an arbitrary number of functions to be executed whenever a window receives a specific message (via `AddHandlerAfter` and `AddHandlerBefore`). Windows has default procedures associated with many window messages. At times it is necessary to replace or augment this default functionality. `DefaultProcessingMode` gives the programmer control over when and if this default Windows functionality. `mode` is used to specify the desired mode. The following table indicates the valid modes:

`0`	Do not execute the Windows default processing
`1`	Execute default processing *after* programmer defined handlers
`2`	Execute default processing *before* programmer defined handlers

Note that the default mode is always `1`, and must be explicitly changed, if desired, for each message associated with each window.

`msg` is the particular message you wish to affect. These messages are fully documented in the Windows documentation in the Messages section. They normally begin with `WM_`.

Example:

```
gDefaultProcessingMode(wind, (unsigned) WM_SIZE, 0);
```

See also: `AddHandlerAfter`

`Dispose::Window` [Dispose]

```
r = gDispose(wind);

object  wind;   /*  a window object    */
object  r;      /*  NULL               */
```

This method is used to remove and dispose of a window object when it is no longer needed. This method should be called on all windows when they are no longer needed.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  myWind;

myWind = gDispose(myWind);
```

See also: `AutoDispose`, `New::ChildWindow`, `New::PopupWindow`

`Erase::Window` [Erase]

```
r = gErase(wind, brow, erow, bcol, ecol);

object  wind;   /*  a window object     */
int     brow;   /*  beginning row       */
int     erow;   /*  ending row          */
int     bcol;   /*  beginning column    */
int     ecol;   /*  ending column       */
object  r;      /*  the window object   */
```

This method is used to delete all text vectors which intersect the rectangle defined by the parameters. These parameters are adjusted according to the current scaling mode (set with `SetScalingMode::Application`. All coordinates are measured from the upper left hand corner of the window and start with `0,0`.

The object returned is the window object passed.

Example:

```
gErase(win, 3, 10, 0, 200);
```

See also: `EraseLines`, `EraseAll`

`EraseAll::Window` [EraseAll]

```
r = gEraseAll(wind);

object  wind;   /*  a window object     */
object  r;      /*  the window object   */
```

This method is used to erase all text lines associated with a window.

The object returned is the window object passed.

Example:

```
gEraseAll(win);
```

See also: `EraseLines`, `Erase`

`EraseLines::Window` [EraseLines]

```
r = gEraseLines(wind, blin, elin);

object  wind;   /*  a window object     */
int     blin;   /*  beginning line      */
int     elin;   /*  ending line         */
object  r;      /*  the window object   */
```

This method is used to delete all text lines from a starting line number (`blin`) to an ending line number (`elin`). All lines are measured from the upper left hand corner of the window and start with `0,0`. Lines positions are calculated based on the current font.

The object returned is the window object passed.

Example:

```
gEraseLines(win, 3, 10);
```

See also: `Erase`, `EraseAll`

`Getch::Window` [Getch]

```
ch = gGetch(wind);

object  wind;   /*  a window object   */
int     ch;     /*  character read    */
```

This method is used to obtain the next input character struck by the user. The `SetBlock` method may be used to determine whether or not `Getch` waits if a character is not available. The value returned is the character struck by the user or 0 if no character was available and the input was non-blocking.

Example:

```
object  myWind;
int     ch;

ch = gGetch(myWind);
```

See also: `Kbhit`, `SetBlock`, `Gets`

`GetName::Window` [GetName]

```
r = gGetName(w);

object  w;      /*  a window object     */
char   *r;      /*  the name associated with the window   */
```

This method is used to get the name associated with window `w`. The name associated with a window is what is displayed at the top of the window, if it has a title bar.

Example:

```
object  myWind;
char    *n;

myWind = vNew(MainWindow, "App Name");
n = gGetName(myWind);   /*  n = "App Name"  */
```

See also: `New::MainWindow`, `SetName`

`GetParent::Window` [GetParent]

```
prnt = gGetParent(wind);

object  wind;   /*  child window object   */
object  prnt;   /*  parent window object  */
```

This method is used to obtain the parent window object associated with window `wind`.

Example:

```
object  myWind, parentWind;

parentWind = gGetParent(myWind);
```

See also: `SetParent`, `New::ChildWindow`

`GetPosition::Window` [GetPosition]

```
r = gGetPosition(wind, vert, horz);

object  wind;   /*  a window object     */
int    *vert;   /*  vertical position   */
int    *horz;   /*  horizontal position */
object  r;      /*  wind arg passed     */
```

This method is used to get the current position of a window. `vert` and `horz` are in increments dictated by the mode selected by `SetScalingMode::Application`. The window object passed is returned.

Example:

```
int     y, x;

gGetPosition(myWind, &y, &x);
```

See also: `SetScalingMode::Application`, `GetSize`, `SetPosition`

`Gets::Window` [Gets]

```
r = gGets(wind, buf, len);

object   wind;  /*  a window object       */
char    *buf;   /*  input buffer          */
unsigned len;   /*  length of buffer      */
char    *r;     /*  buf                   */
```

This method is used to read (accept) a single line of text from a user until a return is hit or `len`-1 key strokes have been entered and place them in `buf`.

This method returns a pointer to the buffer passed unless there is an error, in which case `NULL` is returned. The input accepted will a return terminated line entered by the user up to `len`-1 characters. However, the number of characters may be less if non-blocking io is selected (via `SetBlock`) and there aren't enough characters available. `SetRaw` may also be used to control whether backspace processing will be performed.

Example:

```
object  myWind;
char    buf[80];

gGets(myWind, buf, sizeof(buf));
```

See also: `SetBlock`, `SetRaw`, `Gets`, `Getch`

`GetSize::Window` [GetSize]

```
r = gGetSize(wind, vert, horz);

object  wind;   /*  a window object     */
int    *vert;   /*  vertical size       */
int    *horz;   /*  horizontal size     */
object  r;      /*  wind arg passed     */
```

This method is used to get the current size of a window. `vert` and `horz` are in increments dictated by the mode selected by `SetScalingMode::Application`. The window object passed is returned.

Example:

```
int     y, x;

gGetSize(myWind, &y, &x);
```

See also: `SetScalingMode::Application`, `GetPosition`, `SetSize`, `GetSize::Application`

`GetTag::Window` [GetTag]

```
r = gGetTag(wind);

object  wind;   /*  a window object   */
object  r;      /*  tag               */
```

This method is used to obtain a Dynace object which has been associated with a window via `SetTag`. The value return is the object which has been associated with the window object `wind`. If there is no object associated with the window, `NULL` will be returned.

Example:

```
object  myWind, someObj;

someObj = gGetTag(myWind);
```

See also: `SetTag`, `SetTag::Dialog`

Handle::Window [Handle]

```
h = gHandle(wind);

object  wind;   /*  window object  */
HANDLE  h;      /*  Windows handle */
```

This method is used to obtain the Windows internal handle associated with a window object. Note that this will be `0` prior to executing `Show` on the window because that is the point where Windows creates the handle.

Note that this method may be used with most WDS objects in order to obtain the internal handle that Windows normally associates with each type of object. See the appropriate documentation.

Example:

```
object  myWind;
HANDLE  h;

h = gHandle(myWind);
```

Kbhit::Window [Kbhit]

```
r = gKbhit(wind);

object  wind;   /*  a window object   */
int     r;      /*  characters ready  */
```

This method is used to determine if and how many characters are waiting in the character input queue. This method is used in an attempt to provide a familiar character based user input mechanism.

The value returned is the number of characters ready for input or zero if none are available.

Example:

```
object  myWind;
int     n;

n = gKbhit(myWind);
```

See also: `Getch`, `SetBlock`, `Gets`

`LoadCursor::Window` [LoadCursor]

```
r = mLoadCursor(wind, csr);

object   wind;  /*  a window object    */
unsigned csr;   /*  cursor identifier  */
object   r;     /*  cursor object      */
```

This method is used to load a programmer defined cursor and associate it with window `wind`. The cursor would then be displayed any time the pointer was placed in the window.

`csr` is a programmer defined unsigned integer which identifies the cursor. This identifier is normally a macro and defined through the resource editor. The cursor object will be automatically destroyed whenever the window is destroyed or if a new cursor is loaded.

The value returned is an object representing the cursor loaded, or `NULL` if the cursor was not found.

Example:

```
object  myWind;

mLoadCursor(myWind, MY_CURSOR);
```

See also: `LoadSystemCursor`, `Load::ExternalCursor`, `Use`

`LoadFont::Window` [LoadFont]

```
r = gLoadFont(wind, fname, sz);

object   wind;  /*  a window object  */
char    *fname; /*  font name        */
int      sz;    /*  point size       */
object   r;     /*  font object      */
```

This method is used to load an arbitrary font by name at any point size and associate it as the default font for future text output to window `wind`. `fname` is the full name of the font as it appears when you list the available fonts via the control-panel / fonts Windows utility, minus the font type in parentheses. `sz` indicates the desired point size.

All font objects associated with a window will be automatically destroyed whenever the window is destroyed.

The value returned is an object representing the font loaded, or `NULL` if the font was not found.

Note that `fname` may also be a Dynace object cast as a (`char *`).

Example:

```
object  myWind;

gLoadFont(myWind, "Times New Roman", 12);
```

See also: `LoadSystemFont`, `Indirect::ExternalFont`, `Use`

`LoadIcon::Window` [LoadIcon]

```
r = mLoadIcon(wind, icn);

object   wind;  /*  a window object  */
unsigned icn;   /*  icon identifier  */
object   r;     /*  icon object      */
```

This method is used to load a programmer defined icon and associate it with window `wind`. The icon would then be displayed if window `wind` was iconized.

`icn` is a programmer defined unsigned integer which identifies the icon. This identifier is normally a macro and defined through the resource editor. The icon object will be automatically destroyed whenever the window is destroyed or if a new icon is loaded.

The value returned is an object representing the icon loaded, or `NULL` if the icon was not found.

Example:

```
object  myWind;

mLoadIcon(myWind, ALGOCORP_ICON);
```

See also: `LoadSystemIcon`, `Load::ExternalIcon`, `Use`

`LoadMenu::Window` [LoadMenu]

```
r = mLoadMenu(wind, mnu);

object   wind;  /*  a window object  */
unsigned mnu;   /*  menu identifier  */
object   r;     /*  menu object      */
```

This method is used to load a programmer defined menu and associate it with window `wind`. The menu would be displayed at the top of the window.

`mnu` is a programmer defined unsigned integer which identifies the menu. This identifier is normally a macro and defined through the resource editor. Any menu previously associated with the window is pushed on a stack so that previous menus may be easily returned to in a last-in-first-out basis using the `PopMenu` method. All menus associated with a window will be destroyed when the window is destroyed.

The value returned is an object representing the menu loaded, or `NULL` if the menu was not found.

Example:

```
object  myWind;

mLoadMenu(myWind, MY_MENU);
```

See also: `Associate, PopMenu, LoadMenuStr, Load::ExternalMenu, Use`

`LoadMenuStr::Window` [LoadMenuStr]

```
r = gLoadMenuStr(wind, mnu);

object   wind;  /*  a window object  */
char    *mnu;   /*  menu identifier  */
object   r;     /*  menu object      */
```

This method is used to load a programmer defined menu and associate it with window `wind`. The menu would be displayed at the top of the window.

`mnu` is a programmer defined name which identifies the menu. This name is defined through the resource editor. Any menu previously associated with the window is pushed on a stack so that previous menus may be easily returned to in a last-in-first-out basis using the `PopMenu` method. All menus associated with a window will be destroyed when the window is destroyed.

Note that the `LoadMenu` method is used more frequently since the resource editors assign macros to each menu name.

The value returned is an object representing the menu loaded, or `NULL` if the menu was not found.

Example:

```
object  myWind;

gLoadMenuStr(myWind, "mymenu");
```

See also: `Associate, LoadMenu, LoadStr::ExternalMenu, Use`

LoadSystemCursor::Window [LoadSystemCursor]

```
r = gLoadSystemCursor(wind, csr);

object   wind;  /*  a window object    */
LPCSTR   csr;   /*  cursor identifier  */
object   r;     /*  cursor object      */
```

This method is used to load a Windows predefined cursor and associate it with window `wind`. The cursor would then be displayed if the pointer was positioned over the window.

`csr` is a Windows defined macro which identifies the cursor. The available options are defined in the Windows documentation under the function named `LoadCursor` and normally begin with `IDC_`. The cursor object will be automatically destroyed whenever the window is destroyed or if a new cursor is loaded.

The value returned is an object representing the cursor loaded, or `NULL` if the cursor was not found.

Example:

```
object  myWind;

gLoadSystemCursor(myWind, IDC_CROSS);
```

See also: `LoadCursor`, `LoadSys::SystemCursor`, `Use`

LoadSystemFont::Window [LoadSystemFont]

```
r = gLoadSystemFont(wind, fnt);

object   wind;  /*  a window object  */
unsigned fnt;   /*  font identifier  */
object   r;     /*  font object      */
```

This method is used to load a Windows predefined font and associate it with window `wind`. The last font associated with a window is the one which will be used when any text is output to the window.

`fnt` is a Windows defined macro which identifies the font. The available options are defined in the Windows documentation under the function named `GetStockObject` and normally end with `_FONT`. The font object will be automatically destroyed whenever the window is destroyed.

The value returned is an object representing the font, or `NULL` if the font was not found.

Example:

```
object  myWind;

gLoadSystemFont(myWind, SYSTEM_FONT);
```

See also: `LoadFont`, `Load::SystemFont`, `Use`

`LoadSystemIcon::Window` [LoadSystemIcon]

```
r = gLoadSystemIcon(wind, icn);

object   wind;  /*  a window object  */
LPCSTR   icn;   /*  icon identifier  */
object   r;     /*  icon object      */
```

This method is used to load a Windows predefined icon and associate it with window `wind`. The icon would then be displayed if window `wind` was iconized.

`icn` is a Windows defined macro which identifies the icon. The available options are defined in the Windows documentation under the function named `LoadIcon` and normally begin with `IDI_`. The icon object will be automatically destroyed whenever the window is destroyed or if a new icon is loaded.

The value returned is an object representing the icon loaded, or `NULL` if the icon was not found.

Example:

```
object  myWind;

gLoadSystemIcon(myWind, IDI_APPLICATION);
```

See also: `LoadIcon`, `LoadSys::SystemIcon`, `Use`

`MenuItemMode::Window` [MenuItemMode]

```
r = mMenuItemMode(wind, itm, mod);

object  wind;   /*  a window object     */
unsigned itm;   /*  menu item           */
unsigned mod;   /*  menu item mode      */
object   r;        /*  the menu         */
```

This method is used to set the mode associated with a particular item in the menu which is currently attached to window `wind`.

`itm` is a programmer defined macro which identifies one particular choice among those available within the menu which is currently attached to window `wind`. This macro is defined while the programmer defines the entire menu using the resource editor.

`mod` may be one of `MF_DISABLED`, `MF_ENABLED` or `MF_GRAYED` and is documented in the Windows documentation under the function `EnableMenuItem`.

The menu object associated with the window is returned.

Example:

```
mMenuItemMode(win, ID_FILE_MESSAGE, MF_GRAYED);
```

See also: `LoadMenu`, `Associate`

`Message::Window` [Message]

```
r = gMessage(wind, msg);

object  wind;   /*  window object  */
char    *msg;   /*  message        */
object  r;      /*  window object  */
```

This method is used to open up a temporary informational window. The window will contain the message given by `msg` and the user must acknowledge the window prior to continuing by hitting an OK button.

The value returned is the window passed.

Example:

```
object  myWind;

gMessage(myWind, "Press OK to continue.");
```

See also: `MessageWithTopic`

`MessageWithTopic::Window` [MessageWithTopic]

```
r = gMessageWithTopic(wind, msg, tpc);

object  wind;   /*  window object  */
char    *msg;   /*  message        */
char    *tpc;   /*  help topic     */
object  r;      /*  window object  */
```

This method is used to open up a temporary informational window. The window will contain the message given by `msg` and the user must acknowledge the window prior to continuing by hitting an OK button.

If the user hits the `F1` key while presented with the message, the help topic identified by `tpc` will get displayed via the Windows help system.

The value returned is the window passed.

Example:

```
object  myWind;

gMessageWithTopic(myWind, "Press OK to continue.", "mytopic");
```

See also: `Message` and the `HelpSystem` class.

`New::Window` [New]

```
r = vNew(Window);

object  r;      /*  new window  */
```

This class method is used to create a basic window object. It is used by all the subclasses of `Window` and would not normally be used by a programmer.

The value returned is the new window created.

Example:

```
object  myWind;

myWind = vNew(Window);
```

See also: `New::MainWindow`, `New::ChildWindow`, `New::PopupWindow`, `Show` `Dispose`

`NewBuiltIn::Window` [NewBuiltIn]

```
r = gNewBuiltIn(Window, class, parent);

char    *class; /*  Windows built in class designation  */
object  parent; /*  parent window object                */
object  r;      /*  new child window                    */
```

This class method is used to create a child window which is used as a Windows-built-in control. `Window` represents the `Window` class and is typed in as shown. `class`

is a string representing one of the window classes build into Windows. This string is defined and documented by Windows under the function named `CreateWindow`. `parent` represents the parent window object and must be specified.

The value returned is the new child window (control) created.

Example:

```
object  myWind, ctl;

myWind = vNew(MainWindow, "App Name");
ctl = gNewBuiltIn(Window, "button", myWind);
```

See also: `New::ButtonWindow`

`PopMenu::Window` [PopMenu]

```
r = gPopMenu(wind);

object   wind;  /*  a window object  */
object   r;     /*  menu object      */
```

This method is used to remove and destroy the current menu associated with window `wind` and restore the previous menu associated with the window. A last-in-first-out list of menus associated with a window may be established via the `LoadMenu`, `LoadMenuStr` or `Use` methods.

All function associations and modes previously associated with the new menu will be restored.

The object returned is the new menu object.

Example:

```
object  myWind;

gPopMenu(myWind);
```

See also: `LoadMenu, Use`

`Printf::Window` [Printf]

```
r = vPrintf(wind, fmt, ...);

object   wind;  /*  a window object       */
char    *fmt;   /*  format string         */
int      r;     /*  length of output      */
```

This method is used to display a string of text (`str`) in a sequential fashion on window `wind`. It is analogous to the standard C library function `fprintf`. All of the standard features of your C library `fprintf` function are supported and that documentation should be consulted for full documentation on the arguments.

This method is used to support the standard streams interface, is actually defined by the `Stream` class, and is documented here for convenience.

Note that this method begins with "v" since it takes variable arguments. The length of the resulting output will be returned.

Example:

```
object  myWind;
int     age = 32;

vPrintf(myWind, "My age =- %d\n", age);
```

See also: `TextOut, Puts, Write`

`Puts::Window` [Puts]

```
r = gPuts(wind, str);

object   wind;  /*  a window object       */
char    *str;   /*  string to be output   */
int      r;     /*  length of str         */
```

This method is used to display a string of text (`str`) in a sequential fashion on window `wind`. This method is used to support the standard streams interface, is actually defined by the `Stream` class, and is documented here for convenience.

`str` is the text to be displayed. Text `str` will be displayed on window `wind` and its length will be returned.

Example:

```
object  myWind;

gPuts(myWind, "Hello World\n");
```

See also: `TextOut, Printf, Write`

`Read::Window` [Read]

```
r = gRead(wind, buf, len);

object   wind;  /*  a window object       */
char    *buf;   /*  input buffer          */
unsigned len;   /*  length of buffer      */
int      r;     /*  bytes read            */
```

This method is used to read (accept) `len` key strokes from the user and place them in `buf`.

Since the `Window` class is a subclass of `Stream`, this method is mainly provided to support the standard interface dictated by the `Stream` class.

This method returns the number of characters actually accepted. It will not be more than `len`, but may be less if non-blocking io is selected (via `SetBlock`) and there aren't enough characters available. `SetRaw` may also be used to control whether backspace processing will be performed.

Example:

```
object  myWind;
char    buf[80];

gRead(myWind, buf, sizeof(buf)-1);
```

See also: `SetBlock`, `SetRaw`, `Gets`, `Getch`

`ScrollHorz::Window` [ScrollHorz]

```
r = gScrollHorz(wind, cols);

object  wind;   /*  a window object     */
int     cols;   /*  columns to scroll   */
object  r;      /*  the window object   */
```

This method is used to perform a horizontal scrolling of the text in window `wind`. This has the same effect as if the user caused horizontal scrolling via the horizontal scroll bar at the bottom of the window. No text is lost, a different portion of the text is displayed.

`wind` is the window which is to be affected, and `cols` is the number of columns to scroll. If `cols` is positive the scroll moves the text to the left, and negative moves the text to the right.

The scaling factor associated with `cols` is set by `SetScalingMode::Application`.

Example:

```
object  myWind;

gScrollHorz(myWind, 2);
```

See also: `ScrollVert`, `SetScalingMode::Application`

`ScrollVert::Window` [ScrollVert]

```
r = gScrollVert(wind, rows);

object  wind;   /*  a window object     */
int     rows;   /*  rows to scroll      */
object  r;      /*  the window object   */
```

This method is used to perform a vertical scrolling of the text in window `wind`. This has the same effect as if the user caused vertical scrolling via the vertical scroll bar at the right side of the window. No text is lost, a different portion of the text is displayed.

`wind` is the window which is to be affected, and `rows` is the number of rows to scroll. If `rows` is positive the scroll moves the text up, and negative moves the text down.

The scaling factor associated with `rows` is set by `SetScalingMode::Application`.

Example:

```
object  myWind;

gScrollVert(myWind, 2);
```

See also: `ScrollHorz`, `VertShift`, `SetScalingMode::Application`

`SetBlock::Window` [SetBlock]

```
r = gSetBlock(wind, flag);

object  wind;   /*  a window object     */
int     flag;   /*  enable/disable flag */
int     r;      /*  previous flag       */
```

This method is used to set the blocking mode associated with keyboard IO. It only has effect on `Read`, `Gets`, and `Getch`. If flag is 1, blocking is enabled, and 0 disables blocking. When a new window is created it defaults to blocking enabled.

If blocking is turned on and a keyboard entry is requested (via `Read`, `Gets` or `Getch`), the keyboard entry function will not return until the request can be satisfied. If, however, blocking is disabled, a keyboard request is made, and insufficient characters are available, then the input function will return immediately with a return value indicating the result was short.

The value returned by this method is the previous blocking mode.

Example:

```
object  myWind;

gSetBlock(myWind, 0);
```

See also: `Getch, SetRaw, Gets`

`SetMaxLines::Window` [SetMaxLines]

```
r = gSetMaxLines(wind, rows);

object  wind;   /*  a window object     */
int     rows;   /*  max rows            */
int     r;      /*  previous max rows   */
```

This method is used to set or obtain the maximum number of lines of text which may be associated to a window. This includes all lines associated with a window, including lines not being displayed because they are scrolled off the screen. Whenever `rows` number of lines are exceeded, WDS automatically and permanently removes the lines at the logical top of the internal buffer in order to make room for new lines.

WDS keeps a buffer which holds a programmer definable number of lines of text. The user is then able to scroll through this text. If the application attempts to display more lines than this maximum (via `vPrintf` for example) the system will automatically call `gVertShift` in order to eliminate the top line and make room for the new line being appended.

`wind` is the window which is to be affected, and `rows` is the maximum number of rows to retain. If `rows` is zero or negative, the value associated with the window will not be changed. This is used to obtain the current value without changing it.

The value returned is the previous value associated with the window.

Example:

```
object  myWind;

gSetMaxLines(myWind, 60);
```

See also: `VertShift`

`SetName::Window` [SetName]

```
r = gSetName(wind, name);

object  wind;   /*  a window object     */
char   *name;   /*  the new name        */
object  r;      /*  w arg passed        */
```

This method is used to set the name associated with window `wind`. The name associated with a window is what is displayed at the top of the window, if it has a title bar. `name` may also be an `object` typecast to a (`char *`). The value returned is the window (`wind`) object passed.

Example:

```
object  myWind;

myWind = vNew(MainWindow, "Old Name");
gSetName(myWind, "New Name");
```

See also: `New::MainWindow`, `GetName`

`SetParent::Window` [SetParent]

```
r = gSetParent(wind, prnt);

object  wind;   /*  child window object   */
object  prnt;   /*  parent window object  */
object  r;      /*  wind                  */
```

This method is used to create a child / parent window relationship. This relationship is automatically established when a child window is created. However, this method is provided for increased flexibility.

Whenever the parent window is disposed, WDS will automatically dispose of all its children windows. This relationship should normally be established prior to the child window being `Shown`.

The value returned is the child window passed.

Example:

```
object  myWind, parentWind;

gSetParent(myWind, parentWind);
```

See also: `GetParent`

`SetPosition::Window` [SetPosition]

```
r = gSetPosition(wind, vert, horz);

object  wind;   /*  a window object     */
int     vert;   /*  vertical position   */
int     horz;   /*  horizontal position */
object  r;      /*  wind arg passed     */
```

This method is used to set the initial position of a window. `vert` and `horz` are in increments dictated by the mode selected by `SetScalingMode::Application`. The window object passed is returned.

If this function is not called, Windows will automatically set it to a reasonable default.

Example:

```
object  myWind;

myWind = vNew(MainWindow, "App Name");
gSetPosition(myWind, 3, 10);
```

See also: `SetScalingMode::Application`, `SetSize`, `GetPosition`

`SetRaw::Window` [SetRaw]

```
r = gSetRaw(wind, flag);

object  wind;   /*  a window object     */
int     flag;   /*  enable/disable flag */
int     r;      /*  previous flag       */
```

This method is used to set the raw mode associated with keyboard IO. It only has effect on `Read`, `Gets`, and `Getch`. If flag is 1, raw mode is enabled, and 0 disables raw mode. The default is raw mode disabled.

Normally, with raw mode disabled, when a user enters keyboard data, they are able to correct mistakes by using the backspace and reentering the correct data. The resulting data the program receives is the final, corrected input.

When raw mode is enabled, every key the user hits gets returned. This includes backspaces. Therefore, with raw mode enabled, if the user types “ABD” followed by backspace and then “C”, the program will receive all five characters hit. However, with raw mode disabled, the program would only receive the resulting string, “ABC”.

The value returned by this method is the previous raw mode.

Example:

```
object  myWind;

gSetRaw(myWind, 1);
```

See also: `Getch`, `SetBlock`, `Gets`

`SetSize::Window` [SetSize]

```
r = gSetSize(wind, vert, horz);

object  wind;   /*  a window object     */
int     vert;   /*  vertical size       */
int     horz;   /*  horizontal size     */
object  r;      /*  wind arg passed     */
```

This method is used to set the initial size of a window. `vert` and `horz` are in increments dictated by the mode selected by `SetScalingMode::Application`. The window object passed is returned.

If this function is not called, Windows will automatically set it to a reasonable default.

Example:

```
object  myWind;

myWind = vNew(MainWindow, "App Name");
gSetSize(myWind, 10, 40);
```

See also: `SetScalingMode::Application`, `SetPosition`, `GetSize`

`SetStyle::Window` [SetStyle]

```
r = gSetStyle(wind, sty);

object  wind;   /*  a window object             */
DWORD   sty;    /*  the window style to use     */
object  r;      /*  returns wind                */
```

This method is used to set the window style associated with window `wind` to style `sty`. The `DWORD` data type and valid styles are defined by Windows and fully documented by the Windows documentation. See the Windows documentation for the function called `CreateWindow`. The style types normally begin with `WS_` and would be or'ed together to form the selected style. Note that WDS automatically assigns reasonable defaults to a window style when it is created.

Example:

