z i n c

PROGRAMMER'S GUIDE

INTERFACE LIBRARY™

VERSION 3.0

A new paradigm. . .

Zinc™ Interface Library™

Programmer's Guide

Version 3.0

Zinc Software Incorporated
Pleasant Grove, Utah

TABLE OF CONTENTS

Section I—GETTING STARTED

Section II—BASIC CONCEPTS

Section III—ZINC DESIGNER

Section IV—ADVANCED CONCEPTS

SECTION I
GETTING STARTED

CHAPTER 1 – INTRODUCTION

Overview

Zinc Interface Library is a powerful user interface library that uses unique features of C++, including virtual functions, class inheritance, operator overloading, multiple inheritance, etc. This library is developed specifically for C++ and is compatible with AT&T's C++ V2.1 and ANSI C.

System requirements

To develop applications for DOS you need Zinc Interface Library (DOS version), a C++ compiler for DOS (i.e., Borland C++, Microsoft C++, or Zortech C++), DOS 2.1 or later (DOS 3.1 or later is recommended for accurate country information), 640K RAM and a hard disk drive. For mouse support, you need a mouse and a Microsoft mouse compatible driver.

To develop applications for Microsoft Windows 3.X you need Zinc Interface Library (DOS and Windows version), a C++ compiler for Windows (i.e., Borland C++, Microsoft C++, or Zortech C++) and Microsoft Windows 3.0 or later.

Using The Manuals

Programmer's Guide

The documentation for Zinc Interface Library is contained in three manuals: *Programmer's Guide*, *Programmer's Reference*, and *Programmer's Tutorials*. The *Programmer's Guide* provides an overview to Zinc Interface Library. It contains the following sections:

Section 1—Getting Started gives an overview of Zinc Interface Library, tells how to install the library package on your computer, and how to ship Zinc Interface Library based applications.

Section 2—Basic Concepts gives a high-level description of Zinc Interface Library, including the conceptual operation of the library and its major pieces. An introduction on how to use Zinc for Windows applications is also given.

Section 3—Zinc Designer explains how to create application screens and help information using Zinc's interactive design tool.

Section 4—Advanced Concepts gives an indepth description of Zinc Interface Library and C++ programming. It is recommended that you read this section prior to beginning an application.

Programmer's Reference

The *Programmer's Reference* contains descriptions of Zinc Interface Library classes, the calling conventions used to invoke the class member functions, short code samples using the class member functions, and information about other related classes or example programs. The *Programmer's Reference* contains the following sections:

Class object information—This section (Introduction) contains the class hierarchy and include file (**.HPP**) information associated with class objects and structures available within Zinc Interface Library.

Class object references—This section (Chapters 1 through 66) contains short descriptions about the class objects (or structures), the public and protected member variables and functions, and the calling conventions used with the class object.

Miscellaneous information—This section (Appendix A through F) contains support definitions, system event definitions, logical event definitions, class identifications, storage information and raw DOS keyboard scan code information.

Programmer's Tutorial

The *Programmer's Tutorial* contains a series of tutorials designed to help the programmer in learning the features and practical uses of Zinc Interface Library.

Section 1—Hello World! tells you how to initialize (first four tutorials) and modify (last tutorial) the main components of Zinc Interface Library.

Section 2—Dictionary describes the transition from C programming to C++ programming, adding Zinc Interface Library to an existing application, and using the Zinc storage file.

Section 3—Zinc Application Program describes the overall design and implementation issues you should be concerned about when creating applications using Zinc Interface Library.

Section 4—Derived Classes contains a set of tutorials that show how Zinc Interface Library classes can be modified to perform customized operations. This section should only be read by programmers who want to derive objects in their applications.

Section 5—Persistent Objects contains a set of tutorials that present the concept of persistent objects (i.e., objects that can be stored to and retrieved from disk).

Section 6—Miscellaneous Information (Appendix A through E) contains compiler considerations, compiled BGI files, example programs, Zinc coding standards and common questions and answers.

Typefaces

Special typefaces are used throughout the documentation in order to clarify the usage and meaning of specific terms. Familiarity with the following typeface conventions will be helpful in working with these manuals:

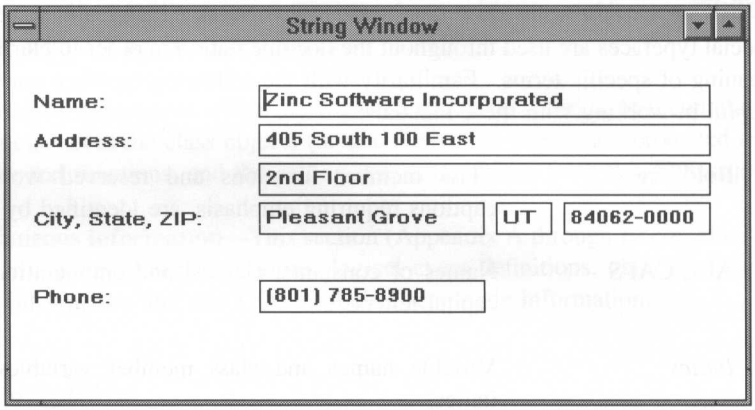
Boldface	Class member functions and reserved words, as well as captions requiring emphasis, are identified by boldface type.
ALL CAPS	Names of constants, classes, and enumerations appear in all capital letters.
Italics	Variable names and class member variables are shown in italics.
`Monospace`	Text as it appears on the screen or within a program is presented in monospace type.
CAPS BOLDFACE	Names of files appear in all capitals and boldface type. (Note: Often the names of constants, classes, and enumerations appear in all capital boldface as part of a caption. This is done only to place emphasis on the words and does not distinguish them as file names.)
[ ]	Optional input items that are dependant upon the system you use are enclosed by square brackets.
< >	Include files, specific keys to be entered from the keyboard, and mouse movements are enclosed in angle brackets.

Underline — Words that require particular emphasis within text are underlined.

Terminology

The following terms are used extensively throughout the documentation:

Field—A window object that can be edited. For example, the border of a window is not considered to be a "field" whereas a number is considered to be a field. The figure below shows a window with several fields. (The fields are shown with outlining borders.)



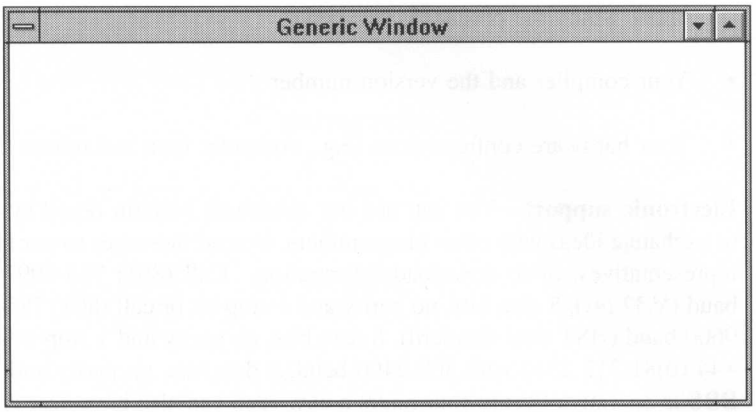
UI_—The prefix identification for all class objects used in Zinc Interface Library. The "UI" stands for "User Interface." This prefix allows programmers to distinguish the user interface part of their application.

UID_—The prefix identification for all device class objects used in the library. The "UID" stands for "User Interface Device" object. All UID type objects are derived from the UI_DEVICE base class.

UIW_—The prefix identification for all window class objects used in the library. The "UIW" stands for "User Interface Window" object. All UIW type objects are derived from the UI_WINDOW_OBJECT base class.

Window—A region of the screen that contains one or more window objects. A window is used by the end user to view or edit information associated with the application program. A window is represented by the UIW_WINDOW class object.

In the figure below, the window is shown as the main rectangle and all blank portions within the rectangle. All non-blank portions of the window are window objects (the border, buttons and title bar).



Window field—A window object that can be edited. This term is synonymous to "field."

Window object—A class object derived from the UI_WINDOW_OBJECT base class. Window objects are used in the context of a parent window or are themselves windows that are attached to the screen display. The figure above shows a window with several window objects (a border, 3 buttons and a title bar).

Zinc Technical Support

Zinc Software Incorporated offers a complete technical support program to registered users, so be sure to complete and return the registration card. As a registered user you will be eligible for the following support services:

Limited warranty—The terms of your limited warranty are explained in the Zinc Interface Library End User Software License Agreement.

Telephone support—If you need assistance beyond what the Zinc Interface Library manuals or your reseller can provide, you can call (801) 785-8998 between the hours of 8:00 a.m. and 5:00 p.m. Mountain Standard Time and talk with one of our technical support representatives at no charge. In Europe call +44 (0)81 855 9918 between 9:00 a.m. and 5:00 p.m. London Time. Technical Support is closed on

Saturdays, Sundays, and holidays. Please have the following information ready before you call:

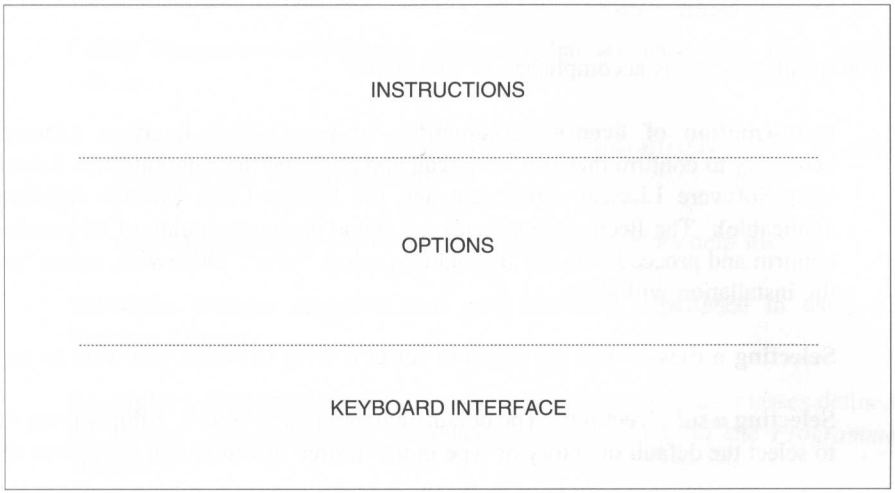
- Your Zinc Interface Library version and serial number, as well as the name under which the product is registered
- Your compiler and the version number
- Your hardware configurations (e.g., computer type and mouse brand)

Electronic support—You can use our electronic bulletin board system (i.e., BBS) to exchange ideas with other programmers, to send messages to our technical support representatives, or to download information. Call (801) 785-8997 with 300-9600 baud (V.32 *bis*), 8 data bits, no parity and 1 stop bit or call (801) 785-8995 with 300-9600 baud (HST dual standard), 8 data bits, no parity and 1 stop bit. In Europe call +44 (0)81 317 2310 with 300-2400 baud, 8 data bits, no parity and 1 stop bit. The BBS is operative twenty-four hours a day. You can also have access to the technical support department via facsimile. Call (801) 785-8996, or +44 (0)81 316 7778 in Europe, twenty-four hours a day. (If you need to send more than one page of code, please use the BBS.)

Special offers—You can receive special promotional offers for new products and product upgrades.

CHAPTER 2 – INSTALLATION

The general structure of all screens in the install program is divided into three sections:



Instructions—This upper section of the screen gives instructions about the next install operation to be performed.

Options—This middle section of the screen displays the selectable options at a particular point in the installation.

Keyboard interface—This lower section of the screen identifies which keys activate the current operation or how to move within the screen.

Before actually installing Zinc Interface Library, we recommend that you back-up your distribution disks.

Pressing <Esc> at any time during the installation will cause the program to abort.

Installation procedure

Installation of Zinc Interface Library requires DOS 2.1 or later, 640K RAM, and a hard disk drive. The installation program copies files to a hard disk or network. It does not modify any system file (e.g., **AUTOEXEC.BAT**, **CONFIG.SYS**.)

Insert the first distribution disk into the desired drive, make it the current drive and invoke the installation program. For example, to install Zinc Interface Library from drive A, insert the first disk and type:

```
a:<Enter>
install<Enter>
```

The install process is accomplished in five steps:

Confirmation of license agreements—To install Zinc Interface Library, it is necessary to confirm that you have read and accepted the Zinc Interface Library End User Software License Agreement and the Source Code License Agreement (if applicable). The license agreements are found at the beginning of this manual. To confirm and proceed with the installation, select "yes." Otherwise, select "no" and the installation will abort.