```
object  myWind;

myWind = vNew(MainWindow, "App Name");
gSetStyle(myWind, WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL);
```

SetTag::Window [SetTag]

```
r = gSetTag(wind, tag);

object  wind;   /*  a window object   */
object  tag;    /*  tag               */
object  r;      /*  previous tag      */
```

This method is used to associated an arbitrary Dynace object with a window object. This may later be retrieved via the `GetTag` method. Since WDS passes around the window object to all `Window` methods this mechanism may be used to pass additional information with the window. And since Dynace treats all objects in a uniform manner, this information attached to the window may be arbitrarily complex.

WDS does not dispose of the tag when the window object is disposed. This method returns any previous object associated with the window or `NULL`.

Example:

```
object  myWind;

gSetTag(myWind, gNewWithInt(ShortInteger, 17));
```

See also: `GetTag`, `SetTag::Dialog`

SetTopic::Window [SetTopic]

```
pt = gSetTopic(wind, tpc);

object  wind;   /*  child window object   */
char    *tpc;   /*  help topic            */
char    *pt;    /*  previous help topic   */
```

This method is used to associate help text with window `wind`. The help text is defined using the Windows help system and labeled with the topic indicated by `tpc`. Then, if the user hits the F1 key while in the window, WDS will automatically bring up the Windows help system and find the indicated topic.

WDS also supports dialog and control specific topics. See the appropriate sections.

This method returns any previous topic associated with the window.

Example:

```
object  myWind;

gSetTopic(myWind, "myWindHelp");
```

See also: The `HelpSystem` class.

`Show::Window` [Show]

```
r = gShow(wind);

object  wind;   /*  a window object     */
int     r;      /*  always 0            */
```

Once a window object is created (via `vNew`) and its various attributes set, `gShow` is called in order to actually create the Windows window, with the selected attributes, and display (show) it.

Example:

```
object  myWind;

myWind = vNew(PopupWindow, "Name", 10, 40);
gShow(myWind);
```

See also: `AutoShow`, `ProcessMessages::MainWindow`

`TextBrush::Window` [TextBrush]

```
r = gTextBrush(wind, brsh);

object  wind;   /*  a window object   */
object  brsh;   /*  brush object      */
object  r;      /*  wind              */
```

This method is used to determine what brush object is used for foreground text which is displayed. Any previously associated brush object will be disposed. This brush object will also be automatically disposed when the window is disposed.

The window passed is returned.

Example:

```
object  myWind;

gTextBrush(myWind, vNew(SolidBrush, 255, 0, 0));
```

See also: `BackBrush`, `Use`, `SetBackBrush::Application` and the `Brush` classes

`TextOut::Window` [TextOut]

```
r = gTextOut(wind, row, col, txt);

object  wind;   /*  a window object          */
int     row;    /*  row number of output     */
int     col;    /*  column number of output  */
char    *txt;   /*  string to be output      */
object  r;      /*  wind                     */
```

This method is used to display a string of text (`txt`) at row `row` and column `col`. `row` and `col` have their origin in the upper left hand corner of the window and are scaled as dictated by the `SetScalingMode::Application` method. Their index origin is 0.

This method returns the window argument passed.

Example:

```
object  myWind;

gTextOut(myWind, 10, 30, "Hello World");
```

See also: `Write`, `Puts`, `Printf`

`Update::Window` [Update]

```
r = gUpdate(wind);

object  wind;   /*  a window object     */
object  r;      /*  wind                */
```

This method is used to explicitly update the display with changes the program may have made to window `wind`. WDS normally handles this need, however, this method is available in case the programmer performs special processing and wishes to update the entire window at one time.

This method returns the window argument passed. `Update::MainWindow`.

Example:

```
object  myWind;

gUpdate(myWind);
```

Use::Window [Use]

```
r = gUse(wind, obj);

object  wind;   /*  a window object     */
object  obj;    /*  arbitrary object    */
object  r;      /*  the object passed   */
```

This method is used as a general purpose mechanism to associated a number of object types with a window. `obj` may be a `Font, Icon, Cursor, Menu,` or the background `Brush` object. Those objects would have normally been created via their associated classes and then may be associated with a window via this method.

Note that the same object should not be associated with more than one window. The problem is that if one window is deleted, WDS would delete all the objects associated with that window and then the other window would reference objects which have been deleted. The way around this is to use the `Copy` method in order to make a copy of an object prior to associating with a new window. This way there would be two independent objects such that if one is deleted the other would still exist.

Note also that the `Application` class may be used to set application wide defaults for these types of objects.

The value returned is the object passed.

Example:

```
object  myWind, myOtherWind, someFont;

someFont = vNew(ExternalFont, "Times New Roman", 12);
if (someFont)  {
        gUse(myWind, someFont);
        gUse(myOtherWind, gCopy(someFont));
}
```

See also: `LoadFont, LoadIcon, LoadCursor, LoadMenu, TextBrush, BackBrush`

`VertShift::Window` [VertShift]

```
r = gVertShift(wind, rows);

object  wind;   /*  a window object     */
int     rows;   /*  rows to shift       */
object  r;      /*  the window object   */
```

This method is used to perform a vertical shift of the text in the memory buffer which is being displayed in the window. WDS keeps a buffer which holds a programmer definable number of lines of text. The user is then able to scroll through this text. If the application attempts to display more lines than this maximum (via `vPrintf` for example) the system will automatically call `gVertShift` in order to eliminate the top line and make room for the new line being appended. This method is mainly used internally.

`wind` is the window which is to be affected, and `rows` is the number of rows to scroll. If `rows` is positive the scroll moves the text up, and negative moves the text down. Scrolling up permanently destroys lines at the beginning of the buffer and scrolling down permanently destroys lines at the end of the buffer.

The scaling factor associated with `rows` is set by `SetScalingMode::Application` and the maximum lines associated with a window may be set with `SetMaxLines`.

Example:

```
object  myWind;

gVertShift(myWind, 2);
```

See also: `ScrollVert`, `SetMaxLines`

`Write::Window` [Write]

```
r = gWrite(wind, txt, len);

object   wind;  /*  a window object       */
char    *txt;   /*  string to be output   */
unsigned len;   /*  length of string      */
int      r;     /*  bytes written         */
```

This method is used to display a string of text (`txt`) in a sequential fashion on window `wind`. Since the `Window` class is a subclass of `Stream`, this method is mainly provided to support the standard interface dictated by the `Stream` class. By providing this method, the `Window` class automatically inherits the `Puts` and `Printf` capability.

`txt` is the text to be displayed and `len` is the length of that string. Text `txt` will be displayed on window `wind` and `len` will be returned.

Example:

```
object  myWind;

gWrite(myWind, "Hello World\n", 12);
```

See also: `TextOut`, `Puts`, `Printf`

4.4.1 Main Window

This class, named `MainWindow`, is used to create and manipulate an application's main window. There is normally one main window associated with each application and this is the first window created.

This class is a subclass of the `Window` class and therefore inherits all of the `Window` class's functionality. The methods documented in this subsection are only those which are particular to the `MainWindow` class.

`New::MainWindow` [New]

```
r = vNew(MainWindow, ttl);

char    *ttl;   /*  window title  */
object  r;      /*  new window    */
```

This class method is used to create the main application window. Since this is a class method the first argument must be literally `MainWindow`. The window title (`ttl`) will appear at the top of the window.

The value returned is the new window created.

Example:

```
object  myWind;

myWind = vNew(MainWindow, "My Application");
```

See also: `New::ChildWindow`, `New::PopupWindow`, `Show`, `Dispose::Window`

`ProcessMessages::MainWindow` [ProcessMessages]

```
r = gProcessMessages(wind);

object  wind;   /*  a window object       */
int     r;      /*  final message result  */
```

Once the main application window is created (via `vNew`) and its various attributes set, `gProcessMessages` is called in order to actually create the Windows window, with the selected attributes, display (show) it, and process the application's messages. Processing the application's messages is what allows the user to interact with the application.

The value returned is the return value specified when the application is terminated. This can be specified via `QuitApplication::Application`.

Example:

```
object  myWind;

myWind = vNew(MainWindow, "Application Name");
gProcessMessages(myWind);
```

See also: `New, Show::Window, ProcessMessages::MessageDispatcher`

4.4.2 Child Windows

The Child Window class, called `ChildWindow`, is used in the creation of windows which are children of other windows. As such, they can not be positioned outside of their parent window, and they get iconized along with their parent window.

This class is a subclass of the `Window` class and therefore inherits all of the `Window` class's functionality. The methods documented in this subsection are only those which are particular to the `ChildWindow` class.

`New::ChildWindow` [New]

```
r = vNew(ChildWindow, prnt, rows, cols);

object  prnt;   /*  parent window object  */
int     rows;   /*  length of window      */
int     cols;   /*  width of window       */
object  r;      /*  new window            */
```

This class method is used to create a child window. Since this is a class method the first argument must be literally `ChildWindow`. `prnt` represents the parent window object of which the new window will be a child. `rows` and `cols` determine the initial size of the window in are specified in increments dictated by `SetScalingMode::Application`.

The default style associated with the child window is `WS_CHILD | WS_VISIBLE` and may be changed with `SetStyle::Window`. The position may be set with `SetPosition::Window`.

The value returned is the new window created.

Example:

```
object  myWind, mainWind;

myWind = vNew(ChildWindow, mainWind, 10, 45);
```

See also: `New::MainWindow`, `New::PopupWindow`, `Show`, `Dispose::Window`

4.4.3 Popup Windows

The `PopupWindow` class is used to create arbitrary windows which function independently of other windows. That is, they may overlap or be moved outside of other windows and iconized independently of other windows. If, however, a popup window is associated with a parent window (via `SetParent::Window`) then it may move outside the parent but will be iconized with it.

This class is a subclass of the `Window` class and therefore inherits all of the `Window` class's functionality. The methods documented in this subsection are only those which are particular to the `PopupWindow` class.

`New::PopupWindow` [New]

```
r = vNew(PopupWindow, name, rows, cols);

char    *name;  /*  window title      */
int     rows;   /*  length of window  */
int     cols;   /*  width of window   */
object  r;      /*  new window        */
```

This class method is used to create a popup window. Since this is a class method the first argument must be literally `PopupWindow`. `name` represents the window title and will appear in the window's top border. `rows` and `cols` determine the initial size of the window in are specified in increments dictated by `SetScalingMode::Application`.

The default style associated with the popup window is `WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_SYSMENU`
and may be changed with `SetStyle::Window`. The position may be set with `SetPosition::Window`.

The value returned is the new window created.

Example:

```
object  myWind, mainWind;

myWind = vNew(PopupWindow, "Window Title", 10, 45);
```

See also: `New::MainWindow`, `New::ChildWindow`, `Show`, `Dispose::Window`

4.5 Printing

The `Printer` class is used for all aspects of printing. It is a subclass of the Dynace `Stream` class and as such inherits all of its functionality.

All methods in the class use an argument referred to as `pntr`. This will always be the printer object returned by `QueryPrinter`, `New` or `NewWithHDC` and used to identify the printer which is to be effected.

The value returned by all methods which cause text or graphics output is very significant. During normal operation, the printer object passed will be returned. However, if an error occurs or the user aborted the report a `NULL` will be returned. In this case further output should be avoided (although it won't hurt - it'll just be ignored) and the printer object should be disposed.

All positioning parameters are in increments dictated by `SetScale`. Positions begin in the upper left hand corner and have an index origin of 0.

`Arc::Printer` [Arc]

```
r = gArc(pntr, yBeg, xBeg, yEnd, xEnd, yaBeg, xaBeg, yaEnd, xaEnd);

object  pntr;   /*  printer object        */
int     yBeg;   /*  starting row          */
int     xBeg;   /*  starting column       */
int     yEnd;   /*  ending row            */
int     xEnd;   /*  ending column         */
int     yaBeg;  /*  starting arc row      */
int     xaBeg;  /*  starting arc column   */
int     yaEnd;  /*  ending arc row        */
int     xaEnd;  /*  ending arc column     */
object  r;      /*  printer object        */
```

This method is used to output an elliptical arc to the printer. The currently selected pen object will indicate the thickness and pattern of the outline of the shape, and the currently selected brush object will be used to determine what pattern the shape will be filled with. The parameters indicate location of the beginning and ending of the shape as well as the coordinated of the arc and are in increments dictated by `SetScale`.

See the note at the beginning of this section regarding the return value.

Example:

```
object  pntr;

if (!gArc(pntr, 10, 10, 180, 140, 12, 12, 128, 135))
        abort report;
```

See also: `NewPage`, `SetScale`, `Use`, `Line`, `Rectangle`, `Ellipse`, `RoundRect`, `Chord`, `Pie`

`Chord::Printer` [Chord]

```
r = gChord(pntr, yBeg, xBeg, yEnd, xEnd,
                 ylBeg, xlBeg, ylEnd, xlEnd);

object  pntr;   /*  printer object         */
int     yBeg;   /*  starting row           */
int     xBeg;   /*  starting column        */
int     yEnd;   /*  ending row             */
int     xEnd;   /*  ending column          */
int     ylBeg;  /*  starting line row      */
int     xlBeg;  /*  starting line column   */
int     ylEnd;  /*  ending line row        */
int     xlEnd;  /*  ending line column     */
object  r;      /*  printer object         */
```

This method is used to output a chord shape to the printer. The currently selected pen object will indicate the thickness and pattern of the outline of the shape, and the currently selected brush object will be used to determine what pattern the shape will be filled with. The parameters indicate location of the beginning and ending of the shape as well as the coordinated of the intersecting line and are in increments dictated by `SetScale`.

See the note at the beginning of this section regarding the return value.

Example:

```
object  pntr;

if (!gChord(pntr, 10, 10, 180, 140, 12, 12, 128, 135))
        abort report;
```

See also: `NewPage`, `SetScale`, `Use`, `Line`, `Rectangle`, `Ellipse`, `RoundRect`, `Pie`, `Arc`

DeepDispose::Printer [DeepDispose]

This method performs the same function as `Dispose`. See that method for details.

Dispose::Printer [Dispose]

```
r = gDispose(pntr);

object  pntr;   /*  printer object  */
object  r;      /*  NULL            */
```

This method is used to flush the final output page, close the printer and dispose of the printer object. It must be called when a report is is complete and the printer is no longer needed.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  pntr;

pntr = gDispose(pntr);
```

Ellipse::Printer [Ellipse]

```
r = gEllipse(pntr, yBeg, xBeg, yEnd, xEnd);

object  pntr;   /*  printer object   */
int     yBeg;   /*  starting row     */
int     xBeg;   /*  starting column  */
int     yEnd;   /*  ending row       */
int     xEnd;   /*  ending column    */
object  r;      /*  printer object   */
```

This method is used to output an ellipse to the printer. The currently selected pen object will indicate the thickness and pattern of the outline of the shape, and the currently selected brush object will be used to determine what pattern the shape will be filled with. The parameters indicate location of the beginning and ending of the shape and are in increments dictated by `SetScale`.

See the note at the beginning of this section regarding the return value.

Example:

```
object  pntr;

if (!gEllipse(pntr, 10, 20, 30, 40))
        abort report;
```

See also: `NewPage`, `SetScale`, `Use`, `Line`, `Rectangle`, `RoundRect`, `Chord`, `Pie`, `Arc`

`Handle::Printer` [Handle]

```
r = gHandle(pntr);

object  pntr;   /*  printer object         */
HANDLE  hdc;    /*  handle device context  */
```

This method is used to obtain the device context handle associated with an opened printer represented by `pntr`. It is used internally by Windows and should not normally be needed.

Example:

```
HDC     hdc;

hdc = (HDC) gHandle(pntr);
```

`Line::Printer` [Line]

```
r = gLine(pntr, yBeg, xBeg, yEnd, xEnd);

object  pntr;   /*  printer object   */
int     yBeg;   /*  starting row     */
int     xBeg;   /*  starting column  */
int     yEnd;   /*  ending row       */
int     xEnd;   /*  ending column    */
object  r;      /*  printer object   */
```

This method is used to output a line to the printer. The currently selected pen object will indicate the thickness and pattern of the line. The parameters indicate location of the beginning and ending of the line and are in increments dictated by `SetScale`.

See the note at the beginning of this section regarding the return value.

Example:

```
object  pntr;

if (!gLine(pntr, 10, 20, 30, 40))
        abort report;
```

See also: `NewPage`, `SetScale`, `Use`, `Arc`, `Rectangle`, `Ellipse`, `RoundRect`, `Chord`, `Pie`

`LoadFont::Printer` [LoadFont]

```
r = gLoadFont(pntr, fname, sz);

object   pntr;  /*  printer object   */
char    *fname; /*  font name        */
int      sz;    /*  point size       */
object   r;     /*  font object      */
```

This method is used to load an arbitrary font by name at any point size and associate it as the default font for future text output to printer `pntr`. `fname` is the full name of the font as it appears when you list the available fonts via the control-panel / fonts Windows utility, minus the font type in parentheses. `sz` indicates the desired point size.

Any font object previously associated with the printer will be disposed. When the printer object is disposed, the font object will also be disposed.

The value returned is an object representing the font loaded, or `NULL` if the font was not found.

Note that `fname` may also be a Dynace object cast as a (`char *`).

Example:

```
object  pntr;

gLoadFont(pntr, "Times New Roman", 12);
```

See also: `LoadSystemFont`, `Indirect::ExternalFont`, `Use`

`LoadSystemFont::Printer` [LoadSystemFont]

```
r = gLoadSystemFont(pntr, fnt);

object   pntr;  /*  a printer object  */
unsigned fnt;   /*  font identifier   */
object   r;     /*  font object       */
```

This method is used to load a Windows predefined font and associate it with printer `pntr`. The last font associated with a printer is the one which will be used when any text is output to the printer.

`fnt` is a Windows defined macro which identifies the font. The available options are defined in the Windows documentation under the function named `GetStockObject` and normally end with `_FONT`. The font object will be automatically destroyed whenever the printer object is disposed.

The value returned is an object representing the font, or `NULL` if the font was not found.

Example:

```
object  pntr;

gLoadSystemFont(pntr, SYSTEM_FONT);
```

See also: `LoadFont`, `Load::SystemFont`, `Use`

`New::Printer` [New]

```
pntr = vNew(Printer, pwind, rname);

object  pwind;  /*  parent window       */
char    *rname; /*  report name         */
object  pntr;   /*  new printer object  */
```

This class method is used to create a new `Printer` object which will be used by the `Printer` instance methods to control the printer, fonts and content of output. This method opens up the default printer.

`pwind` is the window which will act as the parent to any messages which may be displayed when a report is being printed. `rname` determines what the report is referred to as in any messages.

The value returned represents the printer to be used. If the default printer couldn't be opened, `NULL` will be returned. If an object is returned, it must be disposed (via `Dispose`) when it is no longer needed.

Example:

```
object  pntr, pwind;

pntr = vNew(Printer, pwind, "My Report");
```

See also: `QueryPrinter`, `NewWithHDC`

`NewPage::Printer` [NewPage]

```
r = gNewPage(pntr);

object  pntr;   /*  printer object  */
object  r;      /*  printer object  */
```

This method is used to flush and eject the current output page (if anything had been output to it) and prepare for a possible new page of output.

See the note at the beginning of this section regarding the return value.

Example:

```
object  pntr;

if (!gNewPage(pntr))
        abort report;
```

`NewWithHDC::Printer` [NewWithHDC]

```
pntr = gNewWithHDC(Printer, pwind, rname, hdc);

object  pwind;  /*  parent window       */
char    *rname; /*  report name         */
HDC     hdc;    /*  device context      */
object  pntr;   /*  new printer object  */
```

This class method is used to create a new `Printer` object which will be used by the `Printer` instance methods to control the printer, fonts and content of output.

This method opens up the printer identified by `hdc`. This is a Windows internal identifier and may be obtained via the `PrintDialog` class.

`pwind` is the window which will act as the parent to any messages which may be displayed when a report is being printed. `rname` determines what the report is referred to as in any messages.

The value returned represents the printer to be used. If the printer couldn't be opened, `NULL` will be returned.

This method is seldom needed due to `QueryPrinter` and `New`. Use them. If an object is returned, it must be disposed (via `Dispose`) when it is no longer needed.

Example:

```
object  pntr, pwind;
HDC     hdc;

pntr = gNewWithHDC(Printer, pwind, "My Report", hdc);
```

See also: `QueryPrinter`, `New`

`Pie::Printer` [Pie]

```
r = gPie(pntr, yBeg, xBeg, yEnd, xEnd, yaBeg, xaBeg, yaEnd, xaEnd);

object  pntr;   /*  printer object         */
int     yBeg;   /*  starting row           */
int     xBeg;   /*  starting column        */
int     yEnd;   /*  ending row             */
int     xEnd;   /*  ending column          */
int     yaBeg;  /*  starting arc row       */
int     xaBeg;  /*  starting arc column    */
int     yaEnd;  /*  ending arc row         */
int     xaEnd;  /*  ending arc column      */
object  r;      /*  printer object         */
```

This method is used to output a pie shaped wedge to the printer. The currently selected pen object will indicate the thickness and pattern of the outline of the shape, and the currently selected brush object will be used to determine what pattern the shape will be filled with. The parameters indicate location of the beginning and ending of the shape as well as the coordinated of the intersecting line and are in increments dictated by `SetScale`.

See the note at the beginning of this section regarding the return value.

Example:

```
object  pntr;

if (!gPie(pntr, 10, 10, 180, 140, 12, 12, 128, 135))
        abort report;
```

See also: `NewPage`, `SetScale`, `Use`, `Line`, `Rectangle`, `Ellipse`, `RoundRect`, `Chord`, `Arc`

Printf::Printer [Printf]

```
r = vPrintf(pntr, fmt, ...);

object   pntr;  /*  printer object     */
char    *fmt;   /*  format string      */
int      r;     /*  length of output   */
```

This method is used to output a string of text (`str`) in a sequential fashion on printer `pntr`. It is analogous to the standard C library function `fprintf`. All of the standard features of your C library `fprintf` function are supported and that documentation should be consulted for full documentation on the arguments.

This method is used to support the standard streams interface, is actually defined by the `Stream` class, and is documented here for convenience.

Note that this method begins with "v" since it takes variable arguments.

The value returned by this method is very significant. During normal operation, the length of the output string will be returned. However, if an error occurs or the user aborted the report a `-1` will be returned. In this case further output should be avoided (although it won't hurt - it'll just be ignored) and the printer object should be disposed.

Example:

```
object  pntr;
int     age = 32;

if (-1 == vPrintf(pntr, "My age =- %d\n", age))
        goto abort report;
```

See also: `TextOut`, `Puts`

Puts::Printer [Puts]

```
r = gPuts(pntr, str);

object   pntr;  /*  printer object        */
char    *str;   /*  string to be output   */
int      r;     /*  length of str         */
```

This method is used to output a string of text (`str`) in a sequential fashion on printer `pntr`. This method is used to support the standard streams interface, is actually defined by the `Stream` class, and is documented here for convenience. `str` is the text to be output.

The value returned by this method is very significant. During normal operation, the length of the output string will be returned. However, if an error occurs or the user aborted the report a `-1` will be returned. In this case further output should be avoided (although it won't hurt - it'll just be ignored) and the printer object should be disposed.

Example:

```
object  pntr;

if (-1 == gPuts(pntr, "Hello World\n"))
        goto abort report;
```

See also: `TextOut`, `Printf`

`QueryPrinter::Printer` [QueryPrinter]

```
pntr = gQueryPrinter(Printer, pwind, rname);

object  pwind;  /*  parent window       */
char    *rname; /*  report name         */
object  pntr;   /*  new printer object  */
```

This class method is used to create a new `Printer` object which will be used by the `Printer` instance methods to control the printer, fonts and content of output.

This method queries the user for printer selection and optional configuration, and uses that information to open the selected printer.

`pwind` is the window which will act as the parent to any messages which may be displayed when a report is being printed. `rname` determines what the report is referred to as in any messages.

The value returned represents the printer to be used. If the user canceled the printer selection or the printer couldn't be opened, `NULL` will be returned. If an object is returned, it must be disposed (via `Dispose`) when it is no longer needed.

Example:

```
object  pntr, pwind;

pntr = gQueryPrinter(Printer, pwind, "My Report");
```

See also: `New`, `NewWithHDC`

`Rectangle::Printer` [Rectangle]

```
r = gRectangle(pntr, yBeg, xBeg, yEnd, xEnd);

object  pntr;   /*  printer object   */
int     yBeg;   /*  starting row     */
int     xBeg;   /*  starting column  */
int     yEnd;   /*  ending row       */
int     xEnd;   /*  ending column    */
object  r;      /*  printer object   */
```

This method is used to output a rectangle to the printer. The currently selected pen object will indicate the thickness and pattern of the outline of the shape, and the currently selected brush object will be used to determine what pattern the shape will be filled with. The parameters indicate location of the beginning and ending of the shape and are in increments dictated by `SetScale`.

See the note at the beginning of this section regarding the return value.

Example:

```
object  pntr;

if (!gRectangle(pntr, 10, 20, 30, 40))
        abort report;
```

See also: `NewPage`, `SetScale`, `Use`, `Line`, `Ellipse`, `RoundRect`, `Chord`, `Pie`, `Arc`

`RoundRect::Printer` [RoundRect]

```
r = gRoundRect(pntr, yBeg, xBeg, yEnd, xEnd, eHt, eWth);

object  pntr;   /*  printer object         */
int     yBeg;   /*  starting row           */
int     xBeg;   /*  starting column        */
int     yEnd;   /*  ending row             */
int     xEnd;   /*  ending column          */
int     eHt;    /*  corner ellipse height  */
int     eWth;   /*  corner ellipse width   */
object  r;      /*  printer object         */
```

This method is used to output a rectangle with rounded corners to the printer. The currently selected pen object will indicate the thickness and pattern of the outline of the shape, and the currently selected brush object will be used to determine what pattern the shape will be filled with. The parameters indicate location of the beginning and ending of the shape and are in increments dictated by `SetScale`.

See the note at the beginning of this section regarding the return value.

Example:

```
object  pntr;

if (!gRoundRect(pntr, 10, 20, 30, 40, 1, 1))
        abort report;
```

See also: `NewPage`, `SetScale`, `Use`, `Line`, `Rectangle`, `Ellipse`, `Chord`, `Pie`, `Arc`

`SetScale::Printer` [SetScale]

```
r = gSetScale(pntr, ht, wth);

object  pntr;   /*  printer object  */
int     ht;     /*  logical height  */
int     wth;    /*  logical width   */
object  r;      /*  printer object  */
```

This method is used to set the logical scaling factor associated with an output device. Each output page is divided up into `ht` evenly spaced row and `wth` evenly spaced columns. This scaling factor will be used by all text and graphics output when determining output size and location on the page.

Coordinates begin at the upper left hand of the page and are zero origin. The defaults are set to 80 columns and 66 lines.

Example:

```
object  pntr;

gSetScale(pntr, 132, 160);
```

See also: `TextOut`

`TextOut::Printer` [TextOut]

```
r = gTextOut(pntr, row, col, txt);

object  pntr;   /*  printer object  */
int     row;    /*  output row      */
int     col;    /*  output column   */
char    *txt;   /*  text to output  */
object  r;      /*  printer object  */
```

This method is used to output a line of text (`txt`) to the printer. `row` and `col` indicate the position of the text and are in increments dictated by `SetScale`. Positions begin in the upper left hand corner and have an index origin of 0.

The text will be printed in the currently selected font. Note that the actual printing will not occur until the report is complete and the print object disposed.

See the note at the beginning of this section regarding the return value.

Example:

```
object  pntr;

if (!gTextOut(pntr, 10, 20, "output text"))
        abort report;
```

See also: `NewPage, Printf, SetScale, LoadFont`

`Use::Printer` [Use]

```
r = gUse(pntr, obj);

object  pntr;   /*  printer object     */
object  obj;    /*  arbitrary object   */
object  r;      /*  the object passed  */
```

This method is used as a general purpose mechanism to associated a number of object types with a printer. `obj` may be a `Font, Brush` or `Pen` object. Those objects would have normally been created via their associated classes and then may be associated with a printer via this method.

Note that the same object should not be associated with more than one printer object. The problem is that if one printer object is disposed, WDS would dispose of all the objects associated with that printer and then the other printer would reference objects which have been deleted. The way around this is to use the `Copy` method in order to make a copy of an object prior to associating it with a new printer. This way there would be two independent objects such that if one is deleted the other would still exist.

The value returned is the object passed.

Example:

```
object  myPntr, myOtherPntr, someFont;

someFont = vNew(ExternalFont, "Times New Roman", 12);
if (someFont)  {
        gUse(myPntr, someFont);
        gUse(myOtherPntr, gCopy(someFont));
}
```

See also: `LoadFont` and the `Brush` and `Pen` classes.

4.6 Menus

The `Menu` class, although not used directly by an application, is used to house the common functionality associated with the `ExternalMenu` and `InternalMenu` classes. Most of the functionality of those classes is implemented and documented in this section.

External menus (those defined in resources) would be created and used via the `ExternalMenu` class. Internal menus (those defined at runtime by the application code) would be created and used via the `InternalMenu` and `PopupMenu` classes. Since both of these classes are subclasses of the `Menu` class, they inherit most of their functionality from it.

`Associate::Menu` [Associate]

```
r = mAssociate(menu, id, fun);

object  menu;   /*  a menu object     */
int     id;     /*  menu item id      */
long    (*fun)(object);  /*  function to execute  */
object  r;      /*  the menu object   */
```

This method is used to associate a function with a menu item such that if the user selects the menu item, `fun` gets executed.

In the case of external menus, `id` is normally a macro created by the resource editor while the programmer defines the menu and identifies a particular menu option.

`fun` is the application specific function which automatically gets executed whenever the user selects the indicated option. `fun` has the following structure:

```
static  long    menu_option(object wind)
{
            .
            .
            .
        return 0L;
}
```

Where `wind` is the window object which the menu is attached to. The value returned is what is returned by the window procedure associated with the window. This value is normally 0 and is fully documented by the Windows documentation under the `WM_COMMAND` message.

The value returned is the menu object passed.

Example:

```
object  myMenu;

mAssociate(myMenu, ID_FILE_OPEN, menu_option);
```

See also: `AddMenuOption::InternalMenu`, `AddMenuOption::PopupMenu`

`DeepDispose::Menu` [DeepDispose]

```
r = gDeepDispose(menu);

object  menu;   /*  a menu object    */
object  r;      /*  NULL             */
```

This method is used to dispose of a menu object and all menu objects which were pushed (via `Push`) into the same menu stack. It is not often needed because menus associated with a window are automatically disposed of (via this method) when the window is disposed.

If `menu` is an internal menu, all associated popup menus will also be disposed.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  myMenu;

myMenu = gDeepDispose(myMenu);
```

See also: `Dispose`, `Push`

Dispose::Menu [Dispose]

```
r = gDispose(menu);

object  menu;   /*  a menu object   */
object  r;      /*  NULL            */
```

This method is used to dispose of a menu object when it is no longer needed. It is not often needed because menus associated with a window are automatically disposed of (via DeepDispose) when the window is disposed.

If menu is an internal menu, all associated popup menus will also be disposed.

The value returned is always NULL and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  myMenu;

myMenu = gDispose(myMenu);
```

See also: DeepDispose

Handle::Menu [Handle]

```
h = gHandle(menu);

object  menu;   /*  menu object    */
HANDLE  h;      /*  Windows handle */
```

This method is used to obtain the Windows internal handle associated with a menu object.

Note that this method may be used with most WDS objects in order to obtain the internal handle that Windows normally associates with each type of object. See the appropriate documentation.

Example:

```
object  myMenu;
HANDLE  h;

h = gHandle(myMenu);
```

`MenuFunction::Menu` [MenuFunction]

```
fun = mMenuFunction(menu, id);

object  menu;   /*  a menu object     */
int     id;     /*  menu item id      */
long    (*fun)(object);  /*  function to execute  */
```

This method is used to obtain the function which was associated with a menu item.

See also: `Associate`

`Pop::Menu` [Pop]

```
prev = gPop(top);

object  top;    /*  top menu       */
object  prev;   /*  previous menu  */
```

This method is used to dispose of menu `top` and gain access to the next menu in menu `top`'s last in, first out list. It is mainly used by the `Window` class in order to support the ability to change a menu and later return to a previous menu. The value returned is the next menu in the stack.

Example:

```
object  myMenu, subMenu;
HANDLE  h;

subMenu = gPop(myMenu);
```

See also: `Push`, `Use::Window`, `PopMenu::Window`

`Push::Menu` [Push]

```
r = gPush(top, psh);

object  top;    /*  top menu     */
object  psh;    /*  pushed menu  */
object  r;      /*  top menu     */
```

This method is used to push menu (`psh`) into a last in, first out stack accessible through menu `top`. It is mainly used by the `Window` class in order to support the ability to change a menu and later return to a previous menu. The value returned is the top menu passed.

Example:

```
object  myMenu, subMenu;
HANDLE  h;

h = gPush(myMenu, subMenu);
```

See also: `Pop`, `Use::Window`, `PopMenu::Window`

`SetMode::Menu` [SetMode]

```
r = mSetMode(menu, id, mode);

object  menu;   /*  menu       */
unsigned  id;   /*  menu item  */
unsigned  mode; /*  item mode  */
object    r;    /*  menu       */
```

This method is used to set the mode associated with a menu item. This mode specifies whether the menu is enabled, disabled or grayed.

In the case of external menus, `id` is normally a macro created by the resource editor while the programmer defines the menu and identifies a particular menu option. In the case of internal menu's, `id` is the number returned by `AddMenuOption::InternalMenu` or `AddMenuOption::PopupMenu`.

`mode` is a Windows defined macro and may be one of the following:

```
MF_ENABLED      to enable the menu item
MF_DISABLED     to disable the menu item
MF_GRAYED       to gray the menu option
```

The value returned is the menu passed.

Example:

```
object  myMenu;

mSetMode(myMenu, ID_FILE_OPEN, MF_DISABLED);
```

See also: `AddMenuOption::InternalMenu`, `AddMenuOption::PopupMenu`

4.6.1 External Menus

The `ExternalMenu` class is used to access menus which were defined using the resource editor. Since this class is a subclass of `Menu`, all of `Menu`'s functionality is accessible through instance of this class. Therefore, see the `Menu` class for additional functionality.

Load::ExternalMenu [Load]

```
menu = mLoad(ExternalMenu, id);

unsigned  id;   /*  menu id  */
object  menu;   /*  menu     */
```

This method is used to create a new menu object representing a menu defined with the resource editor.

`id` is normally a macro created by the resource editor which identifies a particular menu.

The value returned is a new object representing the menu or `NULL` if the menu identified is not found.

Example:

```
object  myMenu;

myMenu = mLoad(ExternalMenu, MY_MENU);
```

See also: `LoadStr`, `LoadMenu::Window`, `LoadMenuStr::Window`

LoadStr::ExternalMenu [LoadStr]

```
menu = gLoadStr(ExternalMenu, id);

char    *id;    /*  menu id  */
object  menu;   /*  menu     */
```

This method is used to create a new menu object representing a menu defined with the resource editor.

`id` is a string which names the desired menu. This name is defined by the resource editor and identifies a particular menu.

The value returned is a new object representing the menu or `NULL` if the menu identified is not found.

Example:

```
object  myMenu;

myMenu = gLoadStr(ExternalMenu, "mymenu");
```

See also: `Load`, `LoadMenu::Window`, `LoadMenuStr::Window`

4.6.2 Internal Menus

The `InternalMenu` class is used to enable an application to create menus on the fly – without the need to pre-specify the menu's structure with the resource editor. Since this class is a subclass of `Menu`, all of `Menu`'s functionality is accessible through instance of this class. Therefore, see the `Menu` class for additional functionality.

`AddMenuOption::InternalMenu` [AddMenuOption]

```
id = gAddMenuOption(mnu, ttl, fun)

object  menu;   /*  menu     */
char    *ttl;   /*  title    */
long    (*fun)(); /*  function  */
int     id;     /*  item id  */
```

This method is used to append a new option to internal menu object `mnu` and associate it to an application specific function (`fun`). Once this is done, if the user selects the new menu option the function `fun` will be executed.

`ttl` represents the character string which will be displayed in the menu. `ttl` may also contain an embedded ampersand (&) character. This causes the character following the ampersand to be underlined. It also causes the following character to be the Alt key selection the user can use to quickly select a menu option.

`fun` is the function which will be executed when the user selects the menu option and has the following form:

```
long    fun(object wind)
{
        .
        .
        .
        return 0L;
}
```

The function executed (`fun`) is passed the window object and returns a long. The return value is documented in the Windows documentation under the message named `WM_COMMAND`. It should normally be `0L`.

This method returns a unique WDS generated id which identifies this particular menu option and may be used to control the status of the option.

Example:

```
static  long    file_message(object wind)
{
        gMessage(wind, "File_Message");
        return 0L;
}

        .
        .
        mnu = vNew(InternalMenu);
        gAddMenuOption(mnu, "&File", file_message);
```

See also: `AddPopupMenu`

`AddPopupMenu::InternalMenu` [AddPopupMenu]

```
id = gAddPopupMenu(mnu, ttl, pm)

object  mnu;    /*  menu        */
char    *ttl;   /*  title       */
object  pm;     /*  Popup menu  */
object  r;      /*  menu        */
```

This method is used to append a new option to menu object `mnu` and associate it to a popup menu. Once this is done, if the user selects the new menu option the popup menu will be displayed.

`ttl` represents the character string which will be displayed in the menu. `ttl` may also contain an embedded ampersand (&) character. This causes the character following the ampersand to be underlined. It also causes the following character to be the Alt key selection the user can use to quickly select a menu option.

`pm` is a popup menu object created with the `PopupMenu` class.

This method returns `mnu`.

Example:

```
object  mnu, pm;

mnu = vNew(InternalMenu);
pm  = vNew(PopupMenu, mnu);
gAddPopupMenu(mnu, "&File", pm);
```

See also: `AddMenuOption`, `New::PopupMenu`

`New::InternalMenu` [New]

```
menu = vNew(InternalMenu);

object  menu;   /*  menu     */
```

This method is used to create a new internal menu. An internal menu is one which is not pre-specified with the resource editor. It may therefore be structured at runtime.

The object returned is a new internal menu object with no items associated with it.

Example:

```
object  myMenu;

myMenu = vNew(InternalMenu);
```

See also: `AddMenuOption`, `AddMenu`, `Use::Window`, `LoadMenu::Window`

4.7 Popup Menus

The `PopupMenu` class is used in conjunction with the `InternalMenu` class in order to provide a nested menu structure. Nesting of popup menus is supported to arbitrary levels.

`AddMenuOption::PopupMenu` [AddMenuOption]

```
id = gAddMenuOption(mnu, ttl, fun)

object  menu;   /*  menu     */
char    *ttl;   /*  title    */
long    (*fun)(); /*  function  */
int     id;     /*  item id  */
```

This method works exactly like `AddMenuOption::InternalMenu`. See that description for full documentation.

See also: `AddPopupMenu`, `AddSeparator`

`AddPopupMenu::PopupMenu` [AddPopupMenu]

```
id = gAddPopupMenu(mnu, ttl, pm)

object  mnu;    /*  menu        */
char    *ttl;   /*  title       */
object  pm;     /*  Popup menu  */
object  r;      /*  menu        */
```

`mnu` is the popup menu which is to act as the parent of `pm` and may be arbitrarily nested.

This method is the same as `AddPopupMenu::InternalMenu`. See that method for complete documentation.

See also: `AddMenuOption, AddSeparator`

`AddSeparator::PopupMenu` [AddSeparator]

```
r = gAddSeparator(mnu)

object  mnu;    /*  menu        */
object  r;      /*  menu        */
```

This method is used to append a horizontal bar to the end of a popup menu. Any menu items added subsequently will appear after the horizontal separator.

The menu object passed is returned.

Example:

```
gAddSeparator(pm);
```

See also: `AddMenuOption`

`Dispose::PopupMenu` [Dispose]

```
r = gDispose(menu);

object  menu;   /*  a menu object   */
object  r;      /*  NULL            */
```

This method is used to dispose of a menu object when it is no longer needed. It is not often needed because popup menus associated with other menus are automatically disposed of (via `DeepDispose`) when the top level menu is disposed.

This class also provides a `DeepDispose` which performs the same function.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  myMenu;

myMenu = gDispose(myMenu);
```

See also: `DeepDispose::Menu`

`Handle::PopupMenu` [Handle]

```
h = gHandle(menu);

object  menu;   /*  menu object    */
HANDLE  h;      /*  Windows handle */
```

This method is used to obtain the Windows internal handle associated with a popup menu object.

Note that this method may be used with most WDS objects in order to obtain the internal handle that Windows normally associates with each type of object. See the appropriate documentation.

Example:

```
object  myMenu;
HANDLE  h;

h = gHandle(myMenu);
```

`MenuFunction::PopupMenu` [MenuFunction]

```
fun = mMenuFunction(menu, id);

object  menu;   /*  a menu object     */
int     id;     /*  menu item id      */
long    (*fun)(object);  /*  function to execute  */
```

This method is used to obtain the function which was associated with a menu item.

See also: `AddMenuOption`

`New::PopupMenu` [New]

```
pm = vNew(PopupMenu, tm);

object  tm;     /*  top menu    */
object  pm;     /*  popup menu  */
```

This method is used to create a new popup menu. A popup menu is one which pops up or appears when a user selects an associated menu option. Popup menus are used in conjunction with internal menus in order to create arbitrarily complex menu structures.

`tm` is the top menu object. It must be an `InternalMenu`, regardless of where the new popup menu will be nested.

The object returned represents the new popup menu created and may be used to add items to that menu.

Example:

```
object  myMenu, pm;

myMenu = vNew(InternalMenu);
pm = vNew(PopupMenu, myMenu);
gAddPopupMenu(myMenu, "&File", pm);
```

See also: `AddMenuOption`, `AddPopupMenu`, `AddSeparator`, `AddPopupMenu::InternalMenu`

4.8 Dialogs

The `Dialog` class implements all the functionality which is common to the `ModalDialog` and `ModelessDialog` classes, and although this class is not used directly, most of their functionality is documented in this section.

4.8.1 Standard Dialog Method Arguments

Since all dialog instance methods have the dialog object as the first argument (referred to as `dlg`), this argument will not be described each time. It always refers to the dialog object which you wish to perform the desired operation on.

Many of the methods associated with this class take an argument identified as `id`. This is a macro defined by the programmer via the resource editor when the dialog is being defined and is used to uniquely identify a particular control within the dialog. Due to the fact that this argument has the same meaning for every method which uses it, it will not be defined each time.

The variable `ctl` will always refer to a control object. Each control object represents a unique control within a dialog. Each control object will be an instance of one of the subclasses of the `Control` class (such as `TextControl` or `NumericControl`).

4.8.2 Dialog Methods

`AddControl::Dialog` [AddControl]

```
ctl = mAddControl(dlg, ctlClass, id);

object  dlg;       /*  a dialog object   */
object  ctlClass;  /*  class of control  */
unsigned  id;      /*  control id        */
object  ctl;       /*  control object    */
```

This method is used to create a new control object and associate it to a control defined in the dialog. `ctlClass` must be literally one of the subclasses of the `Control` class (such as `TextControl`, `CheckBox`, `RadioButton`, etc). See the class hierarchy for a complete list.

`id` identifies which control within dialog `dlg` to associate the new control object with.

The value returned (`ctl`) is the new control object created and will be an instance of the class indicated by the `ctlClass` argument. This object can be configured and queried via the protocol defined by its class. See that section for available options. Setting various options associated with this object will control default values associated with the control and how this control functions within the dialog.

Example:

```
object  dlg, ctl;

ctl = mAddControl(dlg, TextControl, PERSONS_NAME);
```

See also: `GetControl`

`AddDlgHandlerAfter::Dialog` [AddDlgHandlerAfter]

```
r = gAddDlgHandlerAfter(dlg, msg, func);

object   dlg;      /*  a dialog object  */
unsigned msg;      /*  message          */
BOOL    (*func)(); /*  function pointer */
object  r;         /*  the dialog obj   */
```

This method is used to associate function `func` with Windows dialog message `msg` for dialog `dlg`. Whenever dialog `dlg` receives message `msg`, `func` will be called.

`dlg` is the dialog object who's messages you wish to process. `msg` is the particular message you wish to trap. These messages are fully documented in the Windows documentation in the Messages section. They normally begin with `WM_`.

`func` is the function which gets called whenever the specified message gets received and takes the following form:

```
BOOL    func(object     dlg,
             HWND       hdlg,
             UINT       mMsg,
             WPARAM     wParam,
             LPARAM     lParam)
{
        .
        .
        .
        return FALSE;  /* or whatever is appropriate  */
}
```

Where `dlg` is the dialog being sent the message. The remaining arguments and return value is fully documented in the Windows documentation under the `DialogProc` function and the Windows Messages documentation.

WDS keeps a list of functions associated with each message associated with each dialog. When a particular message is received the appropriate list of handler functions gets executed sequentially. `AddDlgHandlerAfter` appends the new function to the end of this list, and `AddDlgHandlerBefore` adds the new function to the beginning of the list.

Windows will only see the return value of the last message handler executed.

Example:

```
int     hSize, vSize;

static  BOOL    process_wm_size(object  dlg,
                                HWND    hwnd,
                                UINT    mMsg,
                                WPARAM  wParam,
                                LPARAM  lParam)
{
        hSize = LOWORD(lParam);
        vSize = HIWORD(lParam);
        return 0L;
}

        .
        .
        gAddDlgHandlerAfter(dlg, (unsigned) WM_SIZE,
                                        process_wm_size);
        .
        .
```

See also: `AddDlgHandlerBefore`

AddDlgHandlerBefore::Dialog [AddDlgHandlerBefore]

```
r = gAddDlgHandlerBefore(dlg, msg, func);

object   dlg;     /*  a dialog object  */
unsigned msg;      /*  message          */
BOOL    (*func)(); /*  function pointer */
object  r;         /*  the dialog obj   */
```

This function is fully documented under `AddDlgHandlerAfter`.

See also: `AddDlgHandlerAfter`

AutoDispose::Dialog [AutoDispose]

```
pflg = gAutoDispose(dlg, flg);

object  dlg;    /*  a dialog object    */
int     flg;    /*  desired mode       */
int     pflg;   /*  previous mode set  */
```

This method is used to control the auto disposal facility associated with a given dialog. The default is disabled for modal dialogs and enabled for modeless dialogs.

If enabled, the auto dispose feature will cause the dialog object (`dlg`) to be automatically disposed of whenever the user accepts or aborts the dialog. If disabled, the application may continue to use the object (for querying control values for example) after the dialog has been terminated. If the auto dispose feature is disabled the application must explicitly dispose of the object when it is no longer needed.

Note that if the auto dispose feature is used, the `gCompletionFunction` method may be used to access the control values immediately prior to the object's automatic disposal.

Set `flg` to 1 to enable the feature and 0 to disable it. The previous state will be returned.

Example:

```
object  dlg;

gAutoDispose(dlg, 1);
```

See also: `CompletionFunction, Perform, SetResult`

BackBrush::Dialog [BackBrush]

```
r = gBackBrush(dlg, brsh);

object  dlg;    /*  a dialog object   */
object  brsh;   /*  brush object      */
object  r;      /*  dlg               */
```

This method is used to set the brush object which is used for the background of the dialog. Any previously associated brush object will be disposed. This brush object will also be automatically disposed when the dialog is disposed.

The dialog passed is returned.

Example:

```
object  dlg;

gBackBrush(dlg, vNew(SolidBrush, 0, 255, 0));
```

See also: `TextBrush`, `SetBackBrush::Application` and the `Brush` classes

CompletionFunction::Dialog [CompletionFunction]

```
r = gCompletionFunction(dlg, fun);

object  dlg;       /*  dialog object        */
int     (*fun)();  /*  completion function  */
object  r;         /*  dialog object        */
```

This method is used to associated a C function with a dialog such that when the user accepts or aborts the dialog, function `fun` will get executed by WDS. The function will get executed prior to any automatic disposal of the dialog object.

The completion function facility is best used in conjunction with modeless dialogs and the auto dispose facility in order to process control values when the user accepts the dialog and prior to the automatic disposal of the dialog object.

The format of `fun` is as follows:

```
r = fun(dlg, res);

object  dlg;    /*  the dialog object  */
int     res;    /*  result status of the dialog  */
int     r;      /*  result for gPerform  */
```

Where `res` will be `TRUE` if the user accepted the dialog or `FALSE` if the user canceled the dialog. `r` is the value which will become the result of the dialog which is returned by `gPerform` (if it was a modal dialog). `r` will normally be `res`.

Example:

```
static  int     fun(object dlg, int res)
{
        .
        .
        return res;
}

        gCompletionFunction(dlg, fun);
```

See also: `AutoDispose`, `Perform`, `SetTag`, `SetResult`

`CtlDoubleValue::Dialog` [CtlDoubleValue]

```
val = mCtlDoubleValue(dlg, id);

object  dlg;    /*  a dialog object  */
unsigned id;    /*  control id       */
double  val;    /*  control value    */
```

This method is used to gain access to the value associated with a particular control as a C double. For example, if the control was a numeric control and the user entered "3.141", then this method would return a double "3.141".

Example:

```
object  dlg;
double  val;

val = mCtlDoubleValue(dlg, PI_VALUE);
```

See also: `CtlStringValue`, `CtlShortValue`, ... `CtlValue`

`CtlLongValue::Dialog` [CtlLongValue]

```
val = mCtlLongValue(dlg, id);

object  dlg;    /*  a dialog object  */
unsigned id;    /*  control id       */
long    val;    /*  control value    */
```

This method is used to gain access to the value associated with a particular control as a C long. For example, if the control was a numeric control and the user entered "6", then this method would return a long "6".

In the case of date controls, the value returned will be in the form YYYYMMDD. For example November 24, 1956 would be 19561124.

Example:

```
object  dlg;
long    val;

val = mCtlLongValue(dlg, PERSONS_AGE);
```

See also: `CtlStringValue, CtlShortValue, ... CtlValue`

`CtlShortValue::Dialog` [CtlShortValue]

```
val = mCtlShortValue(dlg, id);

object  dlg;    /*  a dialog object  */
unsigned id;    /*  control id       */
short   val;    /*  control value    */
```

This method is used to gain access to the value associated with a particular control as a C short. For example, if the control was a numeric control and the user entered "6", then this method would return a short "6".

In the case of a list box or combo box the value returned will be in terms of an ordinal value. This value returned is a zero based index from top to bottom indicating the selection made by the user. If no selection was made negative value will be returned.

In the case of check boxes and radio buttons this method will return 0 if the button is not selected, 1 if the button is selected, and 2 if the button is grayed.

In the case of scroll bars, this method will return a short value between the minimum and maximum set for the control which indicates where the control was set to.

`mCtlLongValue` should be used for date controls.

Example:

```
object  dlg;
short   val;

val = mCtlShortValue(dlg, PERSONS_AGE);
```

See also: `CtlStringValue, CtlLongValue, ... CtlValue`

CtlStringValue::Dialog [CtlStringValue]

```
val = mCtlStringValue(dlg, id);

object  dlg;    /*  a dialog object  */
unsigned id;    /*  control id       */
char    *val;   /*  control value    */
```

This method is used to gain access to the value associated with a particular control as a C character string. For example, if the control was a text control and the user entered "Miami", then this method would return a pointer to the C string "Miami".

If the control being accessed is a list box or combo box, the value returned will be a string representation of the text associated with the choice made by the user. See `CtlShortValue` or `IndexValue` to get an ordinal value for this control type.

The character pointer returned will not be valid once the dialog or control has been disposed. Therefore, the value should be kept by some other means if it is desired past the life of the dialog.

Example:

```
object  dlg;
char    *val;

val = mCtlStringValue(dlg, PERSONS_NAME);
```

See also: `CtlShortValue, CtlLongValue, ... CtlValue`

CtlUnsignedShortValue::Dialog [CtlUnsignedShortValue]

```
val = mCtlUnsignedShortValue(dlg, id);

object  dlg;    /*  a dialog object  */
unsigned id;    /*  control id       */
unsigned short  val;    /*  control value    */
```

This method is used to gain access to the value associated with a particular control as a C unsigned short. For example, if the control was a numeric control and the user entered "6", then this method would return a unsigned short "6".

`mCtlLongValue` should be used for date controls.

Example:

```
object  dlg;
unsigned short  val;

val = mCtlUnsignedShortValue(dlg, PERSONS_AGE);
```

See also: `CtlStringValue`, `CtlLongValue`, ... `CtlValue`

`CtlValue::Dialog` [CtlValue]

```
val = mCtlValue(dlg, id);

object  dlg;    /*  a dialog object  */
unsigned id;    /*  control id       */
object  val;    /*  control value    */
```

This method is used to gain access to the value associated with a particular control as a Dynace object. For example, if the control was a text control and user entered "Miami", then this method would return a Dynace string object (not a normal C string) which represents the value typed in. This returned value may be used and accessed like any other Dynace object using the appropriate interface mechanism.

If the control being accessed is a list box or combo box, the value returned will be a representation of the text associated with the choice made by the user. See `IndexValue` or `CtlShortValue` to get a numeric representation for this control type.

The object returned will be automatically disposed when the dialog is disposed, therefore, a copy (via `gCopy`) should be made of it if its existence is required subsequent to the dialog's disposal.

WDS also provides several methods to obtain normal C data types directly.

Example:

```
object  dlg, val;

val = mCtlValue(dlg, PERSONS_NAME);
```

See also: `CtlStringValue`, `CtlShortValue`, `CtlLongValue`, ...

`DeepDispose::Dialog` [DeepDispose]

This method performs the same function as `Dispose`. See that method for details.

Dispose::Dialog [Dispose]

```
r = gDispose(dlg);

object  dlg;   /*  a dialog object   */
object  r;     /*  NULL              */
```

This method is used to remove and dispose of a dialog object when it is no longer needed. This method should be called on all dialogs when they are no longer needed. In addition to disposing of the dialog, this method will also dispose of all the control objects associated with the dialog.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  dlg;

dlg = gDispose(dlg);
```

See also: `AutoDispose`

GetBackBrush::Dialog [GetBackBrush]

```
brsh = gGetBackBrush(dlg);

object  dlg;    /*  dialog object  */
object  brsh;   /*  brush object   */
```

This method is used to obtain the background brush object which has been associated with the dialog. If no specific background brush was associated with the dialog this method will return the application default background brush, which is what the dialog uses when no specific brush is specified.

Example:

```
object  dlg, brsh;

brsh = gGetBackBrush(dlg);
```

See also: `BackBrush, GetTextBrush, GetBackBrush::Application`

`GetControl::Dialog` [GetControl]

```
ctl = mGetControl(dlg, id);

object  dlg;    /*  a dialog object  */
unsigned id;    /*  control id       */
object  ctl;    /*  control object   */
```

This method is used to gain access to a control object which has been previously associated with a particular control via its associated control id. The control object is returned.

Example:

```
object  dlg, ctl;

ctl = mGetControl(dlg, PERSONS_NAME);
```

See also: `AddControl`

`GetParent::Dialog` [GetParent]

```
prnt = gGetParent(dlg);

object  dlg;    /*  dialog object         */
object  prnt;   /*  parent window object  */
```

This method is used to obtain the parent window object associated with dialog `dlg`. This parent window would have established when the dialog object was created.