Selecting a drive—You are asked to select a drive to which you want to install.

Selecting a subdirectory—The default subdirectory is \ZINC. Simply press <Enter> to select the default directory or type in the desired directory and then press <Enter>.

Selecting the package option—You are asked to identify which Zinc Interface Library package you purchased. The options are:

- **ZIL 3.0 for DOS only**
- **ZIL 3.0 for DOS with source**
- **ZIL 3.0 for DOS and Windows**
- **ZIL 3.0 for DOS and Windows with source**

Selecting the compiler—You are asked which compiler libraries you wish to install.

Selecting portions to install—You are asked which portions of Zinc Interface Library you want to install. The options are:

DOS Library Files—Contains all library class functions for the DOS version of Zinc Interface Library.

- **ZIL.LIB**—Contains all library class functions for the DOS version of Zinc Interface Library.

Windows Library Files—Contains all library class functions for the Windows version of Zinc Interface Library.

- **ZILW.LIB**—Contains all library class functions for the Windows version of Zinc Interface Library.

Utility Programs—Application programs that are used with Zinc Interface Library.

- **DESIGN.EXE**—Utility program used to interactively create or modify windows and window objects.

- **GENHELP.EXE**—Utility program used to generate help files.

Tutorials—Sample programs that give hands-on experience in using Zinc Interface Library.

Examples—Example C++ files that show how to use specific classes defined in the library. These files are referenced in Appendix C of the *Programmer's Tutorial*.

Selecting "yes" for any of these options will install that portion of the library to your hard drive.

Installation—The program now commences installing the selected material from the distribution disks to your hard drive. The progress of this installation appears on your screen.

Periodically, a prompt for a new disk will appear. Remove the current disk from the drive, insert the disk requested and press any key to continue the installation.

When the process is complete, a message appears on your screen indicating that Zinc Interface Library has been successfully installed.

Shipping applications

Be sure to include the following run-time files (i.e., Distributable Files) when you ship your applications:

- **.DAT** files generated by **GENHELP.EXE** (used by the UI_HELP_SYSTEM class).

- **.DAT** files generated by **DESIGN.EXE** (used by the UI_STORAGE class).
- **ZILD.DLL** file is the DLL version of Zinc Interface Library for use with Microsoft Windows 3.X.

You may also need to include the following run-time files used by your compiler:

- Borland **.BGI** and **.CHR** files (if the application uses the UI_BGI_DISPLAY class). Please note that the **.BGI** files can be linked into an application.
- Microsoft **.FON** files (if the application uses the UI_MSC_DISPLAY class).

SECTION II
BASIC CONCEPTS

CHAPTER 3 – CONCEPTUAL DESIGN

The software dilemma

It seems that software developers are constantly trailing behind the advances of hardware developers. An author commented on this dilemma, "At the onset of the 1990's, software lags behind hardware capabilities by at least two processor generations, and the lag is increasing. There is general agreement that conventional software tools, techniques and abstractions are rapidly becoming inadequate as software systems grow larger and increasingly more complex."[1]

Conventional (i.e., procedural) programming tools are rarely designed with the extensibility to easily integrate new technologies. Developers are therefore frequently forced into starting over in order to include these new technologies in their products.

This dilemma is magnified as software developers struggle to support multiple operating environments in an effort to maximize market opportunities. Development resources are almost always scarce and are diluted when they are allocated to "porting" existing applications rather than being applied to new product development. Additional development resources are also required to maintain and enhance the different versions of an application—further compounding the resource problem.

An object-oriented solution

Zinc Interface Library 3.0 is a new generation object-oriented development tool. Zinc 3.0 helps you easily solve problems related to the software dilemma. With Zinc's single-source support for DOS and Windows, porting becomes a trivial process. You only have one set of source code to maintain so your development resources aren't consumed trying to manage several versions of the same product. Zinc's object-oriented, event-driven architecture is open and extensible by design. With Zinc's modularity you won't find yourself painted into a corner.

In addition to a robust and comprehensive user interface class library, Zinc 3.0 also features the most tightly integrated interactive design tool available with a class library. Zinc Designer accelerates your development cycle by allowing you to interactively design your user interface. Because Zinc Designer was created with the Zinc class library you have direct access to all of the library's features. You also benefit from Zinc Designers' multiplatform storage technology. Screens that you create with Zinc Designer are saved as platform-independent resources. You can develop your interface using the Windows version of Zinc Designer, save it to disk and then retrieve it into the DOS version of Zinc Designer and vice versa.

The C++ pep talk

Like many programmers you may have developed a high degree of proficiency in a structured language such as C. You might question the need to learn the new features of C++ (and more importantly, a new approach to programming). However, as you study this conceptual overview will see many compelling benefits of object-orientation.

The transition to object-oriented programming is not a trivial endeavor. But Zinc Interface Library is a great place to start. Zinc's class hierarchy is straightforward and consistent. The constructors will allow you to build a great deal of your application with very little effort. If you have an existing application that you are updating you will be able to use a lot of your existing code. And Zinc's tutorial and reference manual will help you understand the features and benefits of object-oriented programming. Once you develop an appreciation for the benefits of object-oriented programming with Zinc Interface Library, you should be sufficiently motivated to start incorporating object-oriented techniques in other parts of your programs.

The benefits of OOP

Zinc's object orientation offers you several significant benefits over procedural approaches to interface design.

Consistency—Because of its object-oriented nature, Zinc completely eliminates the problems associated with developing and maintaining multiple versions of source code for multiple platforms. You can focus your efforts on developing, maintaining and enhancing one set of source code and let Zinc manage low-level interactions with the operating environment and screen display, whether it's DOS Text, DOS Graphics or Microsoft Windows.

Ease-of-Use—Zinc Designer lets you create your application screens interactively. Instead of generating source code which is difficult to optimize and not object oriented, Zinc Designer saves your user interface as platform-independent resources. These resources are easily modified with Zinc Designer. Zinc's object-oriented design uses data abstraction to insulate you from the complexities of the operating environment without restricting your access to environment specific features, like Microsoft Windows messages or the raw scan codes from the keyboard. The modular design of Zinc Interface Library is also conceptually intuitive and easy to understand.

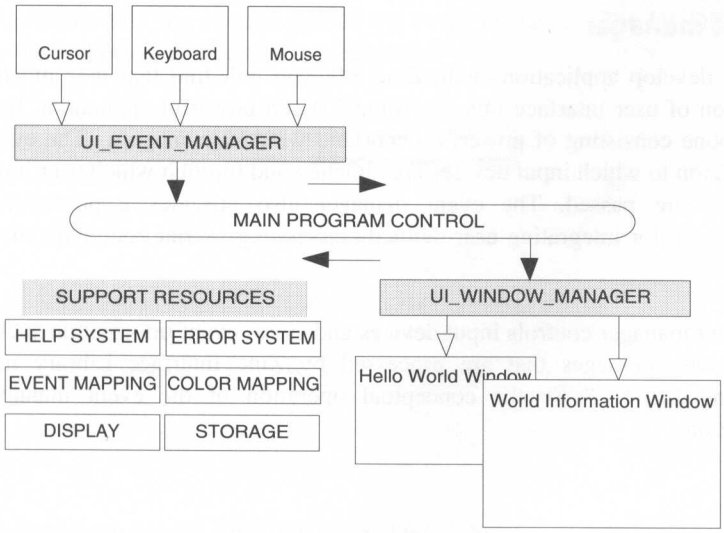
Reusability—Not only are Zinc's base classes reusable, but any object or class that you create can become a part of your tool kit. You save time by using classes that have previously been tested and debugged.

Extensibility—Because Zinc Interface Library is designed from the ground up as an object-oriented class library, you benefit from a powerful feature of OOP—inheritance. Rather than developing an object from scratch you can use Zinc's base classes (with their existing member functions and data) to derive new classes. For example, you can create a new input device such as a digitizer by deriving a new class from Zinc's device class. Thus, your effort is spent creating only the unique characteristics of the new class.

Maintenance—Object-oriented applications are much easier to maintain than structured programs. The data-hiding or encapsulation capability of C++ keeps relevant data and functions together and allows you to modify an object without affecting other parts of the application.

Zinc Interface Library

Zinc Interface Library's simple, yet powerful, architecture (shown below) allows you to quickly develop full-featured object-oriented applications.



The main sections of the library are:

Event manager—Controls the flow of end user input and system messages throughout the application.

Window manager—Controls the presentation of windows and window objects to the screen display.

Screen display—Controls the low-level screen interface for DOS Text, DOS Graphics or Microsoft Windows 3.X applications.

Help system—Controls the presentation of help information during the run-time operation of an application.

Error system—Controls the presentation of error information during the run-time operation of an application.

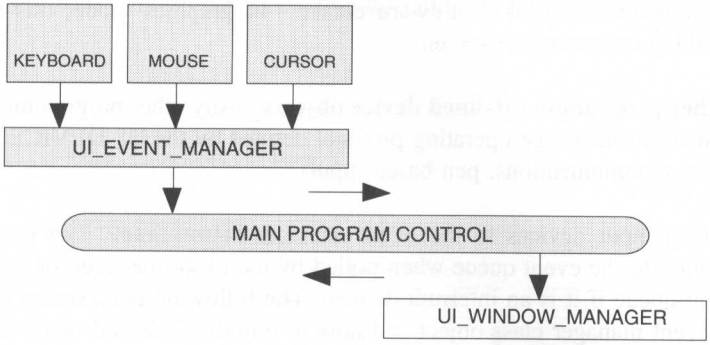
Event mapping—Controls the mapping of raw input (e.g., Microsoft Windows messages, keyboard scan codes) to logical system events (e.g., sizing, moving, redrawing).

Storage and retrieval—Controls the reading and writing of C++ objects to and from disk.

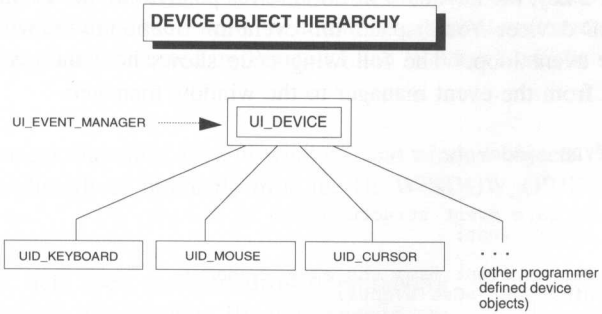
The event manager

As you develop applications with Zinc 3.0, you will find that it is much more than a collection of user interface objects. Zinc 3.0 is a powerful application framework with a backbone consisting of powerful event and window managers. The event manager is the skeleton to which input devices are attached and through which user input and system messages are passed. The event manager also provides a powerful development environment for integrating user-defined classes (e.g., serial communications, pen based input).

The event manager controls input devices and stores event information such as user input and system messages that are processed by Zinc Interface Library modules. The following figure shows the conceptual operation of the event manager in a Zinc application:



Most compiler libraries have a set of functions to get input information from the keyboard (e.g., **getch( )**, **getchar( )**) but seldom have functions to get information from other devices, such as a mouse. They also don't provide functions to integrate multiple input devices. With Zinc Interface Library, all input devices (e.g., keyboard, mouse, and user-defined input devices) are integrated to let you easily control the user's input. This interface is handled by the control part of the event manager. The UI_DEVICE class object has the following device object hierarchy:



Classes derived from the UI_DEVICE base class include:

UID_KEYBOARD—A BIOS level polled keyboard interface that retrieves keyboard information from the end user.

UID_MOUSE—An interrupt driven mouse interface that receives mouse information from the end user.

UID_CURSOR—A blinking cursor shown on the screen. In text mode, this device is implemented as the hardware cursor. In graphics mode, this device paints a blinking cursor on the screen.

Other programmer defined device objects—Any other programmer defined device that conforms to the operating protocol defined by the UI_DEVICE base class (e.g., serial communications, pen based input).