Example:

```
object  dlg, parentWind;

parentWind = gGetParent(dlg);
```

See also: `NewDialog::ModalDialog`, `NewDialog::ModelessDialog`

`GetTag::Dialog` [GetTag]

```
r = gGetTag(dlg);

object  dlg;   /*  dialog object  */
object  r;     /*  tag            */
```

This method is used to obtain a Dynace object which has been associated with a dialog via `SetTag`. The value return is the object which has been associated with the dialog object `dlg`. If there is no object associated with the dialog, `NULL` will be returned.

Example:

```
object  dlg, someObj;

someObj = gGetTag(dlg);
```

See also: `SetTag`, `SetTag::Window`

`GetTextBrush::Dialog` [GetTextBrush]

```
brsh = gGetTextBrush(dlg);

object  dlg;    /*  dialog object  */
object  brsh;   /*  brush object   */
```

This method is used to obtain the text brush object which has been associated with the dialog. If no specific text brush was associated with the dialog this method will return the application default text brush, which is what the dialog uses when no specific brush is specified.

Example:

```
object  dlg, brsh;

brsh = gGetTextBrush(dlg);
```

See also: `TextBrush`, `GetBackBrush`, `GetTextBrush::Application`

`Handle::Dialog` [Handle]

```
h = gHandle(dlg);

object  dlg;    /*  dialog object   */
HANDLE  h;      /*  Windows handle  */
```

This method is used to obtain the Windows internal handle associated with a dialog object. It will only return a valid handle while the dialog is being performed via `gPerform`. `NULL` will be returned otherwise.

Note that this method may be used with most WDS objects in order to obtain the internal handle that Windows normally associates with each type of object. See the appropriate documentation.

Example:

```
object  dlg;
HANDLE  h;

h = gHandle(dlg);
```

IndexValue::Dialog [IndexValue]

```
val = mIndexValue(dlg, id);

object  dlg;    /*  a dialog object  */
unsigned id;    /*  control id       */
object  val;    /*  control value    */
```

This method is used to gain access to a Dynace object (as opposed to a normal C integer) representing the value associated with a list box or combo box in terms of an ordinal value. This value returned is a zero based index from top to bottom indicating the selection made by the user. If no selection was made the object will represent a negative value. This returned value may be used and accessed like any other Dynace object using the appropriate interface mechanisms.

The object returned will be automatically disposed when the dialog is disposed, therefore, a copy (via `gCopy`) should be made of it if its existence is required subsequent to the dialog's disposal.

WDS also provides several methods to obtain normal C data types directly.

Example:

```
object  dlg, val;

val = mIndexValue(dlg, SOME_LISTBOX);
```

See also: `CtlShortValue`, `CtlValue`

InDialog::Dialog [InDialog]

```
flg = gInDialog(dlg);

object  dlg;    /*  a dialog object  */
int     flg;    /*  in dialog flag   */
```

This method is used to determine whether a given dialog is currently being "performed" via `gPerform`. Prior and subsequent to performing a dialog this method will return 0. A 1 will be returned if the dialog is currently being performed.

Example:

```
object  dlg;
int     flg;

flg = gInDialog(dlg);
```

See also: `Perform`

`Message::Dialog` [Message]

```
r = gMessage(dlg, msg);

object  dlg;    /*  dialog object  */
char    *msg;   /*  message        */
object  r;      /*  dialog object  */
```

This method is used to open up a temporary informational window. The window will contain the message given by `msg` and the user must acknowledge the window prior to continuing by hitting an OK button.

This method may only be used either while the dialog is being performed or at any time if it has an associated parent window.

The value returned is the dialog passed.

Example:

```
object  dlg;

gMessage(dlg, "Press OK to continue.");
```

See also: `MessageWithTopic`

`MessageWithTopic::Dialog` [MessageWithTopic]

```
r = gMessageWithTopic(dlg, msg, tpc);

object  dlg;    /*  dialog object  */
char    *msg;   /*  message        */
char    *tpc;   /*  help topic     */
object  r;      /*  dialog object  */
```

This method is used to open up a temporary informational dialog. The dialog will contain the message given by `msg` and the user must acknowledge the dialog prior to continuing by hitting an OK button.

If the user hits the `F1` key while presented with the message, the help topic identified by `tpc` will get displayed via the Windows help system.

This method may only be used either while the dialog is being performed or at any time if it has an associated parent window.

The value returned is the dialog passed.

Example:

```
object  dlg;

gMessageWithTopic(dlg, "Press OK to continue.", "mytopic");
```

See also: `Message` and the `HelpSystem` class.

`Perform::Dialog` [Perform]

```
r = gPerform(dlg);

object  dlg;    /*  a dialog object  */
int     r;      /*  result           */
```

Once a dialog is created and fully specified, this method is used to actually display the dialog and allow the user to interact with it. The dialog will remain active until the user pushes the push buttons with the label `IDOK` or `IDCANCEL`.

What occurs after calling `gPerform` depends on whether the dialog was a modal or modeless dialog.

Control initial values should be set prior to calling `gPerform`.

MODAL DIALOGS

If the dialog being performed is a modal, then once `gPerform` is evoked the user will be able to interact with the dialog and `gPerform` will not return until the user either accepts or aborts the entire dialog. This, in effect, will prevent the user from switching to any other window or dialog within the application until the dialog is completed.

Finally, the value returned by `gPerform` will be the Windows constant `FALSE` if the user canceled the dialog, the Windows constant `TRUE` if the user accepted the dialog, or the value returned by any completion function which was attached to the dialog (via `CompletionFunction`).

Once `gPerform` returns, the dialog object may be queried in order to obtain the final values associated with each control.

Unless the auto dispose feature is manually set for a modal dialog, the dialog object is not automatically disposed and must be when no longer needed.

MODELESS DIALOGS

Modeless dialogs operate different from modal dialogs. When `gPerform` is evoked on a modeless dialog, `gPerform` returns immediately with a 0 value. At that point the user is presented with the dialog and allowed to edit it. During that time your program would typically return to the menu and wait for further user selections. The user may continue with the current dialog, abort it, or switch to another window within the application with the ability to return at any point. All the user interaction is automatically handled by Windows and WDS.

Since the application code has gone on about its business after a call to `gPerform` two things are left to be completed whenever the user finally accepts or aborts the dialog. First, the application will typically want to do something with the data the user has entered on the dialog. And second, the dialog object will need to be disposed when it's no longer needed.

When using modeless dialogs the programmer normally attaches a completion function to the dialog via `gCompletionFunction`. What this does is cause the completion function to be executed whenever the user accepts or aborts the dialog. This application specific function can be used to perform the final processing of the dialog's data. See `gCompletionFunction` for complete details.

By default, the auto dispose flag associated with modeless dialogs is normally enabled. This causes WDS to automatically dispose of a dialog object and all associated controls immediately subsequent to executing the completion function. This way everything is cleaned up automatically. This feature may be disabled via `gAutoDispose`, in which case the application must be sure to dispose of the unneeded object.

Example:

```
object  dlg;
int     r;

r = gPerform(dlg);
```

See also: `CompletionFunction, AutoDispose`

`SetResult::Dialog` [SetResult]

```
r = gSetResult(dlg, obj);

object  dlg;    /*  a dialog object  */
object  obj;    /*  result object    */
object  r;      /*  dlg              */
```

This method is used to associated a Dynace short integer object with a dialog such that when the dialog is completed or canceled by the user, the associated object will be set to the dialog's result value. This feature is most often needed in conjunction with modeless dialogs and the auto dispose feature. It can be used to find out what happened to a dialog subsequent to the dialog object being disposed.

The object associated with the dialog will never be disposed by WDS and must be manually disposed when it is no longer needed.

Example:

```
object  dlg, obj;

gSetResult(dlg, obj = gNewWithInt(ShortInteger, -1));
```

See also: `AutoDispose`, `Perform`, `SetTag`

`SetTag::Dialog` [SetTag]

```
r = gSetTag(dlg, tag);

object  dlg;   /*  a dialog object   */
object  tag;    /*  tag               */
object  r;      /*  previous tag      */
```

This method is used to associated an arbitrary Dynace object with a dialog object. This may later be retrieved via the `GetTag` method. Since WDS passes around the dialog object to all `Dialog` methods this mechanism may be used to pass additional information with the dialog. And since Dynace treats all objects in a uniform manner, this information attached to the dialog may be arbitrarily complex.

WDS does not dispose of the tag when the dialog object is disposed. This method returns any previous object associated with the dialog or `NULL`.

Example:

```
object  dlg;

gSetTag(dlg, gNewWithInt(ShortInteger, 17));
```

See also: `GetTag`, `SetTag::Window`

SetTopic::Dialog [SetTopic]

```
pt = gSetTopic(dlg, tpc);

object  dlg;    /*  dialog object         */
char    *tpc;   /*  help topic            */
char    *pt;    /*  previous help topic   */
```

This method is used to associate help text with dialog `dlg`. The help text is defined using the Windows help system and labeled with the topic indicated by `tpc`. Then, if the user hits the F1 key while in the dialog, WDS will automatically bring up the Windows help system and find the indicated topic.

WDS also supports window and control specific topics. See the appropriate sections.

This method returns any previous topic associated with the dialog.

Example:

```
object  dlg;

gSetTopic(dlg, "myDialogHelp");
```

See also: The `HelpSystem` class and `SetTopic::Control`

TextBrush::Dialog [TextBrush]

```
r = gTextBrush(dlg, brsh);

object  dlg;    /*  a dialog object   */
object  brsh;   /*  brush object      */
object  r;      /*  dlg               */
```

This method is used to set the brush object which is used for foreground text which is displayed. Any previously associated brush object will be disposed. This brush object will also be automatically disposed when the dialog is disposed.

The dialog passed is returned.

Example:

```
object  dlg;

gTextBrush(dlg, vNew(SolidBrush, 255, 0, 0));
```

See also: `BackBrush`, `SetTextBrush::Application` and the `Brush` classes

4.8.3 Modal Dialogs

Modal dialogs are created with the methods described in this section. However, since the `ModalDialog` class is a subclass of `Dialog`, and the `Dialog` class implements all the functionality common to both the `ModalDialog` and `ModelessDialog` classes, the majority of the functionality is inherited from and documented in the `Dialog` class. Therefore, see the `Dialog` class for documentation on additional functionality available to this class.

`NewDialog::ModalDialog` [NewDialog]

```
dlg = mNewDialog(ModalDialog, id, wnd);

unsigned  id;   /*  dialog id      */
object    wnd;  /*  parent window  */
object    dlg;  /*  dialog object  */
```

This method is used to create a new modal dialog. The dialog must have been previously laid out using the resource editor. The `id` is a macro generated by the resource editor while the programmer defines the dialog. This id uniquely identifies a particular dialog.

`wnd` allows the specification of the parent window object associated with the new dialog. This is an optional parameter and may be `NULL` if it not desired. If you do specify a parent window, the dialog will iconize when the window is iconized and will be disposed if the window is disposed.

The object returned represents the new dialog created and may be used to further control the dialog.

Example:

```
object  wind, dlg;

dlg = mNewDialog(ModalDialog, MY_DIALOG, wind);
```

See also: `NewDialogStr`, `Perform::Dialog`, `NewDialog::ModelessDialog`

`NewDialogStr::ModalDialog` [NewDialogStr]

```
dlg = gNewDialogStr(ModalDialog, id, wnd);

char      *id;  /*  dialog id      */
object    wnd;  /*  parent window  */
object    dlg;  /*  dialog object  */
```

This method is used to create a new modal dialog. The dialog must have been previously laid out using the resource editor. The `id` is the string name associated with the dialog. This id uniquely identifies a particular dialog.

`wnd` allows the specification of the parent window object associated with the new dialog. This is an optional parameter and may be `NULL` if it not desired. If you do specify a parent window, the dialog will iconize when the window is iconized and will be disposed if the window is disposed.

The object returned represents the new dialog created and may be used to further control the dialog.

Example:

```
object  wind, dlg;

dlg = gNewDialogStr(ModalDialog, "mydialog", wind);
```

See also: `NewDialog`, `Perform::Dialog`, `NewDialogStr::ModelessDialog`

4.8.4 Modeless Dialogs

Modeless dialogs are created with the methods described in this section. However, since the `ModelessDialog` class is a subclass of `Dialog`, and the `Dialog` class implements all the functionality common to both the `ModalDialog` and `ModelessDialog` classes, the majority of the functionality is inherited from and documented in the `Dialog` class. Therefore, see the `Dialog` class for documentation on additional functionality available to this class.

`NewDialog::ModelessDialog` [NewDialog]

```
dlg = mNewDialog(ModelessDialog, id, wnd);

unsigned  id;   /*  dialog id      */
object    wnd;  /*  parent window  */
object    dlg;  /*  dialog object  */
```

This method is used to create a new modeless dialog. The dialog must have been previously laid out using the resource editor. The `id` is a macro generated by the resource editor while the programmer defines the dialog. This id uniquely identifies a particular dialog.

`wnd` allows the specification of the parent window object associated with the new dialog. This is an optional parameter and may be `NULL` if it not desired. If you do specify a parent window, the dialog will iconize when the window is iconized and will be disposed if the window is disposed.

When a new modeless dialog is created, its auto dispose mode is enabled. This means that, unless disabled, the object representing the dialog will be automatically disposed when the user closes the window. This is done due to the independent nature of modeless dialogs and the need to be sure that object no longer needed are disposed. Note that the programmer may specify a function which gets executed prior to the disposal of a modeless dialog in order to capture any data.

The object returned represents the new dialog created and may be used to further control the dialog.

Example:

```
object  wind, dlg;

dlg = mNewDialog(ModelessDialog, MY_DIALOG, wind);
```

See also: `NewDialogStr`, `Perform::Dialog`, `NewDialog::ModalDialog`, `CompletionFunction::Dialog`, `AutoDispose::Dialog`

`NewDialogStr::ModelessDialog` [NewDialogStr]

```
dlg = gNewDialogStr(ModlessDialog, id, wnd);

char      *id;  /*  dialog id      */
object    wnd;  /*  parent window  */
object    dlg;  /*  dialog object  */
```

This method is used to create a new modeless dialog. The dialog must have been previously laid out using the resource editor. The `id` is the string name associated with the dialog. This id uniquely identifies a particular dialog.

`wnd` allows the specification of the parent window object associated with the new dialog. This is an optional parameter and may be `NULL` if it not desired. If you do specify a parent window, the dialog will iconize when the window is iconized and will be disposed if the window is disposed.

When a new modeless dialog is created, its auto dispose mode is enabled. This means that, unless disabled, the object representing the dialog will be automatically disposed when the user closes the window. This is done due to the independent nature of modeless dialogs and the need to be sure that object no longer needed are disposed. Note that the programmer may specify a function which gets executed prior to the disposal of a modeless dialog in order to capture any data.

The object returned represents the new dialog created and may be used to further control the dialog.

Example:

```
object  wind, dlg;

dlg = gNewDialogStr(ModelessDialog, "mydialog", wind);
```

See also: `NewDialog`, `Perform::Dialog`, `NewDialogStr::ModalDialog`, `CompletionFunction::Dialog`, `AutoDispose::Dialog`

4.9 Controls

The `Control` class is never used directly. It is used to group functionality common to all the control types which are all subclasses of this class. Therefore, this section documents all the methods which are accessible to all subclasses of this class.

Control instances are normally created via `AddControl::Dialog` and may be subsequently accessed via `GetControl::Dialog`.

4.9.1 Standard Control Method Arguments

Since all control instance methods have the control object as their first argument (referred to as `ctl`), this argument will not be described each time. It always refers to the control object which you wish to perform the desired operation on. Each control object will be an instance of one of the subclasses of the `Control` class (such as `TextControl` or `NumericControl`).

Many of the methods associated with this class take an argument identified as `id`. This is a macro defined by the programmer via the resource editor when the dialog is being defined and is used to uniquely identify a particular control within the dialog. Due to the fact that this argument has the same meaning for every method which uses it, it will not be defined each time.

4.9.2 Control Methods

`AddHandlerAfter::Control` [AddHandlerAfter]

```
r = gAddHandlerAfter(ctl, msg, func);

object   ctl;      /*  a control object  */
unsigned msg;      /*  message           */
long    (*func)(); /*  function pointer  */
object  r;         /*  the control obj   */
```

This method is used to associate function `func` with Windows control message `msg` for control `ctl`. Whenever control `ctl` receives message `msg`, `func` will be called.

`ctl` is the control object who's messages you wish to process. `msg` is the particular message you wish to trap. These messages are fully documented in the Windows documentation in the Messages section. They normally begin with `WM_`.

`func` is the function which gets called whenever the specified message gets received and takes the following form:

```
long    func(object     ctl,
             HWND       hwnd,
             UINT       mMsg,
             WPARAM     wParam,
             LPARAM     lParam)
{
        .
        .
        .
        return 0L;  /* or whatever is appropriate  */
}
```

Where `ctl` is the control being sent the message. The remaining arguments and return value is fully documented in the Windows documentation under the `WindowProc` function and the Windows Messages documentation.

WDS keeps a list of functions associated with each message associated with each control. When a particular message is received the appropriate list of handler functions gets executed sequentially. `AddHandlerAfter` appends the new function to the end of this list, and `AddHandlerBefore` adds the new function to the beginning of the list.

WDS may also, and optionally, execute the Windows default procedure associated with a given message either before or after the user added list of functions. This behavior may be controlled via `DefaultProcessingMode`.

Windows will only see the return value of the last message handler executed including, if applicable, the default.

Example:

```
int     hSize, vSize;

static  long    process_wm_size(object  ctl,
                                HWND    hwnd,
                                UINT    mMsg,
                                WPARAM  wParam,
                                LPARAM  lParam)
{
        hSize = LOWORD(lParam);
        vSize = HIWORD(lParam);
        return 0L;
}


        .
        .
        gAddHandlerAfter(ctl, (unsigned) WM_SIZE, process_wm_size);
        .
        .
```

See also: `DefaultProcessingMode`, `AddHandlerBefore`, `CallDefaultProc`

`AddHandlerBefore::Control` [AddHandlerBefore]

```
r = gAddHandlerBefore(ctl, msg, func);

object   ctl;      /*  a control object  */
unsigned msg;      /*  message           */
long    (*func)(); /*  function pointer  */
object  r;         /*  the control obj   */
```

This function is fully documented under `AddHandlerAfter`.

See also: `AddHandlerAfter`

`CallDefaultProc::Control` [CallDefaultProc]

```
r = gCallDefaultProc(ctl, msg, wp, lp)

object   ctl;   /*  a control object  */
unsigned msg;   /*  message           */
WPARAM   wp;    /*  wParam value      */
LPARAM   lp;    /*  lParam value      */
long     r;     /*  result of call    */
```

This method is used to explicitly call the default Windows message handler associated with message `msg`. This method is normally called from within a programmer defined message handler (see `AddHandlerAfter`) which is provided with the `wp` and `lp` parameters. These parameters are fully documented in the Windows Messages documentation and relate directly to the particular message.

`msg` is the particular message you wish to affect. These messages are fully documented in the Windows documentation in the Messages section. They normally begin with `WM_`.

Example:

```
gCallDefaultProc(ctl, msg, wParam, lParam);
```

See also: `DefaultProcessingMode`, `AddHandlerAfter`

`CheckFunction::Control` [CheckFunction]

```
r = gCheckFunction(ctl, fun);

object  ctl;      /*  control object  */
int     (*fun)(); /*  check function  */
object  r;        /*  ctl object      */
```

This method is used to associate an auxiliary checking function to a control. When a dialog is accepted, WDS goes through each control to assure the validity of the data associated with each control as defined by the control object. The ability to associated an additional function for validating the data associated with a control gives the programmer the ability to perform any additional application specific checking against a control's data.

The auxiliary checking function takes the following form:

```
int     fun(object ctl, object val, char *buf)
{
        ....
}
```

Where `ctl` is the control object, and `val` is a Dynace object which represents the value the control has associated with it. `buf` is a pointer to a buffer area which `fun` must set to an appropriate error message if an error occurs. `fun` returns a 1 if an error occurs and 0 otherwise.

All controls support this feature except the `PushButton` class, because it is an immediate action control with no associated value.

Example:

```
object  ctl;

gCheckFunction(ctl, fun);
```

See also: `GetControl::Dialog`

`CheckValue::Control` [CheckValue]

```
r = gCheckValue(ctl)

object   ctl;   /*  a control object  */
int      r;     /*  result of call    */
```

This method is used to explicitly cause all error checking functions and parameters associated with control `ctl` to be checked. If an error is encountered an appropriate

error message will be displayed and a 1 will be returned. If no error is encountered a 0 is returned.

This method is implemented by all subclasses of `Control` and mainly used internally when the dialog is accepted.

Example:

```
gCheckValue(ctl);
```

See also: `CheckFunction::Control`

`DeepDispose::Control` [DeepDispose]

This method performs the same function as `Dispose`. See that method for details.

`DefaultProcessingMode::Control` [DefaultProcessingMode]

```
r = gDefaultProcessingMode(ctl, msg, mode);

object   ctl;   /*  a control object         */
unsigned msg;   /*  message                  */
int      mode;  /*  default processing mode  */
object   r;     /*  the control obj          */
```

This method is used to determine when or if the Windows default message procedure is processed for a given message (`msg`) associated with a particular control (`ctl`).

WDS allows a programmer to specify an arbitrary number of functions to be executed whenever a control receives a specific message (via `AddHandlerAfter` and `AddHandlerBefore`). Windows has default procedures associated with many control messages. At times it is necessary to replace or augment this default functionality. `DefaultProcessingMode` gives the programmer control over when and if this default Windows functionality. `mode` is used to specify the desired mode. The following table indicates the valid modes:

`0`	Do not execute the Windows default processing
`1`	Execute default processing *after* programmer defined handlers
`2`	Execute default processing *before* programmer defined handlers

Note that the default mode is always `1`, and must be explicitly changed, if desired, for each message associated with each control.

`msg` is the particular message you wish to affect. These messages are fully documented in the Windows documentation in the Messages section. They normally begin with `WM_`.

Example:

```
gDefaultProcessingMode(ctl, (unsigned) WM_SIZE, 0);
```

See also: `CallDefaultProc`, `AddHandlerAfter`

`Dispose::Control` [Dispose]

```
r = gDispose(ctl);

object  ctl;   /*  a control object  */
object  r;     /*  NULL              */
```

This method is used to remove and dispose of a control object when it is no longer needed. This method is rarely needed due to the fact that when a dialog is disposed it automatically calls this method on each of its associated controls.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  ctl;

ctl = gDispose(ctl);
```

See also: `DeepDispose`

`Handle::Control` [Handle]

```
h = gHandle(ctl);

object  ctl;    /*  control object   */
HANDLE  h;      /*  Windows handle  */
```

This method is used to obtain the Windows internal handle associated with a control object. It will only return a valid handle while the dialog is being performed via `gPerform`. `NULL` will be returned otherwise.

Note that this method may be used with most WDS objects in order to obtain the internal handle that Windows normally associates with each type of object. See the appropriate documentation.

Example:

```
object  ctl;
HANDLE  h;

h = gHandle(ctl);
```

See also: `GetControl::Dialog`

`SetTopic::Control` [SetTopic]

```
pt = gSetTopic(ctl, tpc);

object  ctl;    /*  control object        */
char    *tpc;   /*  help topic            */
char    *pt;    /*  previous help topic   */
```

This method is used to associate help text with specific control `ctl`. The help text is defined using the Windows help system and labeled with the topic indicated by `tpc`. Then, if the user hits the F1 key while in the control, WDS will automatically bring up the Windows help system and find the indicated topic. If no control specific help is specified, WDS will use any help associated with the dialog as a whole.

WDS also supports dialog and control specific topics. See the appropriate sections.

This method returns any previous topic associated with the control.

Example:

```
object  ctl;

gSetTopic(ctl, "myControlHelp");
```

See also: The `HelpSystem` class and `SetTopic::Dialog`

4.9.3 Text Control

The `TextControl` class is a control which enables the user to enter arbitrary alphanumeric data on a single line. This control has numerous facilities to control the length and type of input the user is allowed to enter.

The `TextControl` class is a subclass of `Control` and as such inherits all the functionality associated with that class. This section only documents functionality particular to this class.

Standard control arguments are documented in the section entitled “Standard Control Method Arguments”.

Attach::TextControl [Attach]

```
r = gAttach(ctl, val);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
object  r;     /*  control object  */
```

This method is used to associate an independent Dynace object with the value associated with a control object. `val` should be an instance of the `String` class and will be automatically updated to reflect the value associated with the control.

This object (`val`) will never be disposed by WDS, even after the control or associated dialog are disposed. Therefore, this is one way of gaining access to a control's value after the life of the control. It is the programmer's responsibility to dispose of the object when it is no longer needed.

Example:

```
object  ctl, val;

val = gNew(String);
gAttach(ctl, val);
```

See also: `Value, StringValue, SetValue`

Capitalize::TextControl [Capitalize]

```
r = gCapitalize(ctl);

object  ctl;   /*  control object  */
object  r;     /*  control object  */
```

This method is used to enable the auto-capitalize feature of the text control. If enabled, all alpha entry made by the user will be converted to upper case both on the display and internally to the control.

Example:

```
object  ctl;

gCapitalize(ctl);
```

See also: `TextRange`

MaxLength::TextControl [MaxLength]

```
r = gMaxLength(ctl, len);

object  ctl;   /*  control object  */
int     len;   /*  maximum length  */
object  r;     /*  control object  */
```

This method is used to set the maximum number of characters the user may enter into a text control. If the user attempts to enter more than `len` characters, including spaces, the system will just beep. If the number of characters exceeds the field display width, the display will scroll.

Example:

```
object  ctl;

gMaxLength(ctl, 25);
```

See also: `MinLength, TextRange, Capitalize, CheckFunction::Control`

MinLength::TextControl [MinLength]

```
r = gMinLength(ctl, len);

object  ctl;   /*  control object  */
int     len;   /*  minimum length  */
object  r;     /*  control object  */
```

This method is used to set the minimum number of characters the user must enter into a text control. If the user attempts to accept the dialog without entering `len` characters, including spaces, WDS will issue an error message and return the user to the incomplete field.

Example:

```
object  ctl;

gMinLength(ctl, 5);
```

See also: `MaxLength, TextRange, Capitalize, CheckFunction::Control`

NewCtl::TextControl [NewCtl]

```
ctl = mNewCtl(TextControl, id);

unsigned  id;   /*  control id      */
object   ctl;   /*  control object  */
```

This method is used to create a new control object to be identified as `id` (see section "Standard Control Method Arguments"). This method is mainly used internally. A programmer would more often use `AddControl::Dialog` to create controls and associated them to a dialog.

Example:

```
object  ctl;

ctl = mNewCtl(TextControl, PERSON_NAME);
```

See also: `AddControl::Dialog`, `GetControl::Dialog`

SetStringValue::TextControl [SetStringValue]

```
r = gSetStringValue(ctl, val);

object  ctl;    /*  control object  */
char    *val;   /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be a pointer to a character string which represents the desired value for the control.

Any previously associated value object will be disposed when the new value is set.

Example:

```
object  ctl;

gSetStringValue(ctl, "Some value");
```

See also: `SetValue`, `StringValue`

`SetValue::TextControl` [SetValue]

```
r = gSetValue(ctl, val);

object  ctl;    /*  control object  */
object  val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be a Dynace object which is an instance of the `String` class and initialized to the value desired for the control.

Any previously associated object will be disposed when the new value is set. Also, `val` will automatically be disposed when the control or associated dialog is disposed.

Example:

```
object  ctl;

gSetValue(ctl, gNewWithStr(String, "Some value"));
```

See also: `SetStringValue`, `Value`, `Attach`

`StringValue::TextControl` [StringValue]

```
val = gStringValue(ctl);

object  ctl;   /*  control object  */
char    *val;  /*  ctl value       */
```

This method is used to obtain a C character pointer which represents the value associated with the control. This pointer will not be valid after the control or associated dialog are disposed.

Example:

```
object  ctl;
char    *val;

val = gStringValue(ctl);
```

See also: `Value`, `Attach`, `SetStringValue`, `CtlStringValue::Dialog`

TextRange::TextControl [TextRange]

```
r = gTextRange(ctl, min, max);

object  ctl;   /*  control object  */
int     min;   /*  minimum length  */
int     max;   /*  maximum length  */
object  r;     /*  control object  */
```

This method is used to set the minimum number of characters the user must enter into a text control as well as the maximum number of characters allowed.

If the user attempts to enter more than `max` characters, including spaces, the system will just beep. If the number of characters exceeds the field display width, the display will scroll.

If the user attempts to accept the dialog without entering `min` characters, including spaces, WDS will issue an error message and return the user to the incomplete field.

Example:

```
object  ctl;

gTextRange(ctl, 5, 25);
```

See also: `MaxLength`, `MinLength`, `Capitalize`, `CheckFunction::Control`

Value::TextControl [Value]

```
val = gValue(ctl);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
```

This method is used to obtain a Dynace object which represents the value associated with the control. The object returned will be an instance of the `String` class. This object will be disposed by WDS when the control object or associated dialog is disposed.

Example:

```
object  ctl, val;

val = gValue(ctl);
```

See also: `StringValue`, `Attach`, `SetValue`

4.9.4 Numeric Control

The `NumericControl` class is a control which enables the user to enter arbitrary numeric data. This control has numerous facilities to control the range and type of input the user is allowed to enter.

The `NumericControl` class is a subclass of `Control` and as such inherits all the functionality associated with that class. This section only documents functionality particular to this class.

Standard control arguments are documented in the section entitled "Standard Control Method Arguments".

`Attach::NumericControl` [Attach]

```
r = gAttach(ctl, val);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
object  r;     /*  control object  */
```

This method is used to associate an independent Dynace object with the value associated with a control object. `val` should be an instance of one of the Dynace numeric classes which are subclasses of the `Number` class, such as `ShortInteger` or `DoubleFloat`. This numeric object will be automatically updated to reflect the value associated with the control.

This object (`val`) will never be disposed by WDS, even after the control or associated dialog are disposed. Therefore, this is one way of gaining access to a control's value after the life of the control. It is the programmer's responsibility to dispose of the object when it is no longer needed.

Example:

```
object  ctl, val;

val = gNewWithDouble(DoubleFloat, 0.0);
gAttach(ctl, val);
```

See also: `Value, DoubleValue, ShortValue, SetValue`

`DoubleValue::NumericControl` [DoubleValue]

```
val = gDoubleValue(ctl);

object  ctl;   /*  control object  */
double  val;   /*  ctl value       */
```

This method is used to obtain a C double which represents the value associated with the control.

Example:

```
object  ctl;
double  val;

val = gDoubleValue(ctl);
```

See also: `Value`, `Attach`, `SetDoubleValue`, `CtlDoubleValue::Dialog` `UnsignedShortValue`, `ShortValue`, `LongValue`

`LongValue::NumericControl` [LongValue]

```
val = gLongValue(ctl);

object  ctl;   /*  control object  */
long    val;   /*  ctl value       */
```

This method is used to obtain a C long integer which represents the value associated with the control.

Example:

```
object  ctl;
long    val;

val = gLongValue(ctl);
```

See also: `Value`, `Attach`, `SetLongValue`, `CtlLongValue::Dialog` `UnsignedShortValue`, `ShortValue`, `DoubleValue`

`NewCtl::NumericControl` [NewCtl]

```
ctl = mNewCtl(NumericControl, id);

unsigned  id;   /*  control id      */
object   ctl;   /*  control object  */
```

This method is used to create a new control object to be identified as `id` (see section "Standard Control Method Arguments"). This method is mainly used internally. A programmer would more often use `AddControl::Dialog` to create controls and associated them to a dialog.

Example:

```
object  ctl;

ctl = mNewCtl(NumericControl, PERSON_AGE);
```

See also: `AddControl::Dialog`, `GetControl::Dialog`

`NumericRange::NumericControl` [NumericRange]

```
r = gNumericRange(ctl, min, max, dp);

object  ctl;   /*  control object  */
double  min;   /*  minimum value   */
double  max;   /*  maximum value   */
int     dp;    /*  decimal places  */
object  r;     /*  control object  */
```

This method is used to set the minimum and maximum allowable values the control will accept from the user. It also sets the number of digits to the right of the decimal point the control will accept.

If the user enters a number outside the specified range, WDS will issue an error message and prompt the user for an acceptable value.

Example:

```
object  ctl;

gNumericRange(ctl, 5.0, 25.0, 1);
```

See also: `CheckFunction::Control`

`SetDoubleValue::NumericControl` [SetDoubleValue]

```
r = gSetDoubleValue(ctl, val);

object  ctl;    /*  control object  */
double  val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be the value which represents the desired value for the control.

Any previously associated value object (associated via `SetValue`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetDoubleValue(ctl, 3.14159265358979);
```

See also: `SetValue`, `DoubleValue`, `CtlDoubleValue::Dialog`, `SetUShortValue`, `SetShortValue`, `SetLongValue`

`SetLongValue::NumericControl` [SetLongValue]

```
r = gSetLongValue(ctl, val);

object  ctl;    /*  control object  */
long    val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be the value which represents the desired value for the control.

Any previously associated value object (associated via `SetValue`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetLongValue(ctl, 66L);
```

See also: `SetValue`, `LongValue`, `CtlLongValue::Dialog`, `SetUShortValue`, `SetShortValue`, `SetDoubleValue`

`SetShortValue::NumericControl` [SetShortValue]

```
r = gSetShortValue(ctl, val);

object  ctl;    /*  control object  */
int     val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be the value which represents the desired value for the control.

Any previously associated value object (associated via `SetValue`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetShortValue(ctl, 66);
```

See also: `SetValue`, `ShortValue`, `CtlShortValue::Dialog`, `SetUShortValue`, `SetLongValue`, `SetDoubleValue`

`SetUShortValue::NumericControl` [SetUShortValue]

```
r = gSetUShortValue(ctl, val);

object    ctl;  /*  control object  */
unsigned  val;  /*  ctl value       */
object    r;    /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be the value which represents the desired value for the control.

Any previously associated value object (associated via `SetValue`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetUShortValue(ctl, 66);
```

See also: `SetValue`, `UnsignedShortValue`, `CtlUnsignedShortValue::Dialog` `SetShortValue`, `SetLongValue`, `SetDoubleValue`

`SetValue::NumericControl` [SetValue]

```
r = gSetValue(ctl, val);

object  ctl;    /*  control object  */
object  val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be a Dynace object which is an instance of one of the subclasses of the `Number` class and initialized to the value desired for the control.

Any previously associated object will be disposed when the new value is set. Also, `val` will automatically be disposed when the control or associated dialog is disposed.

Example:

```
object  ctl;

gSetValue(ctl, gNewWithDouble(DoubleFloat, 3.141));
```

See also: `Value`, `SetShortValue`, `SetUShortValue` `SetLongValue`, `SetDoubleValue`

`ShortValue::NumericControl` [ShortValue]

```
val = gShortValue(ctl);

object  ctl;   /*  control object  */
short   val;   /*  ctl value       */
```

This method is used to obtain a C short integer which represents the value associated with the control.

Example:

```
object  ctl;
short   val;

val = gShortValue(ctl);
```

See also: `Value`, `Attach`, `SetShortValue`, `CtlShortValue::Dialog` `UnsignedShortValue`, `LongValue`, `DoubleValue`

`UnsignedShortValue::NumericControl` [UnsignedShortValue]

```
val = gUnsignedShortValue(ctl);

object  ctl;           /*  control object  */
unsigned short  val;   /*  ctl value       */
```

This method is used to obtain a C unsigned short integer which represents the value associated with the control.

Example:

```
object  ctl;
unsigned short  val;

val = gUnsignedShortValue(ctl);
```

See also: `Value, Attach, SetUShortValue, CtlUnsignedShortValue::Dialog ShortValue, LongValue, DoubleValue`

`Value::NumericControl` [Value]

```
val = gValue(ctl);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
```

This method is used to obtain a Dynace object which represents the value associated with the control. The object returned will be an instance of one of the subclasses of the `Number` class. This object will be disposed by WDS when the control object or associated dialog is disposed.

Example:

```
object  ctl, val;

val = gValue(ctl);
```

See also: `ShortValue, DoubleValue, Attach, SetValue`

4.9.5 Date Control

The `DateControl` class is a control which enables the user to enter date information. The user enters dates in the form mm/dd/yy or mm/dd/yyyy, and when the user leaves the field it changes to the form MMM DD, YYYY. For example, the user would enter 6/8/59 and when the field is accepted it will be changed to display Jun 8, 1959.

Dates are stored internally as C longs in the form YYYYMMDD so that 6/8/59 would be represented as the long 19590608. This control has numerous facilities to control the range and type of input the user is allowed to enter.

The `DateControl` class is a subclass of `Control` and as such inherits all the functionality associated with that class. This section only documents functionality particular to this class.

Standard control arguments are documented in the section entitled “Standard Control Method Arguments”.

`Attach::DateControl` [Attach]

```
r = gAttach(ctl, val);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
object  r;     /*  control object  */
```

This method is used to associate an independent Dynace object with the value associated with a control object. `val` should be an instance of the `Date` class. This date object will be automatically updated to reflect the value associated with the control.

This object (`val`) will never be disposed by WDS, even after the control or associated dialog are disposed. Therefore, this is one way of gaining access to a control's value after the life of the control. It is the programmer's responsibility to dispose of the object when it is no longer needed.

Example:

```
object  ctl, val;

val = vNew(Date, 0L);
gAttach(ctl, val);
```

See also: `Value, LongValue, SetValue, SetLongValue`

`DateRange::DateControl` [DateRange]

```
r = gDateRange(ctl, min, max, an);

object  ctl;   /*  control object  */
long    min;   /*  minimum value   */
long    max;   /*  maximum value   */
int     an;    /*  allow none      */
object  r;     /*  control object  */
```

This method is used to set the minimum and maximum allowable values the control will accept from the user. It also controls whether or not the control will accept no date.

`min` and `max` are of the form YYYYMMDD. For example 6/8/59 would be represented as 19590608. If `an` is 0 the user must enter a valid date between `min` and `max`. If, however, `an` is 1, the user must either enter a valid date between the given ranges or may also leave the field blank. If the user does not enter a date the field value will be 0L.

If the user enters an invalid date or one outside the specified range, WDS will issue an error message and prompt the user for an acceptable value.

Example:

```
object  ctl;

gDateRange(ctl, 19990101L, 19991231L, 1);
```

See also: `CheckFunction::Control`

`LongValue::DateControl` [LongValue]

```
val = gLongValue(ctl);

object  ctl;   /*  control object  */
long    val;   /*  ctl value       */
```

This method is used to obtain a C long integer which represents the value associated with the control. It will be in the form YYYYMMDD. For example 6/8/59 would be represented as 19590608.

Example:

```
object  ctl;
long    val;

val = gLongValue(ctl);
```

See also: `Value`, `Attach`, `SetLongValue`, `CtlLongValue::Dialog`

`NewCtl::DateControl` [NewCtl]

```
ctl = mNewCtl(DateControl, id);

unsigned  id;   /*  control id      */
object   ctl;   /*  control object  */
```

This method is used to create a new control object to be identified as `id` (see section "Standard Control Method Arguments"). This method is mainly used internally. A programmer would more often use `AddControl::Dialog` to create controls and associated them to a dialog.

Example:

```
object  ctl;

ctl = mNewCtl(DateControl, PERSON_BD);
```

See also: `AddControl::Dialog`, `GetControl::Dialog`

`SetLongValue::DateControl` [SetLongValue]

```
r = gSetLongValue(ctl, val);

object  ctl;    /*  control object  */
long    val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be the value which represents the desired value for the control of the form YYYYMMDD. For example 6/8/59 would be 19590608.

Any previously associated value object (associated via `SetValue`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetLongValue(ctl, 19990101L);
```

See also: `SetValue`, `LongValue`, `CtlLongValue::Dialog`

`SetValue::DateControl` [SetValue]

```
r = gSetValue(ctl, val);

object  ctl;    /*  control object  */
object  val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be a Dynace object which is an instance of the `Date` class and initialized to the value desired for the control.

Any previously associated object will be disposed when the new value is set. Also, `val` will automatically be disposed when the control or associated dialog is disposed.

Example:

```
object  ctl;

gSetValue(ctl, vNew(Date, 19940101L));
```

See also: `SetLongValue`, `Value`

`Value::DateControl` [Value]

```
val = gValue(ctl);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
```

This method is used to obtain a Dynace object which represents the value associated with the control. The object returned will be an instance of the `Date` class. This object will be disposed by WDS when the control object or associated dialog is disposed.

Example:

```
object  ctl, val;

val = gValue(ctl);
```

See also: `LongValue`, `Attach`, `SetValue`

4.9.6 Push Buttons

The `PushButton` class represents a control which allows the user to perform an immediate action. That means that there is no particular data associated with the control. The user pushes the button and an associated action gets performed at that point. The `PushButton` class allows the programmer to associate a C function to the control such that whenever the user clicks on the button the specified function will be evoked.

The `PushButton` class is a subclass of `Control` and as such inherits all the functionality associated with that class. This section only documents functionality particular to this class.

Standard control arguments are documented in the section entitled “Standard Control Method Arguments”.

NewCtl::PushButton [NewCtl]

```
ctl = mNewCtl(PushButton, id);

unsigned  id;   /*  control id      */
object   ctl;   /*  control object  */
```

This method is used to create a new control object to be identified as `id` (see section "Standard Control Method Arguments"). This method is mainly used internally. A programmer would more often use `AddControl::Dialog` to create controls and associated them to a dialog.

Example:

```
object  ctl;

ctl = mNewCtl(PushButton, MY_BUTTON);
```

See also: `AddControl::Dialog`, `GetControl::Dialog`

Perform::PushButton [Perform]

```
r = gPerform(ctl);

object  ctl;      /*  control object  */
int     r;        /*  result          */
```

This method is used to manually evoke the function a programmer associated with a push button. The value returned is the result of the function attached to the button.

This method is seldom needed because WDS automatically evokes the function when the user clicks the button.

Example:

```
object  ctl;

gPerform(ctl);
```

See also: `SetFunction`

`SetFunction::PushButton` [SetFunction]

```
r = gSetFunction(ctl, fun);

object  ctl;      /*  control object          */
int     (*fun)(); /*  check function          */
ofun    r;        /*  previous check function */
```

This method is used to associate a C function to a push button such that if the user clicks the button the C function will immediately get evoked. The value returned is the previous function, if any, associated with the control.

The C function takes the following form:

```
int     fun(object ctl, object dlg)
{
        ....
}
```

Where `ctl` is the control object, and `dlg` is the object representing the dialog the control is associated with. The value returned by `fun` only has meaning if the button was either `IDOK` or `IDCANCEL` (the ones normally used to exit the dialog). If the return value is 1 then the dialog will exit like normal. If the return value is 0 then the dialog will not exit.

Example:

```
object  ctl;

gSetFunction(ctl, fun);
```

See also: `Perform`

4.9.7 Check Boxes

The `CheckBox` class represents a control which allows the user to select or de-select a particular option. Unlike radio buttons, check boxes are not mutually exclusive.

The `CheckBox` class is a subclass of `Control` and as such inherits all the functionality associated with that class. This section only documents functionality particular to this class.

Standard control arguments are documented in the section entitled “Standard Control Method Arguments”.

`Attach::CheckBox` [Attach]

```
r = gAttach(ctl, val);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
object  r;     /*  control object  */
```

This method is used to associate an independent Dynace object with the state associated with a control object. `val` should be an instance of the `ShortInteger` class. This object will be automatically updated to reflect the state associated with the control.

The object will represent 0 if the box is not checked, 1 if it is checked, or 2 if the box is grayed.

This object (`val`) will never be disposed by WDS, even after the control or associated dialog are disposed. Therefore, this is one way of gaining access to a control's value after the life of the control. It is the programmer's responsibility to dispose of the object when it is no longer needed.

Example:

```
object  ctl, val;

val = gNewWithInt(ShortInteger, 0);
gAttach(ctl, val);
```

See also: `Value`, `ShortValue`, `SetValue`, `SetShortValue`

`NewCtl::CheckBox` [NewCtl]

```
ctl = mNewCtl(CheckBox, id);

unsigned  id;   /*  control id      */
object   ctl;   /*  control object  */
```

This method is used to create a new control object to be identified as `id` (see section "Standard Control Method Arguments"). This method is mainly used internally. A programmer would more often use `AddControl::Dialog` to create controls and associated them to a dialog.

Example:

```
object  ctl;

ctl = mNewCtl(CheckBox, MY_CHECKBOX);
```

See also: `AddControl::Dialog`, `GetControl::Dialog`

`SetShortValue::CheckBox` [SetShortValue]

```
r = gSetShortValue(ctl, val);

object  ctl;    /*  control object  */
int     val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be the value which represents the desired state associated with the check box.

`val` should be set to 0 to uncheck the box, 1 to check it, or 2 to gray the box (only work on 3-state boxes).

Any previously associated value object (associated via `SetValue`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetShortValue(ctl, 1);
```

See also: `SetValue`, `ShortValue`, `CtlShortValue::Dialog`

`SetValue::CheckBox` [SetValue]

```
r = gSetValue(ctl, val);

object  ctl;    /*  control object  */
object  val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be a Dynace object which is an instance of the `ShortInteger` class and initialized to the value desired for the control.

`val` should represent 0 to uncheck the box, 1 to check it, or 2 to gray the box (only work on 3-state boxes).

Any previously associated object will be disposed when the new value is set. Also, `val` will automatically be disposed when the control or associated dialog is disposed.

Example:

```
object  ctl;

gSetValue(ctl, gNewWithInt(ShortInteger, 1));
```

See also: `SetShortValue`, `Value`

`ShortValue::CheckBox` [ShortValue]

```
val = gShortValue(ctl);

object  ctl;   /*  control object  */
short   val;   /*  ctl value       */
```

This method is used to obtain a C short integer which represents the state associated with the control. The returned integer will be 0 if the box is not checked, 1 if it is checked, or 2 if the box is grayed.

Example:

```
object  ctl;
short   val;

val = gShortValue(ctl);
```

See also: `Value`, `Attach`, `SetShortValue`, `CtlShortValue::Dialog`

`Value::CheckBox` [Value]

```
val = gValue(ctl);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
```

This method is used to obtain a Dynace object which represents the value associated with the control. The object returned will be an instance of the `ShortInteger` class. The returned object will represent 0 if the box is not checked, 1 if it is checked, or 2 if the box is grayed. This object will be disposed by WDS when the control object or associated dialog is disposed.

Example:

```
object  ctl, val;

val = gValue(ctl);
```

See also: `ShortValue, Attach, SetValue`

4.9.8 Radio Buttons

The `RadioButton` class represents a control which allows the user to select or de-select a particular option. Unlike check boxes, however, radio buttons are mutually exclusive within a group - similar to the buttons on old fashioned radios, when one button gets selected all the other ones in the group get de-selected.

When using radio buttons, be sure to use the `Group` function in order to tell WDS which radio buttons should be grouped together.

The `RadioButton` class is a subclass of `Control` and as such inherits all the functionality associated with that class. This section only documents functionality particular to this class.

Standard control arguments are documented in the section entitled "Standard Control Method Arguments".

`Attach::RadioButton` [Attach]

```
r = gAttach(ctl, val);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
object  r;     /*  control object  */
```

This method is used to associate an independent Dynace object with the state associated with a control object. `val` should be an instance of the `ShortInteger` class. This object will be automatically updated to reflect the state associated with the control.

The object will represent 0 if the button is not selected, 1 if it is selected, or 2 if the button is grayed.

This object (`val`) will never be disposed by WDS, even after the control or associated dialog are disposed. Therefore, this is one way of gaining access to a control's value after the life of the control. It is the programmer's responsibility to dispose of the object when it is no longer needed.

Example:

```
object  ctl, val;

val = gNewWithInt(ShortInteger, 0);
gAttach(ctl, val);
```

See also: `Value, ShortValue, SetValue, SetShortValue`

Group::RadioButton [Group]

```
r = mGroup(ctl, id1, id2);

object  ctl;    /*  control object   */
unsigned id1;   /*  starting ctl id  */
unsigned id2;   /*  ending ctl id    */
object  r;      /*  control object   */
```

This method is used to tell WDS which radio buttons are to be considered part of the same mutually exclusive group. `id1` and `id2` are the resource editor defined id's associated with the beginning and end of the group. WDS uses this information to automatically de-select all radio buttons which are part of a group when one of the buttons is selected.

Example:

```
object  ctl;

mGroup(ctl, FIRST_RB, LAST_RB);
```

NewCtl::RadioButton [NewCtl]

```
ctl = mNewCtl(RadioButton, id);

unsigned  id;   /*  control id      */
object   ctl;   /*  control object  */
```

This method is used to create a new control object to be identified as `id` (see section "Standard Control Method Arguments"). This method is mainly used internally. A programmer would more often use `AddControl::Dialog` to create controls and associated them to a dialog.

Example:

```
object  ctl;

ctl = mNewCtl(RadioButton, MY_RADIOBUTTON);
```

See also: `AddControl::Dialog`, `GetControl::Dialog`, `Group`

`SetShortValue::RadioButton` [SetShortValue]

```
r = gSetShortValue(ctl, val);

object  ctl;    /*  control object  */
int     val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be the value which represents the desired state associated with the radio button.

`val` should be set to 0 to de-select the button, 1 to select it, or 2 to gray the button (only work on 3-state buttons).

Any previously associated value object (associated via `SetValue`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetShortValue(ctl, 1);
```

See also: `SetValue`, `ShortValue`, `CtlShortValue::Dialog`, `Group`

`SetValue::RadioButton` [SetValue]

```
r = gSetValue(ctl, val);

object  ctl;    /*  control object  */
object  val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be a Dynace object which is an instance of the `ShortInteger` class and initialized to the value desired for the control.

`val` should represent 0 to de-select the button, 1 to select it, or 2 to gray the button (only work on 3-state buttons).

Any previously associated object will be disposed when the new value is set. Also, `val` will automatically be disposed when the control or associated dialog is disposed.

Example:

```
object  ctl;

gSetValue(ctl, gNewWithInt(ShortInteger, 1));
```

See also: `SetShortValue`, `Value`, `Group`

`ShortValue::RadioButton` [ShortValue]

```
val = gShortValue(ctl);

object  ctl;   /*  control object  */
short   val;   /*  ctl value       */
```

This method is used to obtain a C short integer which represents the state associated with the control. The returned integer will be 0 if the button is not selected, 1 if it is selected, or 2 if the button is grayed.

Example:

```
object  ctl;
short   val;

val = gShortValue(ctl);
```

See also: `Value`, `Attach`, `SetShortValue`, `CtlShortValue::Dialog`

`Value::RadioButton` [Value]

```
val = gValue(ctl);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
```

This method is used to obtain a Dynace object which represents the value associated with the control. The object returned will be an instance of the `ShortInteger` class. The returned object will represent 0 if the button is not selected, 1 if it is selected, or 2 if the button is grayed. This object will be disposed by WDS when the control object or associated dialog is disposed.

Example:

```
object  ctl, val;

val = gValue(ctl);
```

See also: `ShortValue, Attach, SetValue`

4.9.9 List Boxes

The `ListBox` class represents a control which allows the user to select from a list of choices. These choices are presented in a vertical, possibly drop down, list of text choices. This control supports access to the user's choice via the text option selected or an ordinal value.

The application specifies, at runtime, the choices located in the list box.

The main difference between list boxes and combo boxes is that with list boxes the user can only select from the available choices, however, in addition, combo boxes allow the user to enter a choice independent of the available options.

The `ListBox` class is a subclass of `Control` and as such inherits all the functionality associated with that class. This section only documents functionality particular to this class.

Standard control arguments are documented in the section entitled "Standard Control Method Arguments".

`AddOption::ListBox` [AddOption]

```
r = gAddOption(ctl, op);

object  ctl;    /*  control object  */
char    *op;    /*  new option      */
object  r;      /*  control object  */
```

This method is used to append a new option to the list of options associated with a list box. This method may be used prior to or during the execution (via `gPerform`) of a dialog.

`op` may also be a Dynace object, an instance of the `String` class, representing the new choice. If a Dynace object is used, it must be typecast to a string and it will be disposed when the control or associated dialog are disposed.

Example:

```
object  ctl;

gAddOption(ctl, "The first choice");
gAddOption(ctl, "The second choice");
```

See also: `Alphabetize::ListBox, AddOptionAt::ListBox`

`Alphabetize::ListBox` [Alphabetize]

```
r = gAlphabetize(ctl);

object  ctl;    /*  control object  */
object  r;      /*  control object  */
```

This method is used to cause the automatic alphabetization of all options added via the `AddOption` method. In order to function properly, `Alphabetize` must be called prior to any calls to `AddOption`. This method simply returns the object passed.

See also: `AddOption::ListBox`

`Attach::ListBox` [Attach]

```
r = gAttach(ctl, val);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
object  r;     /*  control object  */
```

This method is used to associate an independent Dynace object with the state associated with a control object. `val` should be an instance of the `ShortInteger` class or the `String` class. This object will be automatically updated to reflect the state associated with the control.

If `val` is an instance of the `ShortIntgeger` class it will be used to represent the ordinal value of the selected item. This value is a 0 based index from the top of the list to the bottom. -1 is used to represent no valid selection.

If `val` is an instance of the `String` class it will be used to represent the string represented by the user's choice. If no choice is made it will represent `""`.

This object (`val`) will never be disposed by WDS, even after the control or associated dialog are disposed. Therefore, this is one way of gaining access to a control's value after the life of the control. It is the programmer's responsibility to dispose of the object when it is no longer needed.

Example:

```
object  ctl, val;

val = gNewWithInt(ShortInteger, 0);
gAttach(ctl, val);
```

See also: `Value, ShortValue, StringValue`

`GetSelections::ListBox` [GetSelections]

```
ary = gGetSelections(ctl);

object  ctl;   /*  control object       */
object  ary;   /*  array of selections  */
```

This method is used to obtain an array representing the items selected by the user. It is most useful with list boxes which have multi-selection enabled. The array returned is an instance of the `IntegerArray` class and *must* be explicitly disposed when no longer needed. Each element of the array represents a zero based index of one of the user's selections, and the size of the array represents the number of items selected.

This method may only be used while a dialog is active and will return `NULL` otherwise.

See also: `ValueAt, NumbSelected, Value, ListIndex`

`ListIndex::ListBox` [ListIndex]

```
val = gListIndex(ctl);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
```

This method is used to obtain a Dynace object which represents the value associated with the control. The object returned will be an instance of the `ShortInteger` class. The returned object will represent the ordinal value of the choice made by the user. This ordinal value is a zero based index from top to bottom. -1 indicates that no choice was made.

This object will be disposed by WDS when the control object or associated dialog is disposed.

Example:

```
object  ctl, val;

val = gListIndex(ctl);
```

See also: `ShortValue, Value, StringValue, GetSelections, SetValue`

`NewCtl::ListBox` [NewCtl]

```
ctl = mNewCtl(ListBox, id);

unsigned  id;   /*  control id      */
object   ctl;   /*  control object  */
```

This method is used to create a new control object to be identified as `id` (see section "Standard Control Method Arguments"). This method is mainly used internally. A programmer would more often use `AddControl::Dialog` to create controls and associated them to a dialog.

Example:

```
object  ctl;

ctl = mNewCtl(ListBox, MY_LISTBOX);
```

See also: `AddControl::Dialog`, `GetControl::Dialog`, `AddOption`

`NumbSelected::ListBox` [NumbSelected]

```
n = gNumbSelected(ctl);

object  ctl;    /*  control object   */
int     n;      /*  number selected  */
```

This method is used to obtain the number of list box items the user has selected. It is mainly used with multi-selection list boxes. This method should only be used while the dialog is active and will return -1 otherwise.

See also: `GetSelections`

`RemoveAll::ListBox` [RemoveAll]

```
r = gRemoveAll(ctl);

object  ctl;    /*  control object  */
object  r;      /*  control object  */
```

This method is used to remove all items from a list box. It may be used prior to or during the execution of a dialog. The control object passed is returned.

See also: `RemoveStr`, `RemoveInt`

RemoveInt::ListBox [RemoveInt]

```
r = gRemoveInt(ctl, idx);

object  ctl;    /*  control object  */
int     idx;    /*  index           */
object  r;      /*  control object  */
```

This method is used to remove an item from a list box while the dialog is active. `idx` is a zero based index of the item to be removed. If the operation secceeded `ctl` is returned, otherwise `NULL` is returned.

This method may only be used while a dialog is active. It should not be used prior to performing (via `gPerform`) a dialog or after the user has accepted or canceled the dialog.

Example:

```
object  ctl;

gRemoveInt(ctl, 1);
```

See also: `RemoveStr`, `RemoveAll`

RemoveStr::ListBox [RemoveStr]

```
r = gRemoveStr(ctl, itm);

object  ctl;    /*  control object  */
char   *itm;    /*  item            */
object  r;      /*  control object  */
```

This method is used to remove an item from a list box while the dialog is active. `itm` is the string to be removed. If the operation secceeded `ctl` is returned, otherwise `NULL` is returned.

This method may only be used while a dialog is active. It should not be used prior to performing (`gPerform`) a dialog or after the user has accepted or canceled the dialog.

Example:

```
object  ctl;

gRemoveStr(ctl, "Some Option");
```

See also: `RemoveInt`, `FindMode`

Required::ListBox [Required]

```
r = gRequired(ctl, mode);

object  ctl;    /*  control object  */
int     mode;   /*  required mode   */
object  r;      /*  control object  */
```

This method is used to determine whether or not the user is required to make a valid selection prior to WDS allowing the acceptance of the dialog. If it is required and the user does not make a valid selection, WDS will issue an error message and return them to the control.

`mode` is 1 to make it required and 0 otherwise.

Example:

```
object  ctl;

gRequired(ctl, 1);
```

See also: `CheckFunction::Control, CheckValue::Control`

SetFunction::ListBox [SetFunction]

```
r = gSetFunction(ctl, fun);

object  ctl;      /*  control object          */
int     (*fun)(); /*  check function          */
ofun    r;        /*  previous check function  */
```

This method is used to associate a C function to a list box such that if the user double clicks an item in the list box the C function will immediately get evoked. The value returned by this method is the function, if any, which was previously associated with the control.

The C function takes the following form:

```
int     fun(object ctl, object dlg)
{
        ....
}
```

Where `ctl` is the control object, and `dlg` is the object representing the dialog the control is associated with. The value returned by `fun` is ignored.

Example:

```
object  ctl;

gSetFunction(ctl, fun);
```

See also: `Perform`, `SetChgFunction`

`SetShortValue::ListBox` [SetShortValue]

```
r = gSetShortValue(ctl, val);

object  ctl;    /*  control object  */
int     val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the default selection associated with the control. It is often used to set the initial value prior to performing a dialog.