You attach input devices to the event manager at run-time. The device feeds input information to the event queue when polled by the event manager, or feeds it directly to the event queue if it is an interrupt device. The following code shows how to construct a new event manager class object and how to initialize selected input devices:

```
// Construct the screen display with the Zinc text display constructor.
UI_DISPLAY *display = new UI_TEXT_DISPLAY();

// Construct the event manager and attach the display.
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);

// Add the input devices to the event manager using '+' operator overload.
*eventManager
    + new UID_KEYBOARD
    + new UID_MOUSE
    + new UID_CURSOR;
```

The event manager contains another component called the event queue. All event information in a Zinc program is passed via the event queue. For example, when the end user presses a key, the keyboard information is placed into the event queue by the UID_-KEYBOARD device. You dispatch this event information to the window manager via the applications event loop. The following code shows how the event loop passes event information from the event manager to the window manager:

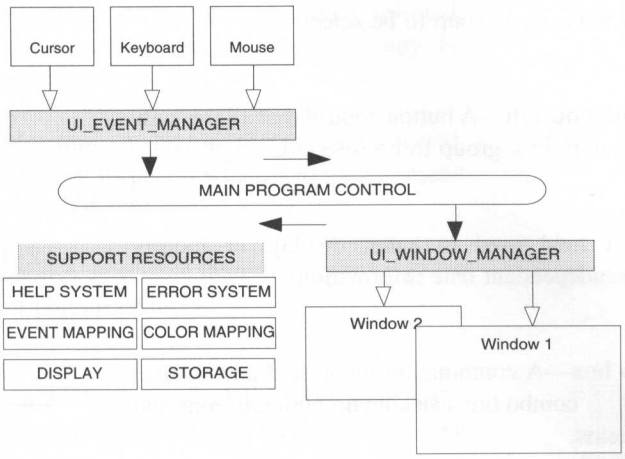
```
EVENT_TYPE ccode;
do
{
    // declare event structure.
    UI_EVENT event;

    // Get an event from the event manager.
    eventManager->Get(event);

    // Pass the event to the window manager.
    ccode = windowManager->Event(event);
} while (ccode != L_EXIT);
```

Other elements of Zinc Interface Library use the event queue to send system or private messages.

The window manager

Zinc's innovative window manager works cohesively with the event manager to control

the screen display and pass input information to the appropriate window or screen object. The illustration below shows the conceptual operation of the window manager in a Zinc application:



The window manager determines the position and priority of windows on the screen. For example, the graphic illustration above shows Window1 overlapping Window2. In this example, the window manager routes all keyboard information to Window1, since it is the top-most window attached to the screen. In addition, any mouse information that overlaps Window1 or the region intersected by Window1 and Window2 will be sent to Window1 for processing.

The window manager maintains a list of windows and window objects. These objects are all derived either directly or indirectly from the UI_WINDOW_OBJECT base class and include:

Bignum—A field used to enter, display, or modify precision numeric information. Bignum numbers are used for monetary values and high precision numbers.

0.00000000

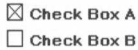
Border—An outlining border drawn around a window.

Button—A rectangular region of the screen that, when selected, performs run-time operations that you specify. The following objects are variations of the button class:

OK

Bitmapped button—A button control with an associated bitmap image.

Check box—A button control that allows multiple items in a group to be selected.

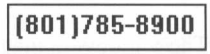


Radio button—A button control that allows only one item in a group to be selected.

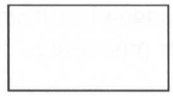
Date—A field used to enter, display, or modify country-independent date information.

Combo box—A combination input field and vertical list box. A combo box can contain buttons, icons and string fields.

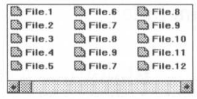
Formatted string—An input field used to enter, display or modify ascii string buffers that contain literal characters or characters that cannot be edited (e.g., phone numbers, social security numbers).



Group—A box used to provide a physical grouping of window objects such as radio buttons or check box buttons.



Horizontal list—A two-dimensional list of related items. These items are organized in a row/column fashion and may be any of the objects described in the window object hierarchy. A horizontal list contains multiple columns and scrolls horizontally.



Icon—A graphical representation of a selectable item. This object is similar to the button object, except that the information is in graphic, rather than textual, form.

Integer—A field used to enter, display, or modify integer numbers. Integer numbers are used for quantity values and indices.

100

Maximize button—A button that, when selected, changes the size of its parent window to occupy the entire screen display.

Minimize button—A button that, when selected, reduces the size of its parent window to the minimum allowed by the window.

Pop up item—A selectable item that is shown in the context of a pop-up menu.

Vt-Item 1

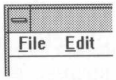
Pop up menu—A group of related UIW_POP_UP_-ITEM objects. The items in this menu are displayed on multiple lines.

Vt-Item 1
Vt-Item 2
Vt-Item 3
Vt-Item 4

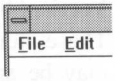
Prompt—A string that is used to describe the contents of another window field.

Fax:

Pull down item—A selectable item that is shown in the context of a pull-down menu.



Pull down menu—A group of related UIW_PULL_-DOWN_ITEM objects. The items in this menu are displayed across a single, horizontal line.



Real—A field used to enter, display, or modify floating point numeric information. Real numbers are used for computation values and fractional numbers.

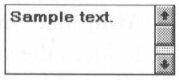
0.00000000

Scroll bar—A selectable region used to scroll the displayed portion of a window, list box or text input field.

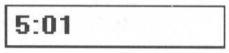
String—A field used to enter, display, or modify an ASCII string buffer.

System button—A button that, when selected, shows general operations that can be performed on the parent window.

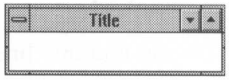
Text—A field used to enter, display, or modify a multi-line text buffer.



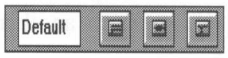
Time—A field used to enter, display, or modify country-independent time information.



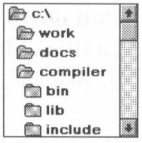
Title—An object that occupies the top region of a window and contains a window's title information.



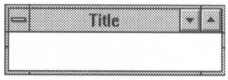
Tool bar—A group of related window objects. The tool bar is similar to a pull-down menu with the exception that it may contain different types of objects, such as: bitmapped buttons, dates, icons, strings, etc.



Vertical list—A one-dimensional list of related items. These items are organized in a single column and may be any of the objects described in the window object hierarchy.

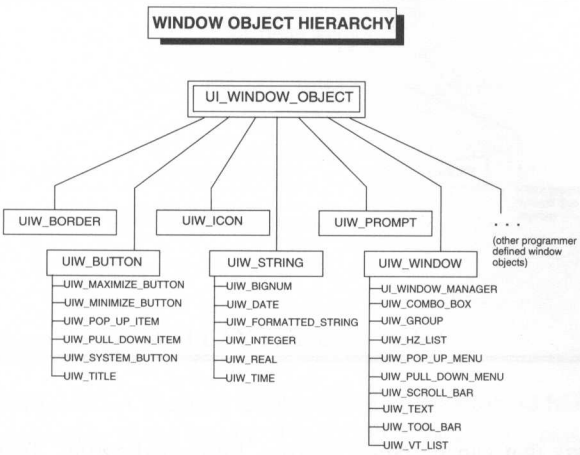


Window—A rectangular region of the screen that is composed of one or more class objects derived from the UI_WINDOW_OBJECT base class. A window may also contain sub-windows (e.g., MDI windows.)



Other programmer defined window objects—Any other programmer defined window object that conforms to the operating protocol defined by the UI_WINDOW_OBJECT base class.

The UI_WINDOW_MANAGER class object has the following window object hierarchy:

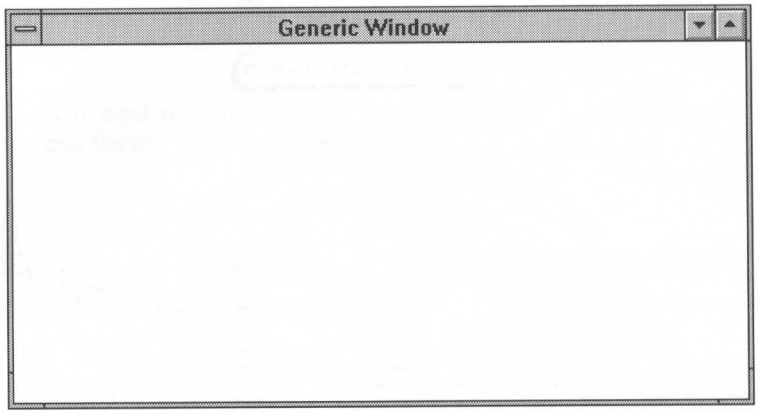


You attach windows to the window manager at run-time. Once a window is attached, it receives event information from the window manager. The following code shows how to construct a new window manager class object and how to initialize a selected window:

```
// Construct the screen display using the Zinc constructor.
UI_DISPLAY *display = new UI_MSWINDOWS_DISPLAY;

// Construct the event manager and attach the display and input devices.
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
*eventManager
    + new UID_KEYBOARD  // Use the '+' operator overload or 'add' member
    + new UID_MOUSE     // function.
    + new UID_CURSOR;

// Construct the window manager and attach it to the event manager.
UI_WINDOW_MANAGER *windowManager = new UI_EVENT_MANAGER(display,
    eventManager);

// Add a simple window to the window manager.
UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11);
*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE("General objects", WOF_NO_FLAGS);
```

```
*windowManager + window;
```

Windows and window objects have distinct representations in DOS Text, DOS Graphics, and Microsoft Windows. For example, the following shows the MS Windows representation of a simple window:



Window objects that can be edited (String, Formatted String, Text, Number, Date and Time) support the following features:

Mark—Marks an area of the current field for use with the cut or copy edit features. Marked regions are shown as shaded regions in a window field.

Cut—Cuts the marked area of the current field and stores the marked contents in a global paste buffer. This data can later be pasted into any other field, as long as the information is valid for that field type (e.g., the text "400" could be pasted into a numeric, string, or text field).

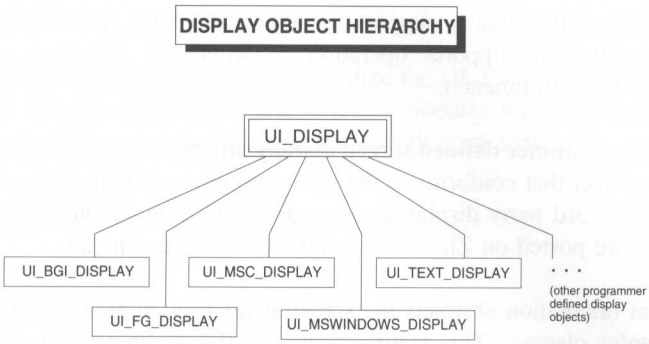
Copy—Copies the marked area of the current field and stores the marked contents in a global paste buffer. This data can later be pasted into any other field, as long as the information is valid for that field type.

Paste—Copies the contents of the global paste buffer into the current field. Data can be pasted into any field, as long as the information is valid for that field type.

The screen display

Modular display classes are supported by Zinc's screen display, which controls all low-

level screen output. The following display objects are supported by Zinc Interface Library:



Classes derived from the UI_DISPLAY base class include:

UI_BGI_DISPLAY—A graphics display that uses the Borland BGI graphics routines to display information to the screen. The UI_BGI_DISPLAY class provides support for CGA, EGA, VGA and Hercules monochrome display adapters running in graphics mode.

UI_FG_DISPLAY—A graphics display that uses the Zortech Flash Graphics routines to display information to the screen. The UI_FG_DISPLAY class provides support for CGA, EGA, VGA, SVGA and Hercules monochrome display adapters running in graphics mode.

UI_MSC_DISPLAY—A graphics display that uses the Microsoft MSC graphics routines to display information to the screen. The UI_MSC_DISPLAY class provides support for CGA, EGA, VGA and Hercules monochrome display adapters running in graphics mode.

UI_MSWINDOWS_DISPLAY—A graphics display that uses the Microsoft Windows 3.X graphics routines to display information to the screen.

UI_TEXT_DISPLAY—A text display that writes the display information to screen memory. The UI_TEXT_DISPLAY class provides support for MDA, CGA, EGA and VGA display adapters running in text mode. This includes the following modes of operation:

- 25 line x 80 column mode,

- 25 line x 40 column mode,
- 43 line x 80 column mode and
- 50 line x 80 column mode.

This class also contains support for snow checking (CGA monitors) and IBM TopView (which supports operation in Microsoft Windows and Quarterdeck desqVIEW environments).