`val` will be used as the ordinal value of the selected item. This value is a 0 based index from the top of the list to the bottom.

Any previously associated object (via `Value`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetShortValue(ctl, 1);
```

See also: `SetStringValue`, `SetValue`

`SetStringValue::ListBox` [SetStringValue]

```
r = gSetStringValue(ctl, val);

object  ctl;    /*  control object  */
char    *val;   /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the default selection associated with the control. It is often used to set the initial value prior to performing a dialog.

`val` will be used to select the option which matches the string.

Any previously associated object (via `val`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetStringValue(ctl, "The choice");
```

See also: `SetShortValue`, `SetValue`

`SetValue::ListBox` [SetValue]

```
r = gSetValue(ctl, val);

object  ctl;    /*  control object  */
object  val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the default selection associated with the control. It is often used to set the initial value prior to performing a dialog. `val` should be a Dynace object which is an instance of either the `String` class or the `ShortInteger` class and initialized to the value desired for the control.

If `val` is an instance of the `ShortIntgeger` class it will be used as the ordinal value of the selected item. This value is a 0 based index from the top of the list to the bottom.

If `val` is an instance of the `String` class it will be used to select the option which matches the string represented by `val`.

Any previously associated object will be disposed when the new value is set. Also, `val` will automatically be disposed when the control or associated dialog is disposed.

Example:

```
object  ctl;

gSetValue(ctl, gNewWithInt(ShortInteger, 1));
        or
gSetValue(ctl, gNewWithStr(String, "The choice"));
```

See also: `SetShortValue`, `SetStringValue`

ShortValue::ListBox [ShortValue]

```
val = gShortValue(ctl);

object  ctl;   /*  control object  */
short   val;   /*  ctl value       */
```

This method is used to obtain a C short which represents the value associated with the control. The returned value will represent the ordinal value of the choice made by the user. This ordinal value is a zero based index from top to bottom. -1 indicates that no choice was made.

Example:

```
object  ctl;
short   val;

val = gShortValue(ctl);
```

See also: `ListIndex`, `Value`, `GetSelections`, `StringValue`, `SetValue`

StringValue::ListBox [StringValue]

```
val = gStringValue(ctl);

object  ctl;   /*  control object  */
char    *val;  /*  ctl value       */
```

This method is used to obtain a character string pointer which represents the value associated with the control. The returned pointer will represent the text associated with the user selected choices. It will point to `""` if the user had not made a valid selection.

The returned pointer will not be valid once the control object or associated dialog is disposed.

Example:

```
object  ctl;
char    *val;

val = gStringValue(ctl);
```

See also: `ShortValue`, `Value`, `ListIndex`, `GetSelections`, `SetValue`

`Value::ListBox` [Value]

```
val = gValue(ctl);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
```

This method is used to obtain a Dynace object which represents the value associated with the control. The object returned will be an instance of the `String` class. The returned object will represent the text associated with the user selected choices. If no selection was made the object will represent `""`.

This object will be disposed by WDS when the control object or associated dialog is disposed.

Example:

```
object  ctl, val;

val = gValue(ctl);
```

See also: `ShortValue`, `StringValue`, `ValueAt`, `ListIndex`, `GetSelections`, `SetValue`

`ValueAt::ListBox` [ValueAt]

```
val = gValueAt(ctl, idx);

object  ctl;   /*  control object  */
int     idx;   /*  index           */
object  val;   /*  value at idx    */
```

This method is used to obtain a Dynace object which represents the text associated with the line indexed by the zero based index `idx`. The object returned will be an instance of the `String` class. If the index is out of range this method will return `NULL`.

This method may only be called when the dialog is active. The object returned *must* be explicitly disposed when no longer needed.

See also: `ShortValue`, `Value`, `ListIndex`, `GetSelections`, `SetValue`

4.9.10 Combo Boxes

The `ComboBox` class represents a control which allows the user to select from a list of choices or enter an alternate text string. These choices are presented in a vertical, possibly drop down, list of text choices. This control supports access to the user's choice via the text option selected or an ordinal value.

The application specifies, at runtime, the choices located in the list box.

The main difference between list boxes and combo boxes is that with list boxes the user can only select from the available choices, however, in addition, combo boxes allow the user to enter a choice independent of the available options.

The `ComboBox` class is a subclass of `Control` and as such inherits all the functionality associated with that class. This section only documents functionality particular to this class.

Standard control arguments are documented in the section entitled "Standard Control Method Arguments".

`AddEdtHandlerAfter::ComboBox` [AddEdtHandlerAfter]

```
r = gAddEdtHandlerAfter(ctl, msg, func);

object   ctl;      /*  a control object  */
unsigned msg;      /*  message           */
long    (*func)(); /*  function pointer  */
object  r;         /*  the control obj   */
```

This method is used to associate function `func` with the text entry window portion of the combo box, message `msg` for control `ctl`. Whenever the text entry window of control `ctl` receives message `msg`, `func` will be called.

`ctl` is the control object who's text window messages you wish to process. `msg` is the particular message you wish to trap. These messages are fully documented in the Windows documentation in the Messages section. They normally begin with `WM_`.

`func` is the function which gets called whenever the specified message gets received and takes the following form:

```
long    func(object     ctl,
             HWND       hwnd,
             UINT       mMsg,
             WPARAM     wParam,
             LPARAM     lParam)
{
        .
        .
        .
        return 0L;  /* or whatever is appropriate  */
}
```

Where `ctl` is the control being sent the message. The remaining arguments and return value is fully documented in the Windows documentation under the `WindowProc` function and the Windows Messages documentation.

WDS keeps a list of functions associated with each message associated with each control. When a particular message is received the appropriate list of handler functions gets executed sequentially. `AddEdtHandlerAfter` appends the new function to the end of this list, and `AddEdtHandlerBefore` adds the new function to the beginning of the list.

WDS may also, and optionally, execute the Windows default procedure associated with a given message either before or after the user added list of functions. This behavior may be controlled via `DefaultEdtProcessingMode`.

Windows will only see the return value of the last message handler executed including, if applicable, the default.

Example:

```
int     hSize, vSize;

static  long    process_wm_size(object  ctl,
                                HWND    hwnd,
                                UINT    mMsg,
                                WPARAM  wParam,
                                LPARAM  lParam)
{
        hSize = LOWORD(lParam);
        vSize = HIWORD(lParam);
        return 0L;
}

        .
        .
        gAddEdtHandlerAfter(ctl, (unsigned) WM_SIZE,
                                        process_wm_size);
        .
        .
```

See also: `DefaultProcessingMode`, `AddEdtHandlerBefore`, `AddHandlerAfter::Control`

`AddEdtHandlerBefore::ComboBox` [AddEdtHandlerBefore]

```
r = gAddEdtHandlerBefore(ctl, msg, func);

object   ctl;      /*  a control object  */
unsigned msg;      /*  message           */
long    (*func)(); /*  function pointer  */
object  r;         /*  the control obj   */
```

This function is fully documented under `AddEdtHandlerAfter`.

See also: `AddEdtHandlerAfter, AddHandlerAfter::Control`

`AddOption::ComboBox` [AddOption]

```
r = gAddOption(ctl, op);

object  ctl;    /*  control object  */
char    *op;    /*  new option      */
object  r;      /*  control object  */
```

This method is used to append a new option to the list of options associated with a combo box. This method may be used prior to or during the execution (via `gPerform`) of a dialog.

`op` may also be a Dynace object, an instance of the `String` class, representing the new choice. If a Dynace object is used, it must be typecast to a string and it will be disposed when the control or associated dialog are disposed.

Example:

```
object  ctl;

gAddOption(ctl, "The first choice");
gAddOption(ctl, "The second choice");
```

See also: `Alphabetize::ComboBox, AddOptionAt::ComboBox`

`Alphabetize::ComboBox` [Alphabetize]

```
r = gAlphabetize(ctl);

object  ctl;    /*  control object  */
object  r;      /*  control object  */
```

This method is used to cause the automatic alphabetization of all options added via the `AddOption` method. In order to function properly, `Alphabetize` must be called prior to any calls to `AddOption`. This method simply returns the object passed.

See also: `AddOption::ComboBox`

`Attach::ComboBox` [Attach]

```
r = gAttach(ctl, val);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
object  r;     /*  control object  */
```

This method is used to associate an independent Dynace object with the state associated with a control object. `val` should be an instance of the `ShortInteger` class or the `String` class. This object will be automatically updated to reflect the state associated with the control.

If `val` is an instance of the `ShortIntgeger` class it will be used to represent the ordinal value of the selected item. This value is a 0 based index from the top of the list to the bottom. -1 is used to represent no valid selection.

If `val` is an instance of the `String` class it will be used to represent the string represented by the user's choice. If no choice is made it will represent `""`.

This object (`val`) will never be disposed by WDS, even after the control or associated dialog are disposed. Therefore, this is one way of gaining access to a control's value after the life of the control. It is the programmer's responsibility to dispose of the object when it is no longer needed.

Example:

```
object  ctl, val;

val = gNewWithInt(ShortInteger, 0);
gAttach(ctl, val);
```

See also: `Value, ShortValue, StringValue`

`DefaultEdtProcessingMode::ComboBox` [DefaultEdtProcessingMode]

```
r = gDefaultEdtProcessingMode(ctl, msg, mode);

object   ctl;   /*  a control object         */
unsigned msg;   /*  message                  */
int      mode;  /*  default processing mode  */
object   r;     /*  the control obj          */
```

This method is used to determine when or if the Windows default message procedure is processed for a given message (`msg`) associated with the text entry window portion of control (`ctl`).

WDS allows a programmer to specify an arbitrary number of functions to be executed whenever the text entry portion of a combo box control receives a specific message (via `AddEdtHandlerAfter` and `AddEdtHandlerBefore`). Windows has default procedures associated with many control messages. At times it is necessary to replace or augment this default functionality. `DefaultEdtProcessingMode` gives the programmer control over when and if this default Windows functionality. `mode` is used to specify the desired mode. The following table indicates the valid modes:

`0`	Do not execute the Windows default processing
`1`	Execute default processing *after* programmer defined handlers
`2`	Execute default processing *before* programmer defined handlers

Note that the default mode is always `1`, and must be explicitly changed, if desired, for each message associated with each control.

`msg` is the particular message you wish to affect. These messages are fully documented in the Windows documentation in the Messages section. They normally begin with `WM_`.

Example:

```
gDefaultEdtProcessingMode(ctl, (unsigned) WM_SIZE, 0);
```

See also: `AddEdtHandlerAfter`

`ListIndex::ComboBox` [ListIndex]

```
val = gListIndex(ctl);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
```

This method is used to obtain a Dynace object which represents the value associated with the control. The object returned will be an instance of the `ShortInteger` class. The returned object will represent the ordinal value of the choice made by the user. This ordinal value is a zero based index from top to bottom. -1 indicates that no choice was made.

This object will be disposed by WDS when the control object or associated dialog is disposed.

Example:

```
object  ctl, val;

val = gListIndex(ctl);
```

See also: `ShortValue`, `Value`, `StringValue`, `Attach`, `SetValue`

`NewCtl::ComboBox` [NewCtl]

```
ctl = mNewCtl(ComboBox, id);

unsigned  id;   /*  control id      */
object   ctl;   /*  control object  */
```

This method is used to create a new control object to be identified as `id` (see section "Standard Control Method Arguments"). This method is mainly used internally. A programmer would more often use `AddControl::Dialog` to create controls and associated them to a dialog.

Example:

```
object  ctl;

ctl = mNewCtl(ComboBox, MY_COMBOBOX);
```

See also: `AddControl::Dialog`, `GetControl::Dialog`, `AddOption`

`RemoveAll::ComboBox` [RemoveAll]

```
r = gRemoveAll(ctl);

object  ctl;    /*  control object  */
object  r;      /*  control object  */
```

This method is used to remove all items from a combo box. It may be used prior to or during the execution of a dialog. The control object passed is returned.

See also: `RemoveStr`, `RemoveInt`

`RemoveInt::ComboBox` [RemoveInt]

```
r = gRemoveInt(ctl, idx);

object  ctl;    /*  control object  */
int     idx;    /*  index           */
object  r;      /*  control object  */
```

This method is used to remove an item from a combo box while the dialog is active. `idx` is a zero based index of the item to be removed. If the operation secceeded `ctl` is returned, otherwise `NULL` is returned.

This method may only be used while a dialog is active. It should not be used prior to performing (`gPerform`) a dialog or after the user has accepted or canceled the dialog.

Example:

```
object  ctl;

gRemoveInt(ctl, 1);
```

See also: `RemoveStr`, `RemoveAll`

`RemoveStr::ComboBox` [RemoveStr]

```
r = gRemoveStr(ctl, itm);

object  ctl;    /*  control object  */
char   *itm;    /*  item            */
object  r;      /*  control object  */
```

This method is used to remove an item from a combo box while the dialog is active. `itm` is the string to be removed. If the operation secceeded `ctl` is returned, otherwise `NULL` is returned.

This method may only be used while a dialog is active. It should not be used prior to performing (`gPerform`) a dialog or after the user has accepted or canceled the dialog.

Example:

```
object  ctl;

gRemoveStr(ctl, "Some Option");
```

See also: `RemoveInt::ComboBox`, `FindMode::ComboBox`

`Required::ComboBox` [Required]

```
r = gRequired(ctl, mode);

object  ctl;    /*  control object  */
int     mode;   /*  required mode   */
object  r;      /*  control object  */
```

This method is used to determine whether or not the user is required to make a valid selection prior to WDS allowing the acceptance of the dialog. If it is required and the user does not make a valid selection, WDS will issue an error message and return them to the control.

`mode` is 1 to make it required and 0 otherwise.

Example:

```
object  ctl;

gRequired(ctl, 1);
```

See also: `CheckFunction::Control`, `CheckValue::Control`

`SetFunction::ComboBox` [SetFunction]

```
r = gSetFunction(ctl, fun);

object  ctl;      /*  control object          */
int     (*fun)(); /*  check function          */
ofun    r;        /*  previous check function  */
```

This method is used to associate a C function to a combo box such that if the user double clicks an item in the combo box the C function will immediately get evoked. The value returned by this method is the function, if any, which was previously associated with the control.

The C function takes the following form:

```
int     fun(object ctl, object dlg)
{
        ....
}
```

Where `ctl` is the control object, and `dlg` is the object representing the dialog the control is associated with. The value returned by `fun` is ignored.

Example:

```
object  ctl;

gSetFunction(ctl, fun);
```

See also: `Perform`, `SetChgFunction`

`SetShortValue::ComboBox` [SetShortValue]

```
r = gSetShortValue(ctl, val);

object  ctl;    /*  control object  */
int     val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the default selection associated with the control. It is often used to set the initial value prior to performing a dialog.

`val` will be used as the ordinal value of the selected item. This value is a 0 based index from the top of the list to the bottom.

Any previously associated object (via `Value`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetShortValue(ctl, 1);
```

See also: `SetStringValue`, `SetValue`

`SetStringValue::ComboBox` [SetStringValue]

```
r = gSetStringValue(ctl, val);

object  ctl;    /*  control object  */
char    *val;   /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the default selection associated with the control. It is often used to set the initial value prior to performing a dialog.

`val` will be used to select the option which matches the string.

Any previously associated object (via `val`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetStringValue(ctl, "The choice");
```

See also: `SetShortValue`, `SetValue`

SetValue::ComboBox [SetValue]

```
r = gSetValue(ctl, val);

object  ctl;    /*  control object  */
object  val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the default selection associated with the control. It is often used to set the initial value prior to performing a dialog. `val` should be a Dynace object which is an instance of either the `String` class or the `ShortInteger` class and initialized to the value desired for the control.

If `val` is an instance of the `ShortIntgeger` class it will be used as the ordinal value of the selected item. This value is a 0 based index from the top of the list to the bottom.

If `val` is an instance of the `String` class it will be used to select the option which matches the string represented by `val`.

Any previously associated object will be disposed when the new value is set. Also, `val` will automatically be disposed when the control or associated dialog is disposed.

Example:

```
object  ctl;

gSetValue(ctl, gNewWithInt(ShortInteger, 1));
        or
gSetValue(ctl, gNewWithStr(String, "The choice"));
```

See also: `SetShortValue`, `SetStringValue`

ShortValue::ComboBox [ShortValue]

```
val = gShortValue(ctl);

object  ctl;   /*  control object  */
short   val;   /*  ctl value       */
```

This method is used to obtain a C short which represents the value associated with the control. The returned value will represent the ordinal value of the choice made by the user. This ordinal value is a zero based index from top to bottom. -1 indicates that no choice was made.

Example:

```
object  ctl;
short   val;

val = gShortValue(ctl);
```

See also: `ListIndex`, `Value`, `StringValue`, `Attach`, `SetValue`

`StringValue::ComboBox` [StringValue]

```
val = gStringValue(ctl);

object  ctl;   /*  control object  */
char    *val;  /*  ctl value       */
```

This method is used to obtain a character string pointer which represents the value associated with the control. The returned pointer will represent the text associated with the user selected choices. It will point to `""` if the user had not made a valid selection.

The returned pointer will not be valid once the control object or associated dialog is disposed.

Example:

```
object  ctl;
char    *val;

val = gStringValue(ctl);
```

See also: `ShortValue`, `Value`, `ListIndex`, `Attach`, `SetValue`

`Value::ComboBox` [Value]

```
val = gValue(ctl);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
```

This method is used to obtain a Dynace object which represents the value associated with the control. The object returned will be an instance of the `String` class. The returned object will represent the text associated with the user selected choices. If no selection was made the object will represent `""`.

This object will be disposed by WDS when the control object or associated dialog is disposed.

Example:

```
object  ctl, val;

val = gValue(ctl);
```

See also: `ShortValue`, `StringValue`, `ValueAt`, `ListIndex`, `Attach`, `SetValue`

`ValueAt::ComboBox` [ValueAt]

```
val = gValueAt(ctl, idx);

object  ctl;   /*  control object  */
int     idx;   /*  index           */
object  val;   /*  value at idx    */
```

This method is used to obtain a Dynace object which represents the text associated with the line indexed by the zero based index `idx`. The object returned will be an instance of the `String` class. If the index is out of range this method will return `NULL`.

This method may only be called when the dialog is active. The object returned *must* be explicitly disposed when no longer needed.

See also: `ShortValue`, `Value`, `ListIndex`, `SetValue`

4.9.11 Scroll Bars

The `ScrollBar` class represents a control which allows the user to select a number between two ranges via a linear, visually intuitive control. The range defaults to 0 to 100 and may be controlled via `ScrollBarRange`.

The `ScrollBar` class is a subclass of `Control` and as such inherits all the functionality associated with that class. This section only documents functionality particular to this class.

Standard control arguments are documented in the section entitled "Standard Control Method Arguments".

`Attach::ScrollBar` [Attach]

```
r = gAttach(ctl, val);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
object  r;     /*  control object  */
```

This method is used to associate an independent Dynace object with the state associated with a control object. `val` should be an instance of the `ShortInteger` class. This object will be automatically updated to reflect the position of the scroll bar.

This object (`val`) will never be disposed by WDS, even after the control or associated dialog are disposed. Therefore, this is one way of gaining access to a control's value after the life of the control. It is the programmer's responsibility to dispose of the object when it is no longer needed.

Example:

```
object  ctl, val;

val = gNewWithInt(ShortInteger, 0);
gAttach(ctl, val);
```

See also: `Value, ShortValue, SetValue, SetShortValue, ScrollBarRange`

`Increment::ScrollBar` [Increment]

```
r = gIncrement(ctl, val);

object  ctl;   /*  control object  */
int     val;   /*  increment value */
object  r;     /*  control object  */
```

This method is used to increment the position of the scroll bar `val` points from its current position. `val` may be negative or positive, which will determine the direction of movement.

Example:

```
object  ctl;

gIncrement(ctl, 10);
```

See also: `SetShortValue, SetValue, ScrollBarRange`

`NewCtl::ScrollBar` [NewCtl]

```
ctl = mNewCtl(ScrollBar, id);

unsigned  id;   /*  control id      */
object   ctl;   /*  control object  */
```

This method is used to create a new control object to be identified as `id` (see section "Standard Control Method Arguments"). This method is mainly used internally. A programmer would more often use `AddControl::Dialog` to create controls and associated them to a dialog.

Example:

```
object  ctl;

ctl = mNewCtl(ScrollBar, MY_SCROLLBAR);
```

See also: `AddControl::Dialog`, `GetControl::Dialog`

`ScrollBarRange::ScrollBar` [ScrollBarRange]

```
r = gScrollBarRange(ctl, min, max, pginc, lininc)

object  ctl;    /*  control object  */
int     min;    /*  minimum range   */
int     max;    /*  maximum range   */
int     pginc;  /*  page increment  */
int     lineinc;/*  line increment  */
object  r;      /*  control object  */
```

This method is used to control the range and scrolling characteristics associated with a given scroll bar control.

`min` and `max` are used to determine the range of the control. These are the values which are returned when the control is at either end of its extremes. The default values are 0 and 100 respectively.

`pginc` is used to control exactly how much the position of the scroll bar will change when the user clicks the control in such a way as to cause a page jump in the position. This is normally done by clicking in the area on either side of position indicating element of the control. The default value for the variable is 10. It must be less than the difference between `max` and `min`, and is normally larger than `lineinc`.

`lineinc` us used to control exactly how much the position of the scroll bar will change when the user clicks the line increment arrows located on the extreme ends of the control. The default value of this control is 2.

Example:

```
object  ctl;

gScrollBarRange(ctl, 0, 100, 10, 2);
```

`SetShortValue::ScrollBar` [SetShortValue]

```
r = gSetShortValue(ctl, val);

object  ctl;    /*  control object  */
int     val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be the value which represents the desired position of the scroll bar. It must be between the minimum and maximum range associated with the control.

Any previously associated value object (associated via `SetValue`) will be disposed when the new value is set.

Example:

```
object  ctl;

gSetShortValue(ctl, 50);
```

See also: `SetValue`, `Increment`, `CtlShortValue::Dialog`, `ScrollBarRange`

`SetValue::ScrollBar` [SetValue]

```
r = gSetValue(ctl, val);

object  ctl;    /*  control object  */
object  val;    /*  ctl value       */
object  r;      /*  control object  */
```

This method is used to set the value associated with a control. It is often used to set the initial value prior to performing a dialog. `val` should be a Dynace object which is an instance of the `ShortInteger` class and initialized to the value desired for the control.

`val` must represent a number between the minimum and maximum range associated with the scroll bar.

Any previously associated object will be disposed when the new value is set. Also, `val` will automatically be disposed when the control or associated dialog is disposed.

Example:

```
object  ctl;

gSetValue(ctl, gNewWithInt(ShortInteger, 50));
```

See also: `SetShortValue, Increment, Value, ScrollBarRange`

`ShortValue::ScrollBar` [ShortValue]

```
val = gShortValue(ctl);

object  ctl;   /*  control object  */
short   val;   /*  ctl value       */
```

This method is used to obtain a C short integer which represents the position of the scroll bar. The returned value will represent the position of the scroll bar control. This number will be between the minimum and maximum range associated with the control via `ScrollBarRange`.

Example:

```
object  ctl;
short   val;

val = gShortValue(ctl);
```

See also: `Value, Attach, SetShortValue, CtlShortValue::Dialog`

`Value::ScrollBar` [Value]

```
val = gValue(ctl);

object  ctl;   /*  control object  */
object  val;   /*  ctl value       */
```

This method is used to obtain a Dynace object which represents the value associated with the control. The object returned will be an instance of the `ShortInteger` class. The returned object will represent the position of the scroll bar control. This number will be between the minimum and maximum range associated with the control via `ScrollBarRange`.

This object will be disposed by WDS when the control object or associated dialog is disposed.