Other programmer defined screen display objects—Any other programmer defined display object that conforms to the operating protocol defined by the UI_DISPLAY base class. 3rd party display classes supporting Metagraphics and Genus graphics libraries are posted on Zinc's BBS and are free to download.

Zinc's object orientation abstracts the screen display in an application by implementing modular display classes. This feature gives you the significant advantage of using one set of source code to produce output for DOS Text, DOS Graphics and MS Windows 3.0 environments. This modular approach will allow Zinc to support additional platforms without forcing you to dramatically alter your source code.

The following code shows how to initialize both graphic and text screen displays in one executable file:

```
// Initialize the display, trying for graphics first.
UI_DISPLAY *display = new UI_MSC_DISPLAY;
if (!display->installed)
{
    delete display;
    display = new UI_TEXT_DISPLAY;
}
```

The help system

The help system is used to present help information to the end user during an application program. The help system uses the Zinc Interface Library windowing system to present help information.

Zinc Interface Library initially does not initialize the UI_HELP_SYSTEM so that you are not forced to have the window help system modules included in your application. The following code shows how to set-up the default help system:

```
// Add in the help system.
UI_WINDOW_OBJECT::helpSystem = new UI_HELP_SYSTEM("help.dat",
    windowManager);
```

The error system

The error system is used to display error information to the end user during an application program. The error system uses the Zinc Interface Library windowing system to present error information.

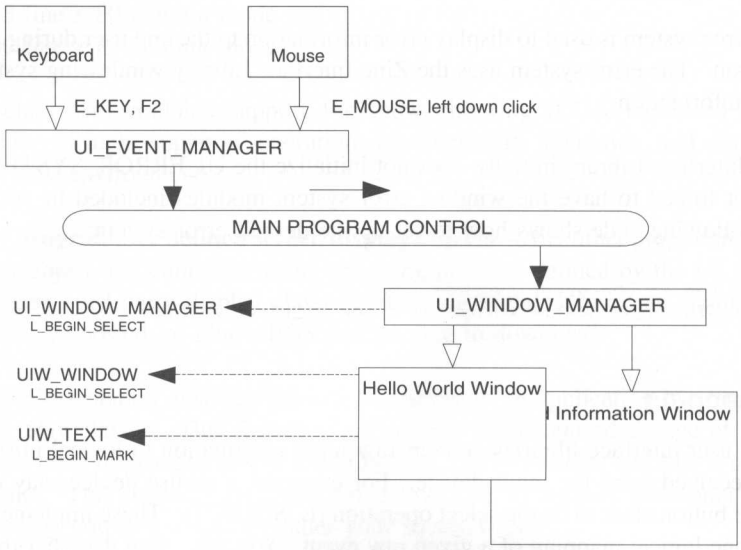
Zinc Interface Library initially does not initialize the UI_ERROR_SYSTEM so that you are not forced to have the window error system modules included in your application. The following code shows how to set-up the default error system:

```
// Add in the error system.
UI_WINDOW_OBJECT::errorSystem = new UI_ERROR_SYSTEM;
```

Event mapping

Many user interface libraries convert raw input information to logical information when it is received from the input device. For example, a mouse device may define the left mouse button click to be the select operation (L_SELECT). These implementations allow only one logical mapping of a given raw event. You must then decipher the L_SELECT operation in the context of your operations. This implementation, however, is inadequate for most applications.

Zinc Interface Library has the powerful capability of interpreting raw events, received from input devices at run-time, at each level of the application according to the type of operation. For example, the graphic illustration below shows how the <F2> key and left mouse click would be interpreted at each level in the library (where a text field is the current window object):



The <F2> key and left-mouse button are processed in the following manner:

- first, the key or mouse information is received by the input device (i.e., UID_-KEYBOARD or UID_MOUSE) and placed in the event queue.
- second, the window manager passes the event to the current window.
- third, the window passes the event to the current window object.
- fourth, the UIW_TEXT window object evaluates both the keyboard and mouse events as the L_BEGIN_MARK command.
- finally, the results of the L_BEGIN_MARK command are returned to the window and then to the window manager.

The benefits of logical event mapping are:

- Each object is allowed to interpret the event according to its mode of operation. The UIW_TEXT object views both events as an L_BEGIN_MARK operation. However, if the left-click were returned, unprocessed, to the window manager, it would be interpreted as an L_BEGIN_SELECT operation while the <F2> key (which is unknown by the window manager) would remain unprocessed.

- You can define additional input devices that generate their own raw event information. With this implementation, you can define logical event mapping for Zinc but still receive all the raw event information generated by the new input device.

- You can easily re-define key mapping without changing the source code of many modules. This allows you to customize your application without interfering with the general operation of Zinc Interface Library.

Storage and retrieval

Zinc Interface Library allows you to store and retrieve C++ objects to and from disk as platform-independent resources. This is accomplished through low-level file management routines as well as persistent object technology. These storage and retrieval classes are used when programmers interactively create and/or modify windows and window objects using Zinc Designer. You can also use the storage and retrieval classes without Zinc Designer.

Conclusion

A thorough understanding of the conceptual design of Zinc Interface Library will assist you as you develop applications. The key components of the library—event manager, window manager, screen display, help system, error system, event mapping, storage and retrieval—all work together to give you the most powerful, flexible and easy-to-use interface library available.

---

1. Winblad, Ann L., Samuel Edwards, and David R. King. Object-Oriented Software. Reading, MA: Addison-Wesley, 1990

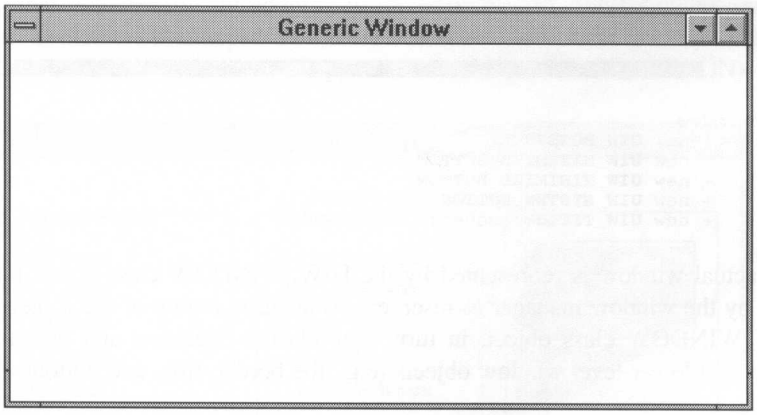
CHAPTER 4 – WINDOW OBJECTS

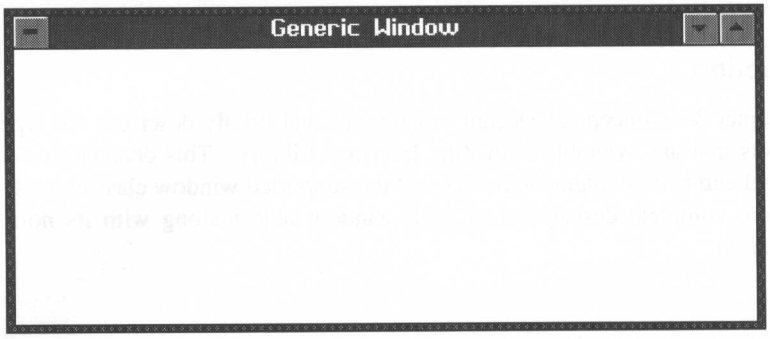
Introduction

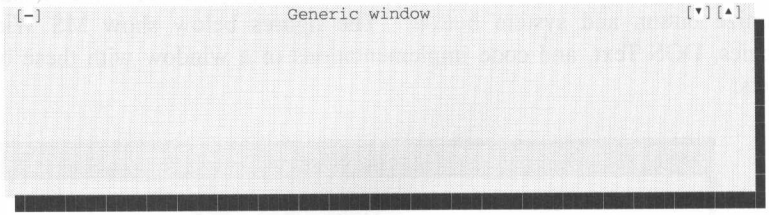
"Chapter 3—Conceptual Design" of this manual briefly describes the types of window objects that are available with Zinc Interface Library. This chapter shows the graphic, textual and code implementations of all the supported window class objects. It also gives a more complete description of each window object along with its normal modes of operation.

Basic window objects

Most windows created for an application will contain a border, title, maximize button, minimize button and system button. The figures below show MS Windows, DOS Graphics, DOS Text, and code implementations of a window with these basic window objects:








```
*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON(SYF_GENERIC);
    + new UIW_TITLE(" General objects ");
```

The actual window is represented by the UIW_WINDOW class object. This object is used by the window manager to reserve a rectangular region of the screen display. The UIW_WINDOW class object, in turn, controls the operation and presentation of any associated lower-level window objects (e.g., the border, title, and buttons shown above).

The window's border, shown as the exterior part of the windows above, is represented by the UIW_BORDER class object. If the application is running in graphics mode, the border is shown as a 3-dimensional shaded region drawn around the window. If the application is running in text mode, the window is displayed with a shadow.

The title bar, shown with the "General objects" information text on the top-center portion of the windows above, is represented by the UIW_TITLE class object. This window object is used to display textual information that uniquely identifies the window.

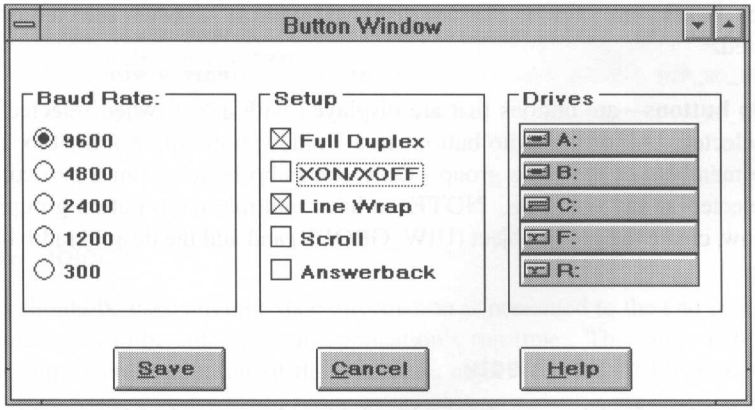
The maximize button, shown with the '▲' character on the top-right side of the windows above, is represented by the UIW_MAXIMIZE_BUTTON class object. This button is used to change the size of its parent window to occupy the entire screen display.

The minimize button, shown with the '▼' character on the top-right side of the windows above, is represented by the UIW_MINIMIZE_BUTTON class object. This button is used to reduce the window to an icon.

The system button, shown with the '–' character on the top-left side of the windows above, is represented by the UIW_SYSTEM_BUTTON class object. This button is used to select window or system specific commands associated with the window object (e.g., size, move, maximize, minimize, close). If menu items are specified with the system button, a pop-up menu is displayed to the screen.

Button window objects

A button field is a rectangular region of the screen that, when selected, performs run-time operations that you specify. In addition to the basic buttons, the following specialized buttons are available: bitmapped buttons, check boxes, and radio buttons. The figure below shows a window with different types of button fields (UIW_BUTTON):




```
*window
    + new UIW_TITLE("Button Window")

    // Add the radio buttons.
    + &(*new UIW_GROUP(1, 2, 13, 7, "Baud Rate:")
        + new UIW_BUTTON(2, 4, "9600", BTF_RADIO_BUTTON)
        + new UIW_BUTTON(2, 5, "4800", BTF_RADIO_BUTTON)
        + new UIW_BUTTON(2, 6, "2400", BTF_RADIO_BUTTON)
        + new UIW_BUTTON(2, 7, "1200", BTF_RADIO_BUTTON)
        + new UIW_BUTTON(2, 7, "300", BTF_RADIO_BUTTON))
```

```
// Add the check boxes.
+ &(*new UIW_GROUP(15, 2, 13, 7, "Setup:")
    + new UIW_BUTTON(18, 4, "Full Duplex", BTF_CHECK_BOX)
    + new UIW_BUTTON(18, 5, "XON/XOFF", BTF_CHECK_BOX)
    + new UIW_BUTTON(18, 6, "Line Wrap", BTF_CHECK_BOX)
    + new UIW_BUTTON(18, 6, "Scroll", BTF_CHECK_BOX)
    + new UIW_BUTTON(18, 7, "Answerback", BTF_CHECK_BOX))

// Add the bitmapped buttons.
+ &(*new UIW_GROUP(29, 2, 13, 7, "Drives:")
    + new UIW_BUTTON(38, 4, "A:", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
        WOF_BORDER | WOF_JUSTIFY_RIGHT, NULL, 0, softDrive)
    + new UIW_BUTTON(38, 5, "B:", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
        WOF_BORDER | WOF_JUSTIFY_RIGHT, NULL, 0, softDrive)
    + new UIW_BUTTON(38, 6, "C:", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
        WOF_BORDER | WOF_JUSTIFY_RIGHT, NULL, 0, hardDrive)
    + new UIW_BUTTON(38, 7, "F:", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
        WOF_BORDER | WOF_JUSTIFY_RIGHT, NULL, 0, networkDrive)
    + new UIW_BUTTON(38, 7, "R:", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
        WOF_BORDER | WOF_JUSTIFY_RIGHT, NULL, 0, networkDrive))

// Add the regular buttons.
+ new UIW_BUTTON(10, 11, "&Save")
+ new UIW_BUTTON(20, 11, "&Cancel")
+ new UIW_BUTTON(32, 11, "&Help");
```

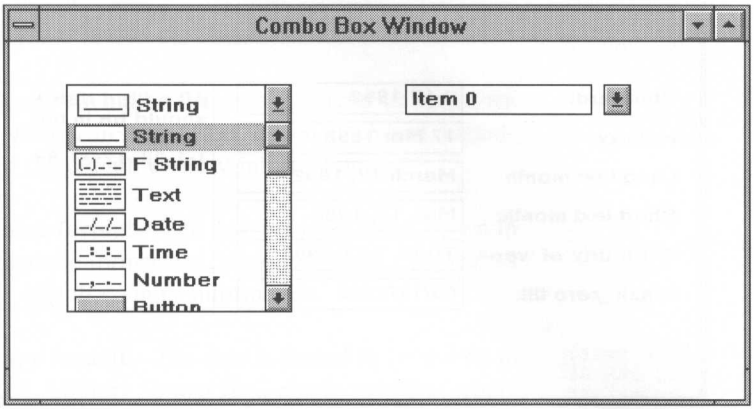
Bitmapped buttons—are buttons displayed with a graphical bitmap. The buttons function the same as the regular buttons, but they display a bitmap. Bitmapped buttons may be used in text mode, but the bitmap will not be displayed.

Check boxes—are buttons that are displayed with a 'X' when selected or ' ' when not selected.

Radio buttons—are buttons that are displayed with a '(•)' when selected or '( )' when not selected. All of the radio buttons in a window, a group, or a list box are considered to be members of the same group. Only one radio button from a particular group may be selected at any one time. **NOTE**: to have multiple radio button groups on the same window, create the group object (UIW_GROUP) and add the desired radio buttons to each group.

Combo box window objects

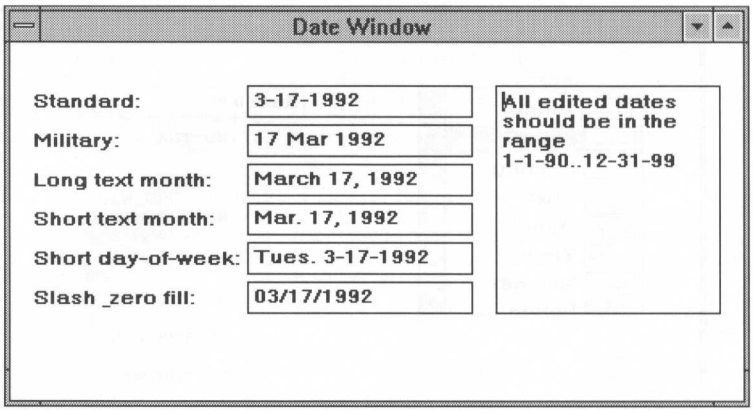
A combo box field is a one line string field with an attached button object. When the button is selected, a vertical list (described below) appears. When an item is selected, it is copied into the initial string field and the menu disappears. The figure below shows a window with two combo boxes (UIW_COMBO_BOX):




```
*window
    + new UIW_TITLE("Combo Box Window")

    + &(*new UIW_COMBO_BOX(2, 2, 11, 7)
        + new UIW_BUTTON(0, 0, 0, "String", BTF_AUTO_SIZE | BTF_NO_3D,
            WOF_BORDER, bitmap1)
        + new UIW_BUTTON(0, 0, 0, "F String", BTF_AUTO_SIZE | BTF_NO_3D,
            WOF_BORDER, bitmap2)
        + new UIW_BUTTON(0, 0, 0, "Text", BTF_AUTO_SIZE | BTF_NO_3D,
            WOF_BORDER, bitmap3)
        + new UIW_BUTTON(0, 0, 0, "Date", BTF_AUTO_SIZE | BTF_NO_3D,
            WOF_BORDER, bitmap4)
        + new UIW_BUTTON(0, 0, 0, "Time", BTF_AUTO_SIZE | BTF_NO_3D,
            WOF_BORDER, bitmap5)
        + new UIW_BUTTON(0, 0, 0, "Number", BTF_AUTO_SIZE | BTF_NO_3D,
            WOF_BORDER, bitmap6)"
        + new UIW_BUTTON(0, 0, 0, "Button", BTF_AUTO_SIZE | BTF_NO_3D,
            WOF_BORDER, bitmap7));
```

Date window objects

Date fields should be used anytime date information is presented to the end user or when date information is to be entered at an application's run-time. The figure below shows a window with several variations of the date class object (UIW_DATE):




```
UI_DATE date;
char *range = "1-1-90..12-31-99";
*window
    + new UIW_TITLE("Dates Window")
    + new UIW_TEXT(43, 1, 20, 6, "All edited dates should be in the range
        1-1-90..12-31-99", 128, WNF_NO_FLAGS,
        WOF_VIEW_ONLY | WOF_NON_SELECTABLE | WOF_BORDER)

    + new UIW_PROMPT(2, 1, "Standard...........")
    + new UIW_DATE(22, 1, 20, &date, range, DTF_SYSTEM)

    + new UIW_PROMPT(2, 2, "Military...........")
    + new UIW_DATE(22, 2, 20, &date, range,
        DTF_MILITARY_FORMAT | DTF_SYSTEM)

    + new UIW_PROMPT(2, 3, "Long text month....")
    + new UIW_DATE(22, 3, 20, &date, range, DTF_ALPHA_MONTH | DTF_SYSTEM)

    + new UIW_PROMPT(2, 4, "Short text month...")
    + new UIW_DATE(22, 4, 20, &date, range, DTF_SHORT_MONTH | DTF_SYSTEM)

    + new UIW_PROMPT(2, 5, "Short day-of-week..")
    + new UIW_DATE(22, 5, 20, &date, range, DTF_SHORT_DAY | DTF_SYSTEM)

    + new UIW_PROMPT(2, 6, "Slash & zero fill..")
    + new UIW_DATE(22, 6, 20, &date, range, DTF_SLASH | DTF_ZERO_FILL |
        DTF_SYSTEM);
```

By default, date class objects are presented and edited in a country-independent fashion. Default information, however, can be overridden by the following special date presentation and edit styles:

Long month—The month shown as an ascii string value so that the entire name of the month is displayed.

```
March 28, 1990
December 4, 1980
January 3, 2003
```

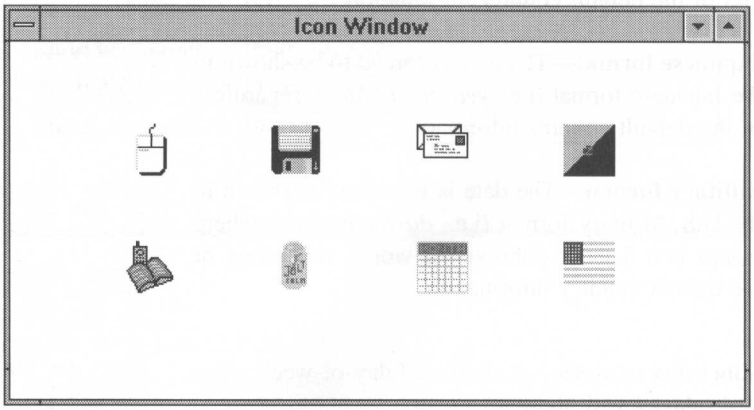
Dash—Each date variable is separated with a dash, regardless of the default country date separator.

```
3-28-1990
12-04-1980
1-3-2003
```

Day of week—The day-of-week is shown as an ascii string value before the date.	Monday May 4, 1992 Friday Dec. 5, 1980 Sunday Jan. 4, 2003
European format—The date is forced to be shown in the European format (i.e., *day/month/year*), regardless of the default country information.	28/3/1990 4 December, 1980 3 Jan., 2003
Japanese format—The date is forced to be shown in the Japanese format (i.e., *year/month/day*), regardless of the default country information.	1990/3/28 1980 December 4 2003 Jan. 3
Military format—The date is forced to be shown in the U.S. Military format (i.e., *day/month/year* where *month* is a 3 letter abbreviated word), regardless of the default country information.	(army style) 28 Mar 1900 03 Jan 2003 (navy style) 28 MAR 1900 03 JAN 2003
Short day of week—A shortened day-of-week value is displayed with the date.	Mon. May 4, 1992 Fri. Dec. 5, 1980 Sun. January 4, 2003
Short month—A shortened alphanumeric month value is shown with the date.	Mar. 28, 1990 Dec. 4, 1980 Jan. 3, 2003
Short year—The year is forced to be shown as a two-digit value.	3/28/90 December 4, '80 Jan. 3, '89
Slash—Each date value is separated with a slash, regardless of the default country date separator.	3/28/90 12/04/1900 1/3/2003
Upper-case—The date is displayed in upper-case lettering.	MARCH 28, 1990 DEC. 4, 1980 SATURDAY JAN 3, 2003
U.S. format—The date is forced to be formatted in the U.S. format (i.e., *month/day/year*), regardless of the default country information.	March 28, 1990 12/4/1980 Jan 3, 2003
Zero fill—The year, month, and day values are forced to be zero filled when their values are less than 10.	March 08, 1990 12/04/1980 01/03/2003

Icons

Icons are selectable graphic images that can be attached to a window or directly to the screen display (if the icon is a minimized window). The figure below shows a window with several icons (UIW_ICON):

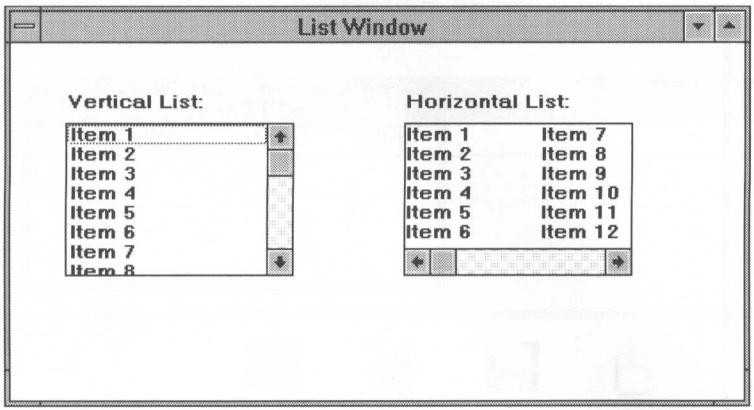



```
*window
    + new UIW_BORDER
    + new UIW_SYSTEM_BUTTON
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_TITLE("Icon Window")
    + new UIW_ICON(15, 4, "mouse")
    + new UIW_ICON(32, 4, "disk")
    + new UIW_ICON(49, 4, "letter")
    + new UIW_ICON(66, 4, "logo")
    + new UIW_ICON(15, 8, "phonebk")
    + new UIW_ICON(32, 8, "jolt")
    + new UIW_ICON(49, 8, "calendar")
    + new UIW_ICON(66, 8, "USA");
```

Icons can be used anytime you want to present a selectable item in graphical form. The main drawback of icons is that they only have graphic implementations. However, in text mode, the icon will still be selectable and its associated text will be displayed.