Example:

```
object  ctl, val;

val = gValue(ctl);
```

See also: `ShortValue, Attach, SetValue`

4.9.12 Custom Controls

4.10 Cursors

The `Cursor` class (as well as its subclasses) are used to represent cursor objects. This class is never used by an application. It is used to house the functionality common to its subclasses. Therefore, this class documents functionality which is common to all of its subclasses.

Note that a more convenient mechanism for using cursors is provided by `LoadCursor::Window` and `LoadSystemCursor::Window`

`Copy::Cursor` [Copy]

```
c = gCopy(csr);

object  csr;    /*  cursor object          */
object  c;      /*  copy of cursor object  */
```

This method is used to create a new cursor object which is a copy of a given cursor object. The value of this is to be able to associate a cursor with multiple objects which will automatically dispose of the cursor when they are disposed. If copies weren't used the second object referencing the cursor would reference a disposed cursor object.

Each cursor object must be disposed (either automatically or manually) when it is no longer needed.

Example:

```
object  c1, c2;

c2 = gCopy(c1);
```

See also: `DeepCopy`

`DeepCopy::Cursor` [DeepCopy]

This method performs the same function as `Copy`. See that method for details.

`DeepDispose::Cursor` [DeepDispose]

This method performs the same function as `Dispose`. See that method for details.

`Dispose::Cursor` [Dispose]

```
r = gDispose(csr);

object  csr;   /*  a cursor object    */
object  r;     /*  NULL               */
```

This method is used to dispose of a cursor object when it is no longer needed. This method is rarely needed due to the fact that when a window is disposed it automatically calls this method to dispose of its associated cursor object.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  csr;

csr = gDispose(csr);
```

See also: `DeepDispose`

`Handle::Cursor` [Handle]

```
h = gHandle(csr);

object  csr;    /*  cursor object   */
HANDLE  h;      /*  Windows handle  */
```

This method is used to obtain the Windows internal handle associated with a cursor object.

Note that this method may be used with most WDS objects in order to obtain the internal handle that Windows normally associates with each type of object. See the appropriate documentation.

Example:

```
object  csr;
HCURSOR h;

h = gHandle(csr);
```

4.10.1 System Cursors

The `SystemCursor` class represents Windows predefined cursors. This class is a subclass of `Cursor` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Cursor` class for additional functionality.

`LoadSys::SystemCursor` [LoadSys]

```
csr = gLoadSys(SystemCursor, id);

char    *id;    /*  cursor id      */
object  csr;    /*  cursor object  */
```

This method is used to create a new object which represents one of the Windows predefined cursors. `id` should be one of the Windows defined macros specifying the desired cursor. It is defined in the Windows documentation under the function `LoadCursor` and starts with `IDC_`.

Example:

```
object  csr;

csr = gLoadSys(SystemCursor, IDC_IBEAM);
```

See also: `LoadSystemCursor::Window`, `SetCursor::Application`

4.10.2 External Cursors

The `ExternalCursor` class represents arbitrary application defined cursors. This class is a subclass of `Cursor` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Cursor` class for additional functionality.

`Load::ExternalCursor` [Load]

```
csr = mLoad(ExternalCursor, id);

unsigned id;    /*  cursor id      */
object   csr;   /*  cursor object  */
```

This method is used to load a programmer defined application specific cursor.

`id` is a programmer defined unsigned integer which identifies the cursor. This identifier is normally a macro and defined through the resource editor.

The value returned is an object representing the cursor loaded, or `NULL` if the cursor was not found.

Example:

```
object  csr;

csr = mLoad(ExternalCursor, MY_CURSOR);
```

See also: `LoadCursor::Window`, `SetCursor::Application`

`LoadStr::ExternalCursor` [LoadStr]

```
csr = mLoadStr(ExternalCursor, id);

char    *id;    /*  cursor id      */
object   csr;   /*  cursor object  */
```

This method is used to load a programmer defined application specific cursor by name.

`id` is a programmer defined name which identifies the cursor. This identifier is normally defined through the resource editor.

The value returned is an object representing the cursor loaded, or `NULL` if the cursor was not found.

Example:

```
object  csr;

csr = mLoadStr(ExternalCursor, "mycursor");
```

See also: `LoadCursor::Window`, `SetCursor::Application`, `Use::Window`

4.11 Icons

The `Icon` class (as well as its subclasses) are used to represent icon objects. This class is never used by an application. It is used to house the functionality common to its subclasses. Therefore, this class documents functionality which is common to all of its subclasses.

Note that a more convenient mechanism for using icons is provided by `LoadIcon::Window` and `LoadSystemIcon::Window`

`Copy::Icon` [Copy]

```
c = gCopy(icn);

object  icn;    /*  icon object          */
object  c;      /*  copy of icon object  */
```

This method is used to create a new icon object which is a copy of a given icon object. The value of this is to be able to associate a icon with multiple objects which will automatically dispose of the icon when they are disposed. If copies weren't used the second object referencing the icon would reference a disposed icon object.

Each icon object must be disposed (either automatically or manually) when it is no longer needed.

Example:

```
object  i1, i2;

i2 = gCopy(i1);
```

See also: `DeepCopy`

`DeepCopy::Icon` [DeepCopy]

This method performs the same function as `Copy`. See that method for details.

`DeepDispose::Icon` [DeepDispose]

This method performs the same function as `Dispose`. See that method for details.

`Dispose::Icon` [Dispose]

```
r = gDispose(icn);

object  icn;   /*  an icon object  */
object  r;     /*  NULL            */
```

This method is used to dispose of an icon object when it is no longer needed. This method is rarely needed due to the fact that when a window is disposed it automatically calls this method to dispose of its associated icon object.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  icn;

icn = gDispose(icn);
```

See also: `DeepDispose`

`Handle::Icon` [Handle]

```
h = gHandle(icn);

object  icn;    /*  icon object     */
HANDLE  h;      /*  Windows handle  */
```

This method is used to obtain the Windows internal handle associated with an icon object.

Note that this method may be used with most WDS objects in order to obtain the internal handle that Windows normally associates with each type of object. See the appropriate documentation.

Example:

```
object  icn;
HICON   h;

h = gHandle(icn);
```

4.11.1 System Icons

The `SystemIcon` class represents Windows predefined icons. This class is a subclass of `Icon` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Icon` class for additional functionality.

`LoadSys::SystemIcon` [LoadSys]

```
icn = gLoadSys(SystemIcon, id);

char    *id;    /*  icon id      */
object  icn;    /*  icon object  */
```

This method is used to create a new object which represents one of the Windows predefined icons. `id` should be one of the Windows defined macros specifying the desired icon. It is defined in the Windows documentation under the function `LoadIcon` and starts with `IDI_`.

Example:

```
object  icn;

icn = gLoadSys(SystemIcon, IDI_HAND);
```

See also: `LoadSystemIcon::Window`, `SetIcon::Application`

4.11.2 External Icons

The `ExternalIcon` class represents arbitrary application defined icons. This class is a subclass of `Icon` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Icon` class for additional functionality.

`Load::ExternalIcon` [Load]

```
icn = mLoad(ExternalIcon, id);

unsigned id;    /*  icon id      */
object   icn;   /*  icon object  */
```

This method is used to load a programmer defined application specific icon.

`id` is a programmer defined unsigned integer which identifies the icon. This identifier is normally a macro and defined through the resource editor.

The value returned is an object representing the icon loaded, or `NULL` if the icon was not found.

Example:

```
object  icn;

icn = mLoad(ExternalIcon, MY_ICON);
```

See also: `LoadIcon::Window`, `SetIcon::Application`

`LoadStr::ExternalIcon` [LoadStr]

```
icn = mLoadStr(ExternalIcon, id);

char    *id;    /*  icon id      */
object   icn;   /*  icon object  */
```

This method is used to load a programmer defined application specific icon by name.

`id` is a programmer defined name which identifies the icon. This identifier is normally defined through the resource editor.

The value returned is an object representing the icon loaded, or `NULL` if the icon was not found.

Example:

```
object  icn;

icn = mLoadStr(ExternalIcon, "myicon");
```

See also: `LoadIcon::Window`, `SetIcon::Application`, `Use::Window`

4.12 Fonts

The `Font` class (as well as its subclasses) are used to represent font objects. This class is never used by an application. It is used to house the functionality common to its subclasses. Therefore, this class documents functionality which is common to all of its subclasses.

Note that a more convenient mechanism for using fonts is provided by `LoadFont::Window`, `LoadSystemFont::Window`, `LoadFont::Printer`, and `LoadSystemFont::Printer`.

`AveCharWidth::Font` [AveCharWidth]

```
r = gAveCharWidth(fnt);

object   fnt;  /*  a font object  */
int      r;    /*  ave char width */
```

This method is used to gain access to the average character width associated with a particular font.

Example:

```
object  fnt;
int     cw;

cw = gAveCharWidth(fnt);
```

See also: `LineHeight`, `GetTM`

`Copy::Font` [Copy]

```
c = gCopy(fnt);

object  fnt;    /*  font object          */
object  c;      /*  copy of font object  */
```

This method is used to create a new font object which is a copy of a given font object. The value of this is to be able to associate a font with multiple objects which will automatically dispose of the font when they are disposed. If copies weren't used the second object referencing the font would reference a disposed font object.

Each font object must be disposed (either automatically or manually) when it is no longer needed.

Example:

```
object  f1, f2;

f2 = gCopy(f1);
```

See also: `DeepCopy`

`DeepCopy::Font` [DeepCopy]

This method performs the same function as `Copy`. See that method for details.

`DeepDispose::Font` [DeepDispose]

This method performs the same function as `Dispose`. See that method for details.

`Dispose::Font` [Dispose]

```
r = gDispose(fnt);

object  fnt;   /*  a font object  */
object  r;     /*  NULL           */
```

This method is used to dispose of a font object when it is no longer needed. This method is rarely needed due to the fact that when a window or printer object is disposed it automatically calls this method to dispose of its associated font object.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  fnt;

fnt = gDispose(fnt);
```

See also: `DeepDispose`

`GetTM::Font` [GetTM]

```
r = gGetTM(fnt, tm);

object      fnt;  /*  a font object  */
TEXTMETRIC  *tm;  /*  font metrics   */
object      r;    /*  fnt            */
```

This method is used to gain access to the font metrics associated with a particular font.

Example:

```
object  fnt;
TEXTMETRIC  tm;

gGetTM(fnt, &tm);
```

See also: `LineHeight`, `AveCharWidth`

`Handle::Font` [Handle]

```
h = gHandle(fnt);

object  fnt;    /*  font object     */
HANDLE  h;      /*  Windows handle  */
```

This method is used to obtain the Windows internal handle associated with a font object.

Note that this method may be used with most WDS objects in order to obtain the internal handle that Windows normally associates with each type of object. See the appropriate documentation.

Example:

```
object  fnt;
HGDIOBJ h;

h = gHandle(fnt);
```

`LineHeight::Font` [LineHeight]

```
r = gLineHeight(fnt);

object   fnt;  /*  a font object  */
int      r;    /*  line height    */
```

This method is used to gain access to the line height associated with a particular font. This number is equal to the height of the largest character in the font plus a reasonable amount of space to separate lines containing the font.

Example:

```
object  fnt;
int     lh;

lh = gLineHeight(fnt);
```

See also: `AveCharWidth, GetTM`

4.12.1 System Fonts

The `SystemFont` class represents Windows predefined fonts. This class is a subclass of `Font` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Font` class for additional functionality.

`Load::SystemFont` [Load]

```
fnt = mLoad(SystemFont, id);

unsigned id;    /*  icon id      */
object   fnt;   /*  icon object  */
```

This method is used to create a new object which represents one of the Windows predefined fonts. `id` should be one of the Windows defined macros specifying the desired font. It is defined in the Windows documentation under the function `GetStockObject` and ends with `_FONT`.

Example:

```
object  fnt;

fnt = mLoad(SystemFont, ANSI_FIXED_FONT);
```

See also: `LoadSystemFont::Window, LoadSystemFont::Printer, SetFont::Application, Use::Window, Use::Printer`

4.12.2 External Fonts

The `ExternalFont` class represents arbitrary external fonts. This class is a subclass of `Font` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Font` class for additional functionality.

Indirect::ExternalFont [Indirect]

```
fnt = gIndirect(ExternalFont, lf);

LOGFONT *lf;    /*  logical font  */
object  fnt;    /*  font object   */
```

This method is used to find and load a font which most closely matches the parameters indicated by `lf`. The structure of `lf` is fully described in the Windows documentation under the structure `LOGFONT`.

The value returned is an object representing the font loaded, or `NULL` if the font was not found.

Example:

```
LOGFONT lf;
object  fnt;

/*  initialize lf  */
fnt = gIndirect(ExternalFont, &lf);
```

See also: `LoadFont::Window`, `LoadFont::Printer`, `SetFont::Application`, `Use::Window`, `Use::Printer`

New::ExternalFont [New]

```
fnt = vNew(ExternalFont, nm, ps);

char    *nm;    /*  font name    */
int     ps;     /*  point size   */
object  fnt;    /*  font object  */
```

This method is used to load an arbitrary font by name at any point size. `nm` is the full name of the font as it appears when you list the available fonts via the control-panel / fonts Windows utility, minus the font type in parentheses. `ps` indicates the desired point size.

The value returned is an object representing the font loaded, or `NULL` if the font was not found.

Note that `nm` may also be a Dynace object cast as a (`char *`).

Example:

```
object  fnt;

fnt = vNew(ExternalFont, "Times New Roman", 12);
```

See also: `LoadFont::Window`, `LoadFont::Printer`, `SetFont::Application`, `Use::Window`, `Use::Printer`

4.13 Brushes

The `Brush` class (as well as its subclasses) are used to represent brush objects. This class is never used by an application. It is used to house the functionality common to its subclasses. Therefore, this class documents functionality which is common to all of its subclasses.

`Color::Brush` [Color]

```
c = gColor(bsh);

object    bsh;  /*  brush object   */
COLORREF  c;    /*  brush color    */
```

This method is used to obtain the color associated with a brush.

Example:

```
object  bsh;
COLORREF  c;

c = gColor(bsh);
```

`Copy::Brush` [Copy]

```
c = gCopy(bsh);

object  bsh;    /*  brush object          */
object  c;      /*  copy of brush object  */
```

This method is used to create a new brush object which is a copy of a given brush object. The value of this is to be able to associate a brush with multiple objects which will automatically dispose of the brush when they are disposed. If copies weren't used the second object referencing the brush would reference a disposed brush object.

Each brush object must be disposed (either automatically or manually) when it is no longer needed.

Example:

```
object  b1, b2;

b2 = gCopy(b1);
```

See also: `DeepCopy`

`DeepCopy::Brush` [DeepCopy]

This method performs the same function as `Copy`. See that method for details.

`DeepDispose::Brush` [DeepDispose]

This method performs the same function as `Dispose`. See that method for details.

`Dispose::Brush` [Dispose]

```
r = gDispose(bsh);

object  bsh;   /*  a brush object  */
object  r;     /*  NULL            */
```

This method is used to dispose of a brush object when it is no longer needed. This method is rarely needed due to the fact that when a window or printer object is disposed it automatically calls this method to dispose of its associated brush object.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  bsh;

bsh = gDispose(bsh);
```

See also: `DeepDispose`

`Handle::Brush` [Handle]

```
h = gHandle(bsh);

object  bsh;    /*  brush object    */
HANDLE  h;      /*  Windows handle  */
```

This method is used to obtain the Windows internal handle associated with a brush object.

Note that this method may be used with most WDS objects in order to obtain the internal handle that Windows normally associates with each type of object. See the appropriate documentation.

Example:

```
object  bsh;
HBRUSH  h;

h = gHandle(bsh);
```

4.13.1 Stock Brushes

The `StockBrush` class represents common Windows defined brushes. This class is a subclass of `Brush` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Brush` class for additional functionality.

`New::StockBrush` [New]

```
bsh = vNew(StockBrush, id);

unsigned id;    /*  brush id      */
object  bsh;    /*  brush object  */
```

This method is used to load one of the Windows pre-defined brushes. The available options are macros documented in the Windows documentation under the function `GetStockObject` and end with `_BRUSH`.

The value returned is an object representing the brush loaded, or `NULL` if the brush was not found.

Example:

```
object  bsh;

bsh = vNew(StockBrush, GRAY_BRUSH);
```

See also: `TextBrush::Window`, `BackBrush::Window`, `Use::Window`, `Use::Printer`

4.13.2 Solid Brushes

The `SolidBrush` class represents arbitrary application defined brushes. This class is a subclass of `Brush` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Brush` class for additional functionality.

`New::SolidBrush` [New]

```
bsh = vNew(SolidBrush, red, green, blue);

int     red;
int     green;
int     blue;
object  bsh;    /*  brush object  */
```

This method is used to create a solid colored brush with specific colors. Each color argument specifies a number between 0 and 255 and indicates the intensity of the related color. If all three are 0 you get black, and all three at 255 is white.

The value returned is an object representing the brush created, or `NULL` if the brush was not created.

Example:

```
object  bsh;

bsh = vNew(SolidBrush, 255, 0, 0);
```

See also: `TextBrush::Window`, `BackBrush::Window`, `Use::Window`, `Use::Printer`

4.13.3 System Brushes

The `SystemBrush` class represents brushes which are those colors which were selected by the user as global to their Windows environment. The programmer may select, for example, the color the user chose for background windows.

This class is a subclass of `Brush` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Brush` class for additional functionality.

`New::SystemBrush` [New]

```
bsh = vNew(SystemBrush, id);

unsigned id;    /*  brush id      */
object  bsh;    /*  brush object  */
```

This method is used to load one of the Windows user defined brushes. The available options are macros documented in the Windows documentation under the function `GetSysColor`.

The value returned is an object representing the brush loaded, or `NULL` if the brush was not found.

Example:

```
object  bsh;

bsh = vNew(SystemBrush, COLOR_ACTIVEBORDER);
```

See also: `TextBrush::Window`, `BackBrush::Window`, `Use::Window`, `Use::Printer`

4.13.4 Hatch Brushes

The `HatchBrush` class represents arbitrary application defined brushes with a specified pattern. This class is a subclass of `Brush` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Brush` class for additional functionality.

`New::HatchBrush` [New]

```
bsh = vNew(HatchBrush, red, green, blue, style);

int     red;
int     green;
int     blue;
int     style;  /*  brush pattern  */
object  bsh;    /*  brush object   */
```

This method is used to create a colored brush with specific colors and pattern Each color argument specifies a number between 0 and 255 and indicates the intensity of the related color. If all three are 0 you get black, and all three at 255 is white.

The `style` parameter indicates the specific pattern for the brush. This parameter is a macro documented in the Windows documentation under the function `CreateHatchBrush` and begin with `HS_`.

The value returned is an object representing the brush created, or `NULL` if the brush was not created.

Example:

```
object  bsh;

bsh = vNew(HatchBrush, 255, 10, 10, HS_VERTICAL);
```

See also: `TextBrush::Window`, `BackBrush::Window`, `Use::Window`, `Use::Printer`

4.14 Pens

The `Pen` class (as well as its subclasses) are used to represent pen objects. This class is never used by an application. It is used to house the functionality common to its subclasses. Therefore, this class documents functionality which is common to all of its subclasses.

`Color::Pen` [Color]

```
c = gColor(pn);

object    pn;   /*  pen object   */
COLORREF  c;    /*  pen color    */
```

This method is used to obtain the color associated with a pen.

Example:

```
object    pn;
COLORREF  c;

c = gColor(pn);
```

`Copy::Pen` [Copy]

```
c = gCopy(pn);

object  pn;     /*  pen object          */
object  c;      /*  copy of pen object  */
```

This method is used to create a new pen object which is a copy of a given pen object. The value of this is to be able to associate a pen with multiple objects which will automatically dispose of the pen when they are disposed. If copies weren't used the second object referencing the pen would reference a disposed pen object.

Each pen object must be disposed (either automatically or manually) when it is no longer needed.

Example:

```
object  p1, p2;

p2 = gCopy(p1);
```

See also: `DeepCopy`

`DeepCopy::Pen` [DeepCopy]

This method performs the same function as `Copy`. See that method for details.

`DeepDispose::Pen` [DeepDispose]

This method performs the same function as `Dispose`. See that method for details.

`Dispose::Pen` [Dispose]

```
r = gDispose(pn);

object  pn;   /*  a pen object  */
object  r;    /*  NULL          */
```

This method is used to dispose of a pen object when it is no longer needed. This method is rarely needed due to the fact that when a window or printer object is disposed it automatically calls this method to dispose of its associated pen object.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  pn;

pn = gDispose(pn);
```

See also: `DeepDispose`

`Handle::Pen` [Handle]

```
h = gHandle(pn);

object  pn;     /*  pen object    */
HANDLE  h;      /*  Windows handle  */
```

This method is used to obtain the Windows internal handle associated with a pen object.

Note that this method may be used with most WDS objects in order to obtain the internal handle that Windows normally associates with each type of object. See the appropriate documentation.

Example:

```
object  pn;
HPEN    h;

h = gHandle(pn);
```

4.14.1 Stock Pens

The `StockPen` class represents common Windows defined pens. This class is a subclass of `Pen` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Pen` class for additional functionality.

`New::StockPen` [New]

```
pn = vNew(StockPen, id);

unsigned id;    /*  pen id      */
object  pn;     /*  pen object  */
```

This method is used to load one of the Windows pre-defined pens. The available options are macros documented in the Windows documentation under the function `GetStockObject` and end with `_PEN`.

The value returned is an object representing the pen loaded, or `NULL` if the pen was not found.

Example:

```
object  pn;

pn = vNew(StockPen, BLACK_PEN);
```

See also: `Use::Printer`

4.14.2 Custom Pens

The `CustomPen` class represents arbitrary application defined pens with a specified pattern and width. This class is a subclass of `Pen` and as such inherits most of its functionality from it. This section documents the methods particular to this class. See the `Pen` class for additional functionality.

`New::CustomPen` [New]

```
pn = vNew(CustomPen, red, green, blue, style, width);

int     red;
int     green;
int     blue;
int     style;  /*  pen pattern     */
int     width;  /*  pen line width  */
object  pn;     /*  pen object      */
```

This method is used to create a colored pen with specific colors, pattern and line width. Each color argument specifies a number between 0 and 255 and indicates the intensity of the related color. If all three are 0 you get black, and all three at 255 is white.

The `style` parameter indicates the specific pattern for the pen. This parameter is a macro documented in the Windows documentation under the function `CreatePen` and begin with `PS_`. The `width` parameter specifies the width of the line in logical units.

The value returned is an object representing the pen created, or `NULL` if the pen was not created.

Example:

```
object  pn;

pn = vNew(CustomPen, 255, 10, 10, PS_DASH, 5);
```

See also: `Use::Printer`

4.15 Help System

The `HelpSystem` class is used to support the standard Windows help system. In addition to being able to support arbitrary evocation of help messages, this class is used by other WDS classes in order to provide complete support for context sensitive help to any level. Each class provides mechanisms in order to associated context sensitive help text to windows, dialogs, menus or controls.

Most classes which provide context sensitive help support have a method called `SetTopic` which is used to associate a particular topic within a given context.

All the methods in this class are class methods. This means that the first argument to all the methods will be `HelpSystem` and there is never an instance object to keep track of.

See the appropriate examples for detailed information on how to create the actual help file.

`HelpContents::HelpSystem` [HelpContents]

```
r = gHelpContents(HelpSystem);

object  r;      /*  HelpSystem  */
```

This method is used to display the contents screen of the help file. `HelpSystem` will be returned if the request is successful and `NULL` otherwise.

Example:

```
gHelpContents(HelpSystem);
```

See also: `HelpFile`, `HelpTopic`, `HelpInContext`

`HelpFile::HelpSystem` [HelpFile]

```
r = gHelpFile(HelpSystem, hf);

char    *hf;    /*  help file   */
object  r;      /*  HelpSystem  */
```

This method is used to determine the external file to be used for all help references. This must be done prior to any of the other help facilities use. The help file will not be opened until one of the help display methods is evoked. The application must also have a main window prior to the evocation of the help system. The help system will automatically terminate when the user terminates the application.

Example:

```
gHelpFile(HelpSystem, "helpfile.hlp");
```

See also: `HelpContents`

`HelpInContext::HelpSystem` [HelpInContext]

```
r = gHelpInContext(HelpSystem);

object  r;      /*  HelpSystem  */
```

This method is used to display the help screen associated with the current context (set with `SetTopic`). If no context is set the help contents will be displayed. `HelpSystem` will be returned if the request is successful and `NULL` otherwise.

Note that `SetTopic` and `HelpInContext`, although available to an application program, are mainly used internally by WDS in order to support the context sensitive

help facility provided by the `Window`, `Dialog`, `Menu`, and `Control` classes. See the `SetTopic` method associated with those classes.

Example:

```
gHelpInContext(HelpSystem);
```

See also: `HelpFile, SetTopic, HelpContents, HelpTopic`

`HelpTopic::HelpSystem` [HelpTopic]

```
r = gHelpTopic(HelpSystem, tpc);

char    *tpc;   /*  help topic  */
object  r;      /*  HelpSystem  */
```

This method is used to display the help screen associated with a given topic. `HelpSystem` will be returned if the request is successful and `NULL` otherwise.

Example:

```
gHelpTopic(HelpSystem, "fileOpen");
```

See also: `HelpFile, HelpContents, HelpInContext`

`SetTopic::HelpSystem` [SetTopic]

```
r = gSetTopic(HelpSystem, tpc);

char    *tpc;   /*  help topic      */
char    *r;     /*  previous topic  */
```

This method is used to set the current help topic context. It is used by `HelpInContext` to display the help text associated with the current context. The previous help topic will be returned.

Note that `SetTopic` and `HelpInContext`, although available to an application program, are mainly used internally by WDS in order to support the context sensitive help facility provided by the `Window`, `Dialog`, `Menu`, and `Control` classes. See the `SetTopic` method associated with those classes.

Example:

```
gSetTopic(HelpSystem, "someTopic");
```

See also: `HelpFile, HelpInContext, HelpContents, HelpTopic`

4.16 Common Dialogs

The `CommonDialog` class is only used to group the common dialogs which are all subclasses of this class. There is no specific functionality associated with this class. See the particular subclasses for documentation.

4.16.1 File Selection Dialog

The `FileDialog` class provides a standard and convenient method of querying the user for a file name. The user is able to browse the disk or enter a new file name.

The `fd` parameter used by all methods in this class refer to the file dialog object returned by the `New` method.

`AppendFilter::FileDialog` [AppendFilter]

```
r = gAppendFilter(fd, ttl, fltr);

object  fd;     /*  file dialog  */
char    *ttl;   /*  title        */
char    *fltr;  /*  file filter  */
object  r;      /*  file dialog  */
```

This method is used to group files according to some file naming criteria for user selection. There may be any number of groups. The user selects the group they are interested in and are presented with a list of files which meet the criteria.

`ttl` is the name of the group which the user is presented with. `fltr` is the file filter and may consist of several filters, each separated with a semicolon.

This method may be called any number of times to create several groups.

Example:

```
object  fd;

gAppendFilter(fd, "Document Files", "*.txt;*.doc");
gAppendFilter(fd, "Source Files", "*.c;*.h");
```

See also: `SetFile`, `InitialDir`, `DefExt`

`DefExt::FileDialog` [DefExt]

```
r = gDefExt(fd, ext);

object  fd;     /*  file dialog     */
char    *ext;   /*  file extension  */
object  r;      /*  file dialog     */
```

This method is used to set the default file extension which will be displayed for the user. It should not contain a period.

Example:

```
object  fd;

gDefExt(fd, "txt");
```

See also: `SetFile, AppendFilter`

`Dispose::FileDialog` [Dispose]

```
r = gDispose(fd);

object  fd;   /*  file dialog  */
object  r;    /*  NULL         */
```

This method is used to dispose of a file dialog object. This method must be called to dispose of the dialog when it is no longer needed.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  fd;

fd = gDispose(fd);
```

`GetFile::FileDialog` [GetFile]

```
fname = gGetFile(fd);

object  fd;     /*  file dialog  */
char    *fname; /*  file name    */
```

This method is used to obtain the file name the user selected once the file dialog has been completed. This value will include the full path, file name and extension. If multiple selections are allowed, the path will be returned, then a space delimited list of selected file names.