List window objects

List fields are select only fields (i.e., items within the list object cannot be edited) that are used to present related information in a vertical column or a horizontal list with one or more columns. The figure below shows a window with two list fields (UIW_VT_LIST and UIW_HZ_LIST):

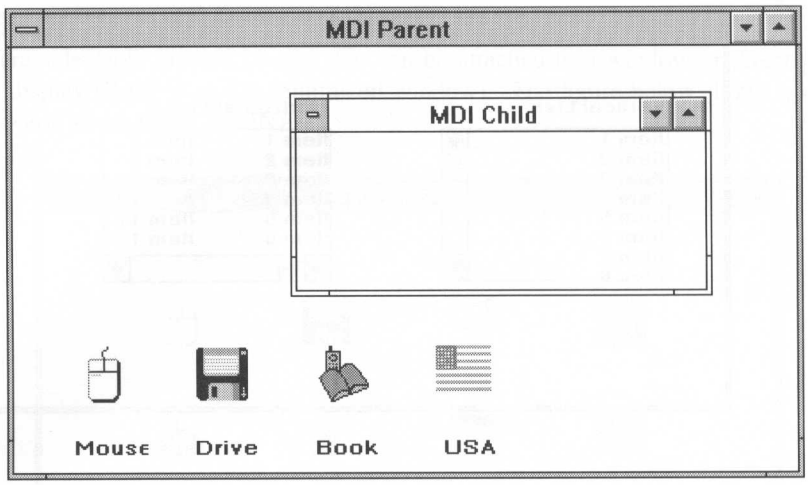



```
*window
    + new UIW_TITLE("List Window")

    + new UIW_PROMPT(2, 2, "Vertical List:")
    + &(*new UIW_VT_LIST(2, 3, 11, 6)
        + new UIW_SCROLL_BAR(0, 0, 0, SBF_VERTICAL, WOF_NON_FIELD_REGION)
        + new UIW_STRING(0, 0, 0, "Item 1", 64)
        + new UIW_STRING(0, 0, 0, "Item 2", 64)
        + new UIW_STRING(0, 0, 0, "Item 3", 64)
        + new UIW_STRING(0, 0, 0, "Item 4", 64)
        + new UIW_STRING(0, 0, 0, "Item 5", 64)
        + new UIW_STRING(0, 0, 0, "Item 6", 64)
        + new UIW_STRING(0, 0, 0, "Item 7", 64)
        + new UIW_STRING(0, 0, 0, "Item 8", 64)
        + new UIW_STRING(0, 0, 0, "Item 9", 64)
        + new UIW_STRING(0, 0, 0, "Item 10", 64));
```

In addition to the standard list fields, the list classes permit the creation of a list object that takes the complete window region (inside the border). This type of list is created whenever the WOF_NON_FIELD_REGION window flag is specified for the list object.

MDI windows

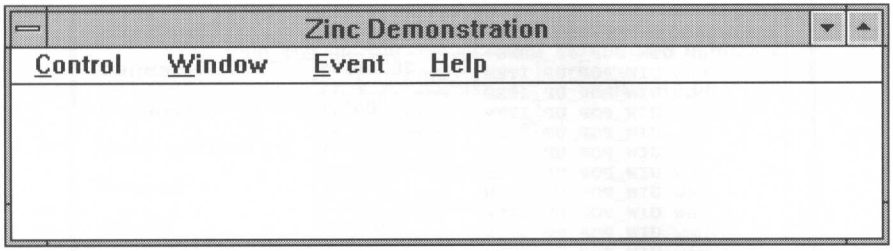
In addition to the standard use of windows (see the "Basic window objects" section of this chapter), windows may be added to other windows. These types of windows are known as MDI (multiple-document interface) windows. An MDI parent window is the controlling window that is added to the screen. MDI child windows are those sub-windows that are added to an MDI parent. The MDI child windows may be maximized, minimized, moved, or sized within the MDI parent. The restriction on MDI child windows is that they cannot move outside of their parent (i.e., they are clipped at the inside of their parent's border). The figure below shows an MDI parent window with a MDI child window and several minimized MDI child windows:

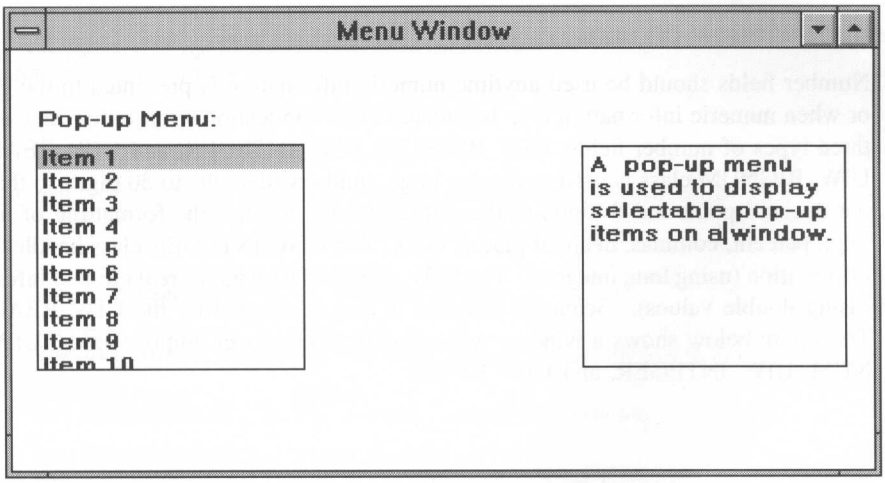



```
*window
    + UIW_WINDOW::Generic(10, 2, 15, 5, "MDI Child", WOF_NO_FLAGS,
        WOAF_MDI_OBJECT);
```

Menu window objects

Menus should be used anytime you want to present selection information to the end user. Pull-down items should be used when a hierarchal grouping of selection items is to be used. The pull-down menu serves as the first level in the selection process. The figures below show Windows implementations of a window with a pull-down menu and a window with a pop-up menu. (The pull-down menu is shown as the horizontal line with the Control, Window, Event, and Help pull-down items. The pop-up menu is shown as the vertical group of Item1-10 pop-up items.)



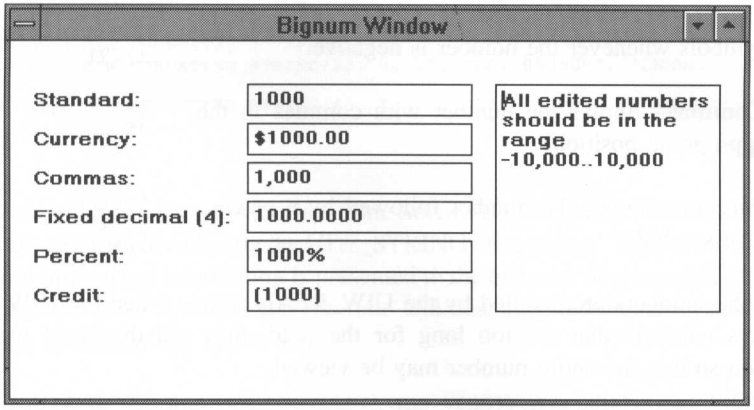



```
*window1
    + new UIW_TITLE("Zinc Demonstration")

    + &(*new UIW_PULL_DOWN_MENU(0)
        + &(*new UIW_PULL_DOWN_ITEM(" &Control ")
            + new UIW_POP_UP_ITEM("Option 1.1")
            + new UIW_POP_UP_ITEM("Option 1.2")
            + new UIW_POP_UP_ITEM("Option 1.3"))
        + &(*new UIW_PULL_DOWN_ITEM(" &Window ")
            + new UIW_POP_UP_ITEM("Option 2.1")
            + new UIW_POP_UP_ITEM("Option 2.2")
            + new UIW_POP_UP_ITEM("Option 2.3"))
        + &(*new UIW_PULL_DOWN_ITEM(" &Event ")
            + new UIW_POP_UP_ITEM("Option 3.1")
            + new UIW_POP_UP_ITEM("Option 3.2")
            + new UIW_POP_UP_ITEM("Option 3.3"))
        + &(*new UIW_PULL_DOWN_ITEM(" &Help ")
```

```
            + new UIW_POP_UP_ITEM("Option 4.1")
            + new UIW_POP_UP_ITEM("Option 4.2")
            + new UIW_POP_UP_ITEM("Option 4.3")));

*window1
    + new UIW_PROMPT(2, 1, "Pop-up menu:")
    + &(*new UIW_POP_UP_MENU(22, 1, MNF_SELECT_ONE)
        + new UIW_POP_UP_ITEM(" Option 1 ")
        + new UIW_POP_UP_ITEM(" Option 2 ")
        + new UIW_POP_UP_ITEM(" Option 3 ")
        + new UIW_POP_UP_ITEM(" Option 4 ")
        + new UIW_POP_UP_ITEM(" Option 5 ")
        + new UIW_POP_UP_ITEM(" Option 6 ")
        + new UIW_POP_UP_ITEM(" Option 7 ")
        + new UIW_POP_UP_ITEM(" Option 8 ")
        + new UIW_POP_UP_ITEM(" Option 9 ")
        + new UIW_POP_UP_ITEM(" Option 10 "))
    + new UIW_TEXT(43, 1, 20, 5,
        "A pop-up menu is used to display selectable "
        "pop-up items on a window", 128);
```

Number window objects

Number fields should be used anytime numeric information is presented to the end user or when numeric information is to be entered at an application's run-time. Zinc supports three types of number fields: UIW_BIGNUM, UIW_INTEGER, and UIW_REAL. The UIW_BIGNUM class is used to display large numbers (defaults to 30 digits to the left of the decimal point and 8 digits to the right). It also handles the formatting of numbers (e.g., percent, commas, decimal places, etc.). The UIW_INTEGER class handles integer information (using long integers). The UIW_REAL class handles real number information (using double values). Scientific notation is also performed by the UIW_REAL class. The figure below shows a window with several variations of number fields (UIW_BIGNUM, UIW_INTEGER, and UIW_REAL):




```
char *range = "0..10000";
UI_BIGNUM value = 1000;
UI_BIGNUM dvalue = 1000.0;
double sNumber = 1010000000000000;
*window
    + new UIW_TITLE("Bignum Window")

    + new UIW_TEXT(43, 1, 20, 6, "All edited numbers should be in
       the range 0..10,000", 128, WNF_NO_FLAGS,
       WOF_VIEW_ONLY | WOF_NON_SELECTABLE | WOF_BORDER)

    + new UIW_PROMPT(2, 1, "Standard: ")
    + new UIW_BIGNUM(22, 1, 20, &value, range)

    + new UIW_PROMPT(2, 2, "Currency ")
    + new UIW_BGINUM(22, 2, 20, &dvalue, range, NMF_CURRENCY | NMF_DECIMAL(2))

    + new UIW_PROMPT(2, 3, "Commas: ")
    + new UIW_BIGNUM(22, 3, 20, &value, range, NMF_COMMAS)

    + new UIW_PROMPT(2, 4, "Fixed decimal (2): ")
    + new UIW_BIGNUM(22, 4, 20, &dvalue, range, NMF_DECIMAL(2))

    + new UIW_PROMPT(2, 5, "Percent: ")
    + new UIW_BIGNUM(22, 5, 20, &value, range, NMF_PERCENT)

    + new UIW_PROMPT(2, 6, "Scientific: ")
    + new UIW_REAL(22, 6, 20, &sNumber, range);
```

The UIW_BIGNUM class object permits the following presentation and edit styles:

Decimal—Shows the number with a decimal point at a fixed location.

```
10,000.00
43.45
$149.95.
```

Currency—Shows the number with the country-specific currency symbol.

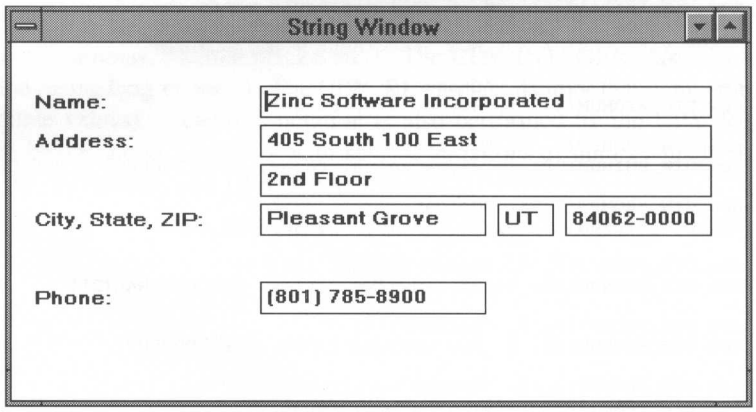
```
$10,000.00
DM100
£195
```

Credit—Shows the number with the '(' and ')' credit symbols whenever the number is negative.	(1000) (23040) (759)
Commas—Shows the number with commas in the appropriate positions.	$10,000.00 45,000 1,195
Percent—Shows the number followed by a percentage symbol.	100% 4.5% 10%

Scientific numbers are handled by the UIW_REAL class. When the UIW_REAL class displays numbers that are too long for the field, they are displayed using scientific notation so that the entire number may be viewed.

String window objects

Several types of strings are supported by Zinc Interface Library. They include single line string fields (UIW_STRING) and formatted or masked strings (UIW_FORMATTED_-STRING). The figure below shows a window containing several string window objects (UIW_STRING):




```
*window
    + new UIW_TITLE("Strings Window")

    + new UIW_PROMPT(2, 1, "String............")
    + new UIW_STRING(22, 1, 41, "Zinc Software Incorporated", 256)

    + new UIW_PROMPT(2, 2, "String............")
    + new UIW_STRING(22, 2, 41, "405 South 100 East 2nd Floor", 256)

    + new UIW_PROMPT(2, 3, "Strings............")
    + new UIW_STRING(22, 3, 20, "Pleasant Grove", 256)
```

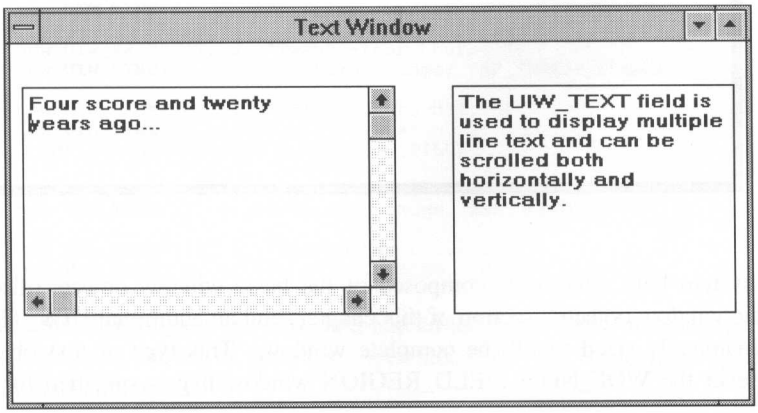
```
+ new UIW_STRING(43, 3, 4, "UT", 3)

+ new UIW_PROMPT(2, 2, "Formatted strings..")
+ new UIW_FORMATTED_STRING(22, 2, 20, "8017858900", "LNNNLLNNNLXXXX",
    "(...) ...-....")
+ new UIW_FORMATTED_STRING(43, 2, 20, "840620000", "NNNNNLNNNN",
    ".....-....")
```

The first string object, shown with the "Zinc Software Incorporated" default string in the window above, is represented by the UIW_STRING class object. This class object should be used anytime string information is presented to the end user or when string information is to be entered at an application's run-time and that information can best be presented on a single scrollable line of the screen.

The formatted string objects, shown with the "(801) 785-8900" and "84062-0000" default information in the windows above, are represented by the UIW_FORMATTED_-STRING class object. This class object should be used anytime pre-defined string format information is presented to the end user or when string information is to be entered at an application's run-time. Formatted strings restrict the type of information that an end user can enter.

Text window objects

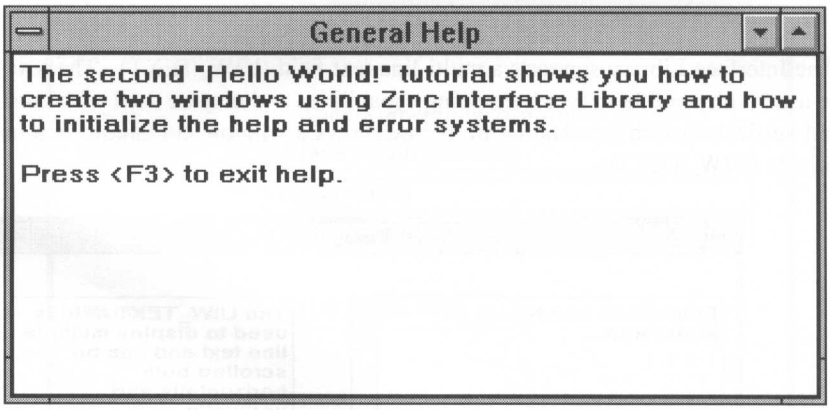
Zinc Interface Library supports a multi-line text field (UIW_TEXT). The text fields may be used with or without word-wrapping capabilities and may be used with both horizontal and vertical scroll bars. The figure below shows a window containing two text window objects (UIW_TEXT):



```
*window
    + new UIW_TITLE("Text Window")

    + &(*new UIW_TEXT(26, 2, 36, 5, "The UIW_TEXT field is used to display "
        "multiple-line text and can be scrolled both horizontally and "
        "vertically."))

    + &(*new UIW_TEXT(1, 2, 36, 5, "For score and twenty years ago...")
        + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_VERTICAL, WOF_NON_FIELD_REGION)
        + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_HORIZONTAL,
            WOF_NON_FIELD_REGION)
        + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_CORNER, WOF_NON_FIELD_REGION));
```

The text object, shown with the "The UIW_TEXT field..." default text in the window above, is represented by the UIW_TEXT class object. This class object should be used anytime text information is presented to the end user or when text information is to be entered at an application's run-time and the information can best be presented on multiple word-wrapped lines of the screen. Single-line information is best handled by the UIW_STRING class object.

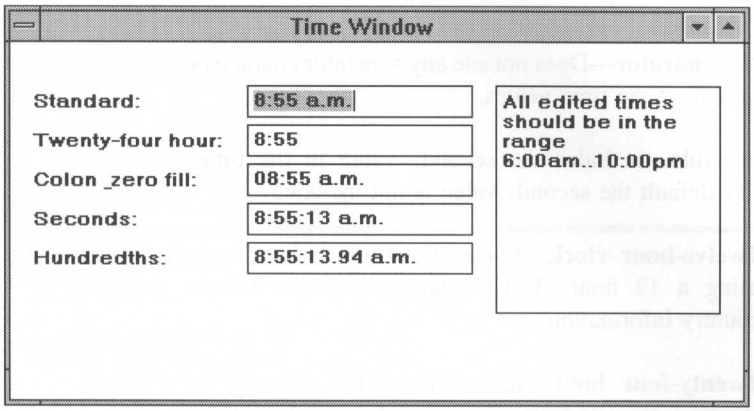
In addition to the standard text field, the UIW_TEXT class permits the creation of a text object that takes the complete window region (inside the border). For example, the graphic image below shows the help window system where the help text is shown in a text object:



The system help window is composed of the basic window objects (discussed in the "Basic window objects" section of this chapter) and an additional UIW_TEXT field that is dynamically sized to fill the complete window. This type of text object is created whenever the WOF_NON_FIELD_REGION window flag is specified for the text obje ct.

Time window objects

Time fields should be used whenever time information is presented to the end user or when time information is to be entered at an application's run-time. The figure below shows a window with several variations of a time field (UIW_TIME):




```
UI_TIME time;
char *range = "6:00am..10:00pm";
*window
    + new UIW_TITLE("Time Window")

    + new UIW_TEXT(43, 1, 20, 6,
        "All edited times should be in the range 6:00am..10:00pm",
        128, WNF_NO_FLAGS, WOF_VIEW_ONLY | WOF_NON_SELECTABLE | WOF_BORDER)

    + new UIW_PROMPT(2, 2, "Standard...........")
    + new UIW_TIME(22, 2, 20, &time, range)

    + new UIW_PROMPT(2, 3, "Twenty-four hour...")
    + new UIW_TIME(22, 3, 20, &time, range, TMF_TWENTY_FOUR_HOUR)

    + new UIW_PROMPT(2, 4, "Colon & zero fill..")
    + new UIW_TIME(22, 4, 20, &time, range,
        TMF_COLON_SEPARATOR | TMF_ZERO_FILL)

    + new UIW_PROMPT(2, 5, "Seconds............")
    + new UIW_TIME(22, 5, 20, &time, range, TMF_SECONDS)

    + new UIW_PROMPT(2, 6, "Hundredths.........")
    + new UIW_TIME(22, 6, 20, &time, range, TMF_HUNDREDTHS);
```

By default, time class objects are presented and edited in a country-independent fashion. Default information, however, can be overridden by the following special time presentation and edit styles:

Colon separator—Separates each time variable with a colon.

```
12:00
13:00:00
12:00 a.m.
```

Hundredths—Includes the hundredths value in the time. (By default the hundredths value is not included.)	`1:05:00:00`
	`23:15:05:99`
	`7:45:59:00 a.m.`

Lower-case—Shows the time in a lower-case format.	`12:00 p.m.`
	`1:00 a.m.`
	`7:00 p.m.`

No separator—Does not use any separator characters to delimit the time values.	`120`
	`130000`
	`17500`

Seconds—Includes the seconds value in the time. (By default the seconds value is not included.)	`8:09:30`
	`14:00:00`
	`3:24:59 p.m.`

Twelve-hour clock—Forces the time to be shown using a 12 hour clock, regardless of the default country information.	`12:00 a.m.`
	`1:00 p.m.`
	`5:00 p.m.`

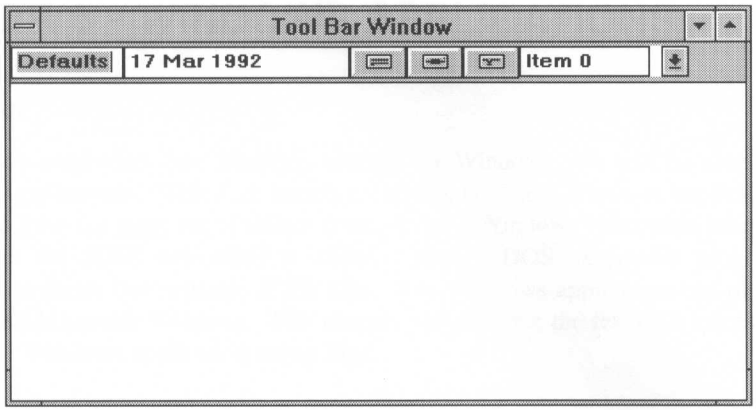
Twenty-four hour clock—Forces the time to be shown using a 24 hour clock, regardless of the default country information.	`12:00`
	`13:00`
	`17:00`

Upper-case—Shows the time in an upper-case format.	`12:00 P.M.`
	`1:00 A.M.`
	`7:00 P.M.`

Zero fill—Forces the hour, minute and second values to be zero filled when their values are less than 10.	`01:10 a.m.`
	`13:05:03`
	`01:01 p.m.`

Tool bar window objects

Tool bar objects are very similar to menus, with the exception that they may be used to display objects of various types, such as icons, buttons with bitmaps, strings, etc. The figure below shows a window with a tool bar (UIW_TOOL_BAR):




```
*window
    + new UIW_TITLE("Tool Bar Window")
    + &(*new UIW_TOOL_BAR(0, 0, 0, 0)
        + new UIW_STRING(0, 0, 0, "Defaults", 64)
        + new UIW_DATE(0, 0, 0, &date)
        + new UIW_BUTTON(38, 5, "", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER, NULL, 0, softDrive)
        + new UIW_BUTTON(38, 6, "", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER, NULL, 0, hardDrive)
        + new UIW_BUTTON(38, 7, "", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER, NULL, 0, networkDrive)
        + new UIW_COMBO_BOX(0, 0, 15, 5));
```

CHAPTER 5 - WINDOWS APPLICATIONS

Introduction

If you have purchased Zinc Interface Library for Windows you will be able to create Windows applications. With Zinc Interface Library, DOS and Windows applications may be created from the same set of source code. Since a Windows executable program (i.e., a file with the **.EXE** extension) is different than a DOS executable program, it is necessary to create two separate **.EXE** files. Any Windows application can only be run from within Microsoft Windows. This chapter will describe the few differences required to create a Windows application using Zinc.

Windows library

The Windows version of Zinc Interface Library has been compiled into a single library file called **ZILW.LIB**. When creating a Windows application, **ZILW.LIB** must be linked into the .EXE file. Be careful not to link in more than one library file (i.e., do not link the DOS library in as well.)