Example:

```
object  fd;
char    *files;

files = gGetFile(fd);
```

See also: `GetOpenFile`, `GetSaveFile`, `SetFile`, `SetFlags`

`InitialDir::FileDialog` [InitialDir]

```
r = gInitialDir(fd, pth);

object  fd;     /*  file dialog  */
char    *pth;   /*  initial path */
object  r;      /*  file dialog  */
```

This method is used to set the initial path associated with the file dialog. If no initial path is set, the system will use the current directory of the application.

Example:

```
object  fd;

gInitialDir(fd, "c:\\mypath");
```

See also: `SetFile`, `AppendFilter`, `DefExt`

`New::FileDialog` [New]

```
fd = vNew(FileDialog, pwind);

object  pwind;  /*  parent window  */
object  fd;     /*  file dialog    */
```

This method is used to create a new file dialog object. It will not be displayed until the appropriate method is called. The object returned must be disposed when it no longer needed.

The `pwind` parameter refers to either the application's main window or any child window object.

Example:

```
object  fd, pwind;

fd = vNew(FileDialog, pwind);
```

See also: `Dispose`, `GetOpenFile`, `GetSaveFile`

`GetOpenFile::FileDialog` [GetOpenFile]

```
r = gGetOpenFile(fd);

object  fd;     /*  file dialog    */
int     r;      /*  return status  */
```

This method is used to actually display the dialog and obtain user input. The type of file dialog presented will be one in which the user must select a pre-existing file, presumably to open.

The return value is non-zero if the user makes a valid selection and zero if the user did not make a selection or if an error occurred.

Example:

```
object  fd;
int     r;

r = gGetOpenFile(fd);
```

See also: `GetSaveFile`

`GetSaveFile::FileDialog` [GetSaveFile]

```
r = gGetSaveFile(fd);

object  fd;     /*  file dialog    */
int     r;      /*  return status  */
```

This method is used to actually display the dialog and obtain user input. The type of file dialog presented will be one in which the user may select a pre-existing file or type in a new name, presumably to save some data to.

The return value is non-zero if the user makes a valid selection and zero if the user did not make a selection or if an error occurred.

Example:

```
object  fd;
int     r;

r = gGetSaveFile(fd);
```

See also: `GetOpenFile`

SetFile::FileDialog [SetFile]

```
r = gSetFile(fd, fname);

object  fd;     /*  file dialog  */
char    *fname; /*  file name    */
object  r;      /*  file dialog  */
```

This method is used to set the default file name displayed when the file dialog is displayed.

Example:

```
object  fd;

gSetFile(fd, "somefile.txt");
```

See also: `InitialDir`, `GetFile`, `SetFlags`, `DefExt`

SetFlags::FileDialog [SetFlags]

```
r = gSetFlags(fd, flgs);

object  fd;     /*  file dialog  */
DWORD   flgs;   /*  flags        */
object  r;      /*  file dialog  */
```

This method is used to set the option flags associated with a file dialog. The available flags are fully documented in the Windows documentation under the `OPENFILENAME` structure and all begin with `OFN_`.

Example:

```
object  fd;

gSetFlags(fd, OFN_ALLOWMULTISELECT);
```

See also: `SetFile`, `DefExt`

SetTitle::FileDialog [SetTitle]

```
r = gSetTitle(fd, ttl);

object  fd;     /*  file dialog  */
char    *ttl;   /*  dialog title */
object  r;      /*  file dialog  */
```

This method is used to set the text which appears at the top of the file dialog. If no value is set the system will use appropriate defaults.

Example:

```
object  fd;

gSetTitle(fd, "Select File");
```

See also: `SetFile`, `AppendFilter`, `DefExt`

4.16.2 Printer Selection and Configuration Dialog

The `PrintDialog` class is used to present the user with a standard and convenient dialog for printer selection and configuration.

This class is seldom needed because the `Printer` class has the `QueryPrinter::Printer` method which automatically calls this class for printer information. However, this class may be used independently in order to have better control of user options.

The `pd` parameter used by all methods in this class refer to the print dialog object returned by the `New` method.

`Copies::PrintDialog` [Copies]

```
cp = gCopies(fd);

object  fd;     /*  file dialog  */
int     cp;     /*  copies       */
```

This method is used to obtain the number of copies selected by the user subsequent to calling `Perform`.

Example:

```
object  fd;
int     cp;

cp = gCopies(fd);
```

See also: `Perform`, `SetFlags`, `GetPageRange`

`Dispose::PrintDialog` [Dispose]

```
r = gDispose(pd);

object  pd;     /*  print dialog   */
object  r;      /*  NULL           */
```

This method is used to dispose of a print dialog object. This must be done when it is no longer needed.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  pd;

pd = gDispose(pd);
```

See also: `New`, `Perform`

`GetPageRange::PrintDialog` [GetPageRange]

```
r = gGetPageRange(fd, start, end);

object  fd;     /*  file dialog           */
int     *start; /*  starting page number  */
int     *end;   /*  ending page number    */
object  r;      /*  file dialog           */
```

This method is used to obtain the page range selected by the user subsequent to calling `Perform`.

Example:

```
object  fd;
int     start, end;

gGetPageRange(fd, &start, &end);
```

See also: `Perform`, `SetFlags`, `Copies`

`Handle::PrintDialog` [Handle]

```
hdc = gHandle(pd);

object  pd;     /*  print dialog   */
HANDLE  hdc;    /*  handle to dc   */
```

This method is used to obtain the device context handle associated with the print dialog. It will only be valid after performing the dialog (via `Perform`). If the handle is obtained via this method, the `Dispose` method will not release the handle as it would normally do. This is done so the device context can be used for the printer to

be opened. If the handle as obtained it is the application's responsibility to release the handle when it is no longer needed. This may be done via the `DeleteDC` Windows function.

This method is mainly used internally to WDS and is only made available for convenience.

Example:

```
object  pd;
HDC     hdc;

hdc = gHandle(pd);
```

See also: `Perform`

`New::PrintDialog` [New]

```
pd = vNew(PrintDialog, pwind);

object  pwind;  /*  parent window  */
object  pd;     /*  print dialog   */
```

This method is used to create a new print dialog object. It will not be displayed until the appropriate method is called. The object returned must be disposed when it no longer needed.

The `pwind` parameter refers to either the application's main window or any child window object.

Example:

```
object  pd, pwind;

pd = vNew(PrintDialog, pwind);
```

See also: `Perform, Dispose`

`Perform::PrintDialog` [Perform]

```
r = gPerform(pd);

object  pd;     /*  print dialog   */
int     r;      /*  return status  */
```

This method is used to actually present the user with the print dialog. The return value is non-zero if the user appropriately selects a printer. Zero will be returned if the user cancels the dialog or an error occurs.

Example:

```
object  pd;
int     r;

r = gPerform(pd);
```

See also: `New`, `Dispose`

`SetFlags::PrintDialog` [SetFlags]

```
r = gSetFlags(pd, flgs);

object  pd;     /*  print dialog  */
DWORD   flgs;   /*  flags         */
object  r;      /*  print dialog  */
```

This method is used to set the option flags associated with a print dialog. The available flags are fully documented in the Windows documentation under the `PRINTDLG` structure and all begin with `PD_`.

Example:

```
object  pd;

gSetFlags(pd, PD_ALLPAGES);
```

4.16.3 Color Selection Dialog

The `ColorDialog` class is used to present the user with a standard and convenient dialog for color selection and configuration.

The `cd` parameter used by all methods in this class refer to the color dialog object returned by the `New` method.

`Dispose::ColorDialog` [Dispose]

```
r = gDispose(cd);

object  cd;     /*  color dialog   */
object  r;      /*  NULL           */
```

This method is used to dispose of a color dialog object. This must be done when it is no longer needed.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  cd;

cd = gDispose(cd);
```

See also: `New`, `Perform`

`GetColor::ColorDialog` [GetColor]

```
clr = gGetColor(cd);

object   cd;    /*  color dialog    */
COLORREF clr;   /*  selected color  */
```

This method is used to obtain the users color selection subsequent to executing `Perform`.

Example:

```
object  cd;
COLORREF clr;

clr = gGetColor(cd);
```

See also: `Perform`, `SetColor`, `GetCustomColorPtr`

`GetCustomColorPtr::ColorDialog` [GetCustomColorPtr]

```
cv = gGetCustomColorPtr(cd);

object   cd;    /*  color dialog    */
COLORREF *cv;   /*  color vector    */
```

This method is used to obtain a pointer to 16 `COLORREF`s selected as custom colors by the user. This vector should be inspected subsequent to calling `Perform` and prior to calling `Dispose`. One `Dispose` is called, the returned pointer will no longer be valid.

Example:

```
object  cd;
COLORREF *cv;

cv = gGetCustomColorPtr(cd);
```

See also: `Perform`, `GetColor`

`New::ColorDialog` [New]

```
cd = vNew(ColorDialog, pwind);

object  pwind;  /*  parent window  */
object  cd;     /*  color dialog   */
```

This method is used to create a new color dialog object. It will not be displayed until the appropriate method is called. The object returned must be disposed when it no longer needed.

The `pwind` parameter refers to either the application's main window or any child window object.

Example:

```
object  cd, pwind;

cd = vNew(ColorDialog, pwind);
```

See also: `Perform`, `Dispose`

`Perform::ColorDialog` [Perform]

```
r = gPerform(cd);

object  cd;     /*  color dialog   */
int     r;      /*  return status  */
```

This method is used to actually present the user with the color selection dialog. The return value is non-zero if the user appropriately selects a color. Zero will be returned if the user canceled the dialog or an error occurred.

Example:

```
object  cd;
int     r;

r = gPerform(cd);
```

See also: `New`, `Dispose`

`SetColor::ColorDialog` [SetColor]

```
r = gSetColor(cd, clr);

object   cd;    /*  color dialog    */
COLORREF clr;   /*  selected color  */
object   r;     /*  color dialog    */
```

This method is used to set the initial color associated with the color selection dialog. The Windows `RGB` macro may be used to obtain a valid `COLORREF`.

Example:

```
object  cd;
COLORREF clr;

gSetColor(cd, RGB(255, 0 ,0));
```

See also: `Perform`, `GetColor`

`SetFlags::ColorDialog` [SetFlags]

```
r = gSetFlags(cd, flgs);

object  cd;     /*  color dialog  */
DWORD   flgs;   /*  flags         */
object  r;      /*  color dialog  */
```

This method is used to set the option flags associated with a color dialog. The available flags are fully documented in the Windows documentation under the `CHOOSECOLOR` structure and all begin with `CC_`.

Example:

```
object  cd;

gSetFlags(cd, CC_FULLOPEN);
```

4.16.4 Font Selection Dialog

The `FontDialog` class is used to present the user with a standard and convenient dialog for font selection.

The `fd` parameter used by all methods in this class refer to the font dialog object returned by the `New` method.

Dispose::FontDialog [Dispose]

```
r = gDispose(fd);

object  fd;     /*  font dialog   */
object  r;      /*  NULL          */
```

This method is used to dispose of a font dialog object. This must be done when it is no longer needed.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  fd;

fd = gDispose(fd);
```

See also: `New`, `Perform`

Font::FontDialog [Font]

```
fnt = gFont(fd);

object  fd;     /*  font dialog  */
object  font;   /*  font object  */
```

This method is used to create a font object representing the font the user selected. This font object will be an instance of the `ExternalFont` class and must be disposed (either directly or indirectly) when it is no longer needed. This method should only be called subsequent to `Perform`. `NULL` will be returned if no font was selected or the font object could not be created.

Example:

```
object  fd, fnt;

fnt = gFont(fd);
```

See also: `Perform`

GetColor::FontDialog [GetColor]

```
clr = gGetColor(fd);

object   fd;    /*  font dialog  */
COLORREF clr;   /*  color        */
```

This method is used to get the user selected color once `Perform` has been called. This will only work if the `CF_EFFECTS` option is set.

Example:

```
object  fd;
COLORREF clr;

clr = gGetColor(fd);
```

See also: `SetFlags`, `SetColor`, `Perform`

New::FontDialog [New]

```
fd = vNew(FontDialog, pwind);

object  pwind;  /*  parent window  */
object  fd;     /*  font dialog    */
```

This method is used to create a new font dialog object. It will not be displayed until the appropriate method is called. The object returned must be disposed when it no longer needed.

The `pwind` parameter refers to either the application's main window or any child window object.

Example:

```
object  fd, pwind;

fd = vNew(FontDialog, pwind);
```

See also: `Perform`, `Dispose`

Perform::FontDialog [Perform]

```
r = gPerform(fd);

object  fd;     /*  font dialog    */
int     r;      /*  return status  */
```

This method is used to actually present the user with the font selection dialog. The return value is non-zero if the user appropriately selects a font. Zero will be returned if the user canceled the dialog or an error occurred.

Example:

```
object  fd;
int     r;

r = gPerform(fd);
```

See also: `New`, `Dispose`

SetColor::FontDialog [SetColor]

```
clr = gSetColor(fd);

object   fd;    /*  font dialog  */
COLORREF clr;   /*  color        */
```

This method is used to set the initial color of the user color selection portion of the font dialog. This will only work if the `CF_EFFECTS` option is set. The Windows supplied `RGB` macro may be used to create the `COLORREF`.

Example:

```
object  fd;

gSetColor(fd, RGB(255, 0, 0));
```

See also: `SetFlags`, `GetColor`, `Perform`

SetFlags::FontDialog [SetFlags]

```
r = gSetFlags(fd, flgs);

object  fd;     /*  font dialog  */
DWORD   flgs;   /*  flags        */
object  r;      /*  font dialog  */
```

This method is used to set the option flags associated with a font dialog. The available flags are fully documented in the Windows documentation under the `CHOOSEFONT` structure and all begin with `CF_`.

Example:

```
object  fd;

gSetFlags(fd, CF_ANSIONLY);
```

4.17 Dynamic Link Libraries

The `DynamicLibrary` class is used to support the dynamic link library (DLL) facility of Windows. This class provides a convenient mechanism to load, free and use DLLs.

Throughout this section `dl` will refer to the object which was returned by the `LoadLibrary` method and represents a DLL.

`DeepDispose::DynamicLibrary` [DeepDispose]

This method performs the same function as `Dispose`. See that method for details.

`Dispose::DynamicLibrary` [Dispose]

```
r = gDispose(dl);

object  dl;   /*  a DLL object  */
object  r;    /*  NULL          */
```

This method is used to release and dispose of a DLL object when it is no longer needed. All DLL objects are automatically released by WDS when the application terminates.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  dl;

dl = gDispose(dl);
```

See also: `FreeAll`, `DeepDispose`

FindStr::DynamicLibrary [FindStr]

```
dl = gFindStr(DynamicLibrary, fname);

char    *fname; /*  DLL file name  */
object  dl;     /*  DLL object     */
```

This class method is used to obtain the DLL object (previously loaded via `LoadLibrary`) via its associated file name. This won't work after the DLL object is disposed.

Example:

```
object  dl;

dl = gFindStr(DynamicLibrary, "mydll.dll");
```

See also: `LoadLibrary`

FreeAll::DynamicLibrary [FreeAll]

```
gFreeAll(DynamicLibrary);
```

This class method is used to release and dispose of all DLL objects. It is automatically called by WDS when the application terminates.

Example:

```
gFreeAll(DynamicLibrary);
```

See also: `Dispose`, `DeepDispose`

GetProcAddress::DynamicLibrary [GetProcAddress]

```
fun = gGetProcAddress(dl, fname);

object  dl;     /*  DLL object          */
char    *fname; /*  function name       */
FARPROC fun;    /*  pointer to function */
```

This method is used to obtain a pointer to a function within a DLL. The DLL function may then be evoked with the pointer.

Example:

```
FARPROC fun;
object  dl;

fun = gGetProcAddress(dl, "myfunc");
```

See also: `FindStr`, `LoadLibrary`

`LoadLibrary::DynamicLibrary` [LoadLibrary]

```
dl = gLoadLibrary(DynamicLibrary, file);

char    *file;  /*  DLL file    */
object  dl;     /*  DLL object  */
```

This method is used to open a DLL file and create a WDS object which represents the loaded library. If the named file does not contain a path, the normal Windows search path will be searched. If the DLL cannot be found or loaded, `NULL` will be returned.

Example:

```
object  dl;

dl = gLoadLibrary(DynamicLibrary, "mydll.dll");
```

See also: `GetProcAddress`, `Dispose`, `FindStr`

Method Index

A

AddControl::Dialog 95
AddDlgHandlerAfter::Dialog 96
AddDlgHandlerBefore::Dialog 98
AddEdtHandlerAfter::ComboBox 158
AddEdtHandlerBefore::ComboBox 159
AddHandlerAfter::Control 116
AddHandlerAfter::Window 36
AddHandlerBefore::Control 118
AddHandlerBefore::Window 38
AddMenuOption::InternalMenu 90
AddMenuOption::PopupMenu 92
AddOption::ComboBox 160
AddOption::ListBox 148
AddPopupMenu::InternalMenu 91
AddPopupMenu::PopupMenu 92
AddSeparator::PopupMenu 93
Alphabetize::ComboBox 160
Alphabetize::ListBox 149
AppendFilter::FileDialog 197
Arc::Printer 71
Associate::Menu 84
Associate::Window 38
Attach::CheckBox 141
Attach::ComboBox 161
Attach::DateControl 135
Attach::ListBox 149
Attach::NumericControl 128
Attach::RadioButton 144
Attach::ScrollBar 169
Attach::TextControl 123
AutoDispose::Dialog 98
AutoDispose::Window 39
AutoShow::Window 40
AveCharWidth::Font 181

B

BackBrush::Dialog 99
BackBrush::Window 40

C

CallDefaultProc::Control 118
Capitalize::TextControl 123
CheckFunction::Control 119
CheckValue::Control 119
Chord::Printer 72
CmdLine::Application 25
Color::Brush 186
Color::Pen 191
CompletionFunction::Dialog 99
Copies::PrintDialog 202
Copy::Brush 186
Copy::Cursor 174
Copy::Font 181
Copy::Icon 177
Copy::Pen 191
CtlDoubleValue::Dialog 100
CtlLongValue::Dialog 100
CtlShortValue::Dialog 101
CtlStringValue::Dialog 102
CtlUnsignedShortValue::Dialog 102
CtlValue::Dialog 103

D

DateRange::DateControl 135
DeepCopy::Brush 187
DeepCopy::Cursor 174
DeepCopy::Font 182
DeepCopy::Icon 178
DeepCopy::Pen 192
DeepDispose::Brush 187
DeepDispose::Control 120
DeepDispose::Cursor 174
DeepDispose::Dialog 103
DeepDispose::DynamicLibrary 212
DeepDispose::Font 182
DeepDispose::Icon 178
DeepDispose::Menu 85
DeepDispose::Pen 192
DeepDispose::Printer 73
DefaultEdtProcessingMode::ComboBox 161
DefaultProcessingMode::Control 120
DefaultProcessingMode::Window 41
DefExt::FileDialog 197
Dispose::Brush 187
Dispose::ColorDialog 205
Dispose::Control 121
Dispose::Cursor 175
Dispose::Dialog 104
Dispose::DynamicLibrary 212
Dispose::FileDialog 198
Dispose::Font 182
Dispose::FontDialog 209
Dispose::Icon 178
Dispose::Menu 86
Dispose::Pen 192
Dispose::PopupMenu 93
Dispose::PrintDialog 202
Dispose::Printer 73
Dispose::Window 41
DoubleValue::NumericControl 128

E

Ellipse::Printer 73
Erase::Window 42
EraseAll::Window 42
EraseLines::Window 43
Error::Application 25

F

FindStr::DynamicLibrary 213
Font::FontDialog 209
FreeAll::DynamicLibrary 213

G

GetBackBrush::Application 26
GetBackBrush::Dialog 104
Getch::Window 43
GetColor::ColorDialog 206
GetColor::FontDialog 210
GetControl::Dialog 105
GetCursor::Application 26
GetCustomColorPtr::ColorDialog 206
GetFile::FileDialog 198
GetFont::Application 27
GetIcon::Application 27
GetName::Application 28
GetName::Window 44
GetOpenFile::FileDialog 200
GetPageRange::PrintDialog 203
GetParent::Dialog 105
GetParent::Window 44
GetPosition::Window 45
GetProcAddress::DynamicLibrary 213
Gets::Window 45
GetSaveFile::FileDialog 200
GetScalingMode::Application 28
GetSelections::ListBox 150
GetSize::Application 29
GetSize::Window 46
GetTag::Dialog 105
GetTag::Window 46
GetTextBrush::Application 28
GetTextBrush::Dialog 106
GetTM::Font 183
Group::RadioButton 145

H

Handle::Brush 187
Handle::Control 121
Handle::Cursor 175
Handle::Dialog 106
Handle::Font 183
Handle::Icon 179
Handle::Menu 86
Handle::Pen 192
Handle::PopupMenu 94
Handle::PrintDialog 203
Handle::Printer 74
Handle::Window 47
HelpContents::HelpSystem 195
HelpFile::HelpSystem 195
HelpInContext::HelpSystem 195
HelpTopic::HelpSystem 196

I

Increment::ScrollBar 170
IndexValue::Dialog 107
InDialog::Dialog 107
Indirect::ExternalFont 185
InitialDir::FileDialog 199
Instance::Application 29

K

Kbhit::Window 47

L

Line::Printer 74
LineHeight::Font 183
ListIndex::ComboBox 162
ListIndex::ListBox 150
Load::ExternalCursor 176
Load::ExternalIcon 180
Load::ExternalMenu 89
Load::SystemFont 184
LoadCursor::Window 48
LoadFont::Printer 75
LoadFont::Window 48
LoadIcon::Window 49
LoadLibrary::DynamicLibrary 214
LoadMenu::Window 49
LoadMenuStr::Window 50
LoadStr::ExternalCursor 177
LoadStr::ExternalIcon 180
LoadStr::ExternalMenu 89
LoadSys::SystemCursor 176
LoadSys::SystemIcon 179
LoadSystemCursor::Window 51
LoadSystemFont::Printer 76
LoadSystemFont::Window 51

LoadSystemIcon::Window 52
LongValue::DateControl 136
LongValue::NumericControl 129

M

MaxLength::TextControl 124
MenuFunction::Menu 87
MenuFunction::PopupMenu 94
MenuItemMode::Window 52
Message::Dialog 108
Message::Window 53
MessageWithTopic::Dialog 108
MessageWithTopic::Window 53
MinLength::TextControl 124

N

New::ChildWindow 69
New::ColorDialog 207
New::CustomPen 194
New::ExternalFont 185
New::FileDialog 199
New::FontDialog 210
New::HatchBrush 190
New::InternalMenu 92
New::MainWindow 68
New::PopupMenu 94
New::PopupWindow 70
New::PrintDialog 204
New::Printer 76
New::SolidBrush 189
New::StockBrush 188
New::StockPen 193
New::SystemBrush 189
New::Window 54
NewBuiltIn::Window 54
NewCtl::CheckBox 141
NewCtl::ComboBox 163
NewCtl::DateControl 136
NewCtl::ListBox 151
NewCtl::NumericControl 129
NewCtl::PushButton 139
NewCtl::RadioButton 145
NewCtl::ScrollBar 170
NewCtl::TextControl 125
NewDialog::ModalDialog 113
NewDialog::ModelessDialog 114
NewDialogStr::ModalDialog 113
NewDialogStr::ModelessDialog 115
NewPage::Printer 77
NewWithHDC::Printer 77
NumbSelected::ListBox 151
NumericRange::NumericControl 130

P

Perform::ColorDialog 207
Perform::Dialog 109
Perform::FontDialog 211
Perform::PrintDialog 204
Perform::PushButton 139
Pie::Printer 78
Pop::Menu 87
PopMenu::Window 55
PrevInstance::Application 30
Printf::Printer 79
Printf::Window 55
ProcessMessages::MainWindow 68
Push::Menu 87
Puts::Printer 79
Puts::Window 56

Q

QueryPrinter::Printer 80
QuitApplication::Application 30

R

Read::Window 57
Rectangle::Printer 81
RemoveAll::ComboBox 163
RemoveAll::ListBox 151
RemoveInt::ComboBox 163
RemoveInt::ListBox 152
RemoveStr::ComboBox 164
RemoveStr::ListBox 152
Required::ComboBox 164
Required::ListBox 153
RoundRect::Printer 81

S

ScaleToCurrentMode::Application 31
ScaleToPixels::Application 31
ScrollBarRange::ScrollBar 171
ScrollHorz::Window 57
ScrollVert::Window 58
SetBackBrush::Application 32
SetBlock::Window 58
SetColor::ColorDialog 208
SetColor::FontDialog 211
SetCursor::Application 33
SetDoubleValue::NumericControl 130
SetFile::FileDialog 201
SetFlags::ColorDialog 208
SetFlags::FileDialog 201
SetFlags::FontDialog 211
SetFlags::PrintDialog 205
SetFont::Application 33
SetFunction::ComboBox 165

SetFunction::ListBox 153
SetFunction::PushButton 140
SetIcon::Application 34
SetLongValue::DateControl 137
SetLongValue::NumericControl 131
SetMaxLines::Window 59
SetMode::Menu 88
SetName::Application 34
SetName::Window 60
SetParent::Window 60
SetPosition::Window 61
SetRaw::Window 61
SetResult::Dialog 110
SetScale::Printer 82
SetScalingMode::Application 34
SetShortValue::CheckBox 142
SetShortValue::ComboBox 166
SetShortValue::ListBox 154
SetShortValue::NumericControl 131
SetShortValue::RadioButton 146
SetShortValue::ScrollBar 172
SetSize::Window 62
SetStringValue::ComboBox 166
SetStringValue::ListBox 154
SetStringValue::TextControl 125
SetStyle::Window 62
SetTag::Dialog 111
SetTag::Window 63
SetTextBrush::Application 35
SetTitle::FileDialog 201
SetTopic::Control 122
SetTopic::Dialog 112
SetTopic::HelpSystem 196
SetTopic::Window 63
SetUShortValue::NumericControl 132
SetValue::CheckBox 142
SetValue::ComboBox 167
SetValue::DateControl 137
SetValue::ListBox 155
SetValue::NumericControl 132
SetValue::RadioButton 146
SetValue::ScrollBar 172
SetValue::TextControl 126
ShortValue::CheckBox 143
ShortValue::ComboBox 167
ShortValue::ListBox 156
ShortValue::NumericControl 133
ShortValue::RadioButton 147
ShortValue::ScrollBar 173
Show::Application 36
Show::Window 64
StringValue::ComboBox 168
StringValue::ListBox 156
StringValue::TextControl 126

T

TextBrush::Dialog 112
TextBrush::Window 64
TextOut::Printer 82
TextOut::Window 65
TextRange::TextControl 127

U

UnsignedShortValue::NumericControl 133
Update::Window 65
Use::Printer 83
Use::Window 66

V

Value::CheckBox 143
Value::ComboBox 168
Value::DateControl 138
Value::ListBox 157
Value::NumericControl 134
Value::RadioButton 147
Value::ScrollBar 173
Value::TextControl 127
ValueAt::ComboBox 169
ValueAt::ListBox 157
VertShift::Window 67

W

Write::Window 67