Compiler options

When creating a Windows application, the following compiler options should be selected:

Windows application—If your compiler is able to create applications for both DOS and Windows environments, you should select the compiler option to create the application as a Windows executable program.

Large model—Set the compiler option to compile using the large memory model. Since Zinc Interface Library is shipped only with the large memory model, all user applications must also be compiled with the large memory model.

WinMain

Ordinary C++ programs begin with calling **main( )** as the first function. However, in Windows, **WinMain( )** is the first function called. **WinMain** is used to allow Windows to begin execution of an application. Here is the definition of the **WinMain**:

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpszCmdLine, int nCmdShow);
```

In order to maintain a single set of source code, Zinc Interface Library uses compiler directives (e.g., *#ifdef*) to separate code that is specific to a particular environment. The following code shows the format of the main routines for both DOS and Windows:

```
#ifdef _WINDOWS
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpszCmdLine, int nCmdShow)
{
    .
    .
    .
#else
main(int argc, char *argv[])
{
    .
    .
    .
#endif
    .
    .
    .
}
```

Derived objects

C++ offers the powerful ability to derive classes in order to create similar, yet unique, classes. While there are no limitations regarding the derivation of Zinc classes, it should be done with caution. For example, each object contains an **Event** function that processes messages, and these messages differ between DOS and Windows. See the "Help Bar" tutorial in the *Programmer's Tutorial* for detailed information on how to create a new object that meets the specifications for both DOS and Windows.

SECTION III
ZINC DESIGNER

CHAPTER 6 - INTRODUCTION

Zinc 3.0 offers the tightest integration available between an interactive design tool and the supporting class library. Most Windows developers use a resource tool to help create their program interface. Resource tools are language and library dependent by design and therefore cannot access all the features of a given class library. This results in a fragmented approach to application development with isolated user functions and non-specific documentation. The developer is saddled with the not-so-obvious details of integrating his or her code with both the class library and the resource tool. The seamless integration of Zinc Designer and Zinc 3.0 contrasts sharply with this a la carte approach.

Zinc Designer was created with Zinc Interface Library and lets you access all of the available features in the library. Zinc Designer lets you interactively create your application screens using Zinc objects. You simply select windows and window objects from the menu or toolbar and place them on the screen.

Interactive Editors

You can easily customize objects with Zinc Designer's interactive editors. Every Zinc object that can be customized has an editor in Zinc Designer. These editors are the focal point for modifying the attributes of the objects that you place on the screen. Each editor is customized for its specific object but most of the editors have these general features:

Option Lists—A scrolling list of all general and specific option flags that are relevant to the object. The general flags are options that apply to more than one object such as borders. The specific flags are options that apply only to a given object such as the currency flag for the bignum class.

String identifiers—A field that gives the object a unique ID which you may use to access the object from a user procedure. This identifier allows you to access a given object even if it is grouped with several other objects in a window and saved as a single resource.

Default information—Many object editors allow you to enter default information and validation ranges for the object (e.g., date ranges for the date object, number ranges for the number object).

Context sensitive help—Help can be attached both at the object and window levels. If you do not attach a specific help screen to an object then the object will use the help screen that is attached to its parent window.

User Functions—A powerful feature of Zinc Designer that allows you to integrate your user functions and validation routines to specific window objects such as menus, input fields, buttons and icons. You write and compile your user functions outside of Zinc Designer and then add the name of the procedure in the object editor. Zinc automatically passes three messages from an object to an attached user function: when the object becomes current; when the object becomes non-current; and when the object is selected. You can determine which of these messages will execute the body of your user function.

Utilities

Zinc Designer includes two very useful utilities. The Image Editor allows you to create and edit bitmaps and icons for Windows and DOS Graphics modes. Bitmaps and Icons that you create with the Image Editor are assigned to objects (e.g., attach an icon that a window will minimize to, attach a bitmap to a button) through the object editors. A combo box allows you to select the desired bitmap or icon.

The Help Editor allows you to create and edit help information. Help screens that you create with the Help Editor are assigned to windows or objects through the object editors. A combo box allows you to select the appropriate help screen for the window or object.

Getting around

Most applications associate one file for each document. File operations are usually located in the File menu. Zinc Designer adds the concept of resources to this model. A given Zinc Designer file, ending with an extension of **.DAT**, may contain one or more resources. A resource is a window with its associated objects. In order to create and save resources Zinc Designer uses a Resource Menu in addition to the File menu. The Resource Menu contains the commands that allow you to create, store, edit and delete multiple resources in a single **.DAT** file.

Zinc Designer files

After creating your screens (or resources) you save them to disk in a file with a **.DAT** extension. You add these resources to your application with one line of code. The following code segment demonstrates how to load a resource called WINDOW_1, with its associated objects, from a file called **SAMPLE.DAT**.

```
// Add a window created with Zinc Designer to the window manager.
*windowManager
    + new UIW_WINDOW("SAMPLE.DAT~WINDOW_1");
```

The resources that you create with Zinc Designer are platform-independent. Resources you create with the Windows version of Zinc Designer can be opened and edited with the DOS version and vice versa.

Zinc Designer's complete access to the Zinc class library, straightforward integration of your code and platform-independent storage can dramatically enhance your productivity.

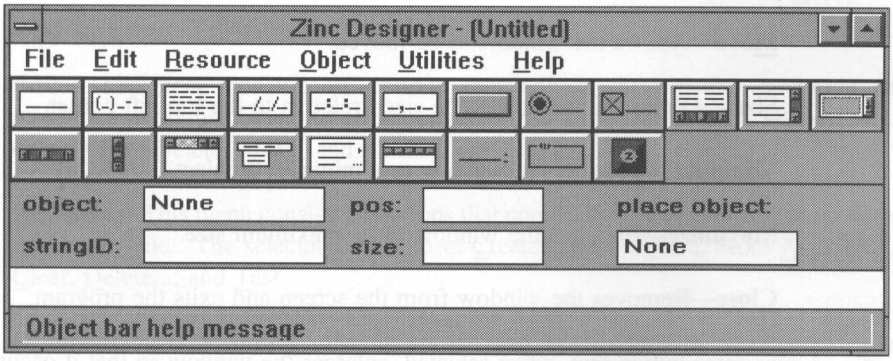
CHAPTER 7 - GETTING STARTED

Zinc Designer is an interactive tool created in order to save you, the programmer, time and effort in developing Zinc Interface Library applications. This chapter discusses the overall layout of the interactive design tool, as well as the basic procedures used in creating resources with it.

THE DESIGNER SCREEN

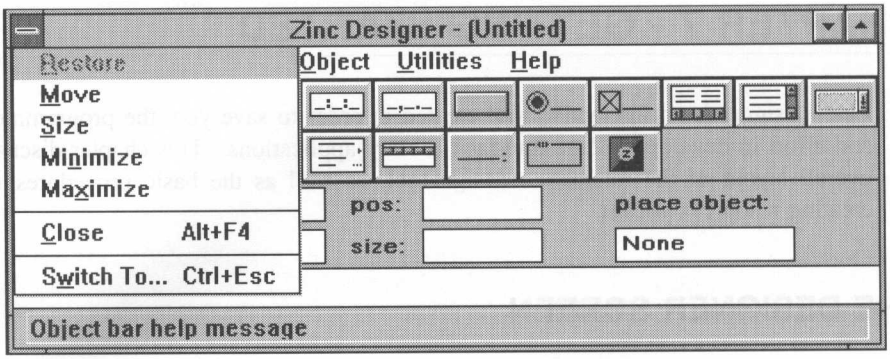
Overview

When you enter Zinc Designer, a main control window similar to the following appears:



This control window includes eight main elements with which you should be familiar:

- A title bar that identifies this window as the main control window of Zinc Designer. When a specific application is being created, the title also includes the name of the current file.
- A system button, which, when selected, displays the following pop-up menu:



Restore—Restores the window to its original size if it is in either a maximized or a minimized state.

Move—Allows the window to be moved.

Size—Allows the window to be sized relative to the top left corner.

Minimize—Reduces the window to a minimized object (i.e., icon).

Maximize—Enlarges the window to its maximum size.

Close—Removes the window from the screen and exits the program.

- A maximize button that, when selected, enlarges the window so that it occupies the entire screen. Selecting this button when the window is already in its maximized state causes the window to return to its original size.

- A minimize button that, when selected, reduces the window to its smallest representable form. Selecting this button when the window is already in its minimized state causes the window to return to its original size.

- A pull-down menu from which the main action items can be selected for interaction within the Designer. The options associated with the menu bar are described in further detail below.

- An object bar containing buttons that display various window objects. Selecting one of these buttons allows the associated object to be added to the current resource. Interaction with the object bar is described in further detail below.

- A status bar, which displays information associated with the current object. The fields associated with the status bar are described in further detail below.

- A help bar, which displays the help context associated with the current field.

The menu bar

Using the options presented as menus in the main window of Zinc Designer, applications can be created and saved for use at run-time. Selecting some menu items causes an action to take place immediately, while selecting others causes a related window to appear, from which more options are available. Menu items that cause another window to appear are distinguished by ellipses (...). A brief explanation of each menu item follows:

File—This menu consists of options that control the creation of files and exiting from the program. The selectable items on this menu are: New..., Open..., Save, Save As..., Delete..., Preferences..., and Exit.

Edit—This menu consists of options that edit or control the operation and presentation of objects within an application. The edit options are: Object..., Cut, Copy, Paste, Delete, Move, and Size.

Resource—This menu consists of options that control the creation of resources within the current file. The selectable items are: Create, Load..., Store, Store As..., Edit..., Clear, Delete..., and Test...

Object—This menu presents the objects, divided into four groups, that can be created with the Designer. The four groups presented in the first pull-down menu are: Input, Control, Menu, and Static. Selecting one of these items causes another menu to appear which contains the actual window objects of that group.

Utilities—This menu allows access to the two utility editors of Zinc Designer. The selectable options are Image Editor and Help Editor.

Help—This menu provides a list of the following selectable help contexts: Index, File, Edit, Object, Resource, Utilities, and About designer.

All of these menu items are discussed in more detail in their respective chapters that follow.

The object bar

The object bar presents some of the available window objects within Zinc Designer. It is designed to allow you to easily select these items with a mouse and then attach them directly to your current resource. When one of the objects is selected, its name appears in the "place object" field on the status bar, where it remains until it is attached to a window, or until another object is selected from the object bar. The object is attached to a resource by positioning the cursor on the desired location and clicking the mouse button.

By default the objects on the object bar are displayed by their bitmap representations, but they can also be displayed as text only or as text and bitmaps. (Refer to the Preferences section in Chapter 8 of this manual for information on how to alter the object bar defaults.)

NOTE: All of the window objects available in Zinc Designer are not represented on the object bar. For the complete set of objects, the Object option must be used.

For more information on creating and modifying window objects, refer to Chapters 11 through 15, which discuss each object in detail.

The status bar

The status bar displays the state of the current resource on the screen. The following fields are present:

object—Indicates what the current object is.

stringID—Displays the string identification of the current object.

pos—Indicates the position, in cell coordinates, of the current object. If the current object is attached to a parent window, its position is relative to that parent window.

size—Indicates the size, in cells, of the current object (i.e., width by height).

place object—Indicates the object that has been most recently selected from the object bar (or from the Object options menu) that is ready to be placed on a resource window.

HOW TO START

Once you have entered Zinc Designer, the following steps can be followed for creating a basic application:

1—Open a new file for the application by selecting File | New... Select the drive and directory to which the file is to be saved, and enter a name for the file at the "File Name" prompt. If all of the information is correct, select the "OK" button. (To move between fields without a mouse, use the <Tab> key.)

2—Create a new resource by selecting Resource | Create. A generic window will appear on the screen that can be moved and sized.

3—Attach the desired objects to the window:

a) Select the objects with the mouse directly from the object bar, or select them from the Objects options.

b) Position the cursor in the window at the desired location and press the left mouse button.

4—Edit the objects:

a) Call the editor by double clicking on the object itself, or double click on the resource window and then select the object from the "Objects" field.

b) Change the default information by positioning the cursor on a field, press the left mouse button, and enter the new information. Flags are toggled by clicking on the associated check box. (Refer to Chapters 11 through 15 for specific information on the capabilities of each object.) When all of the necessary information is entered, select the "OK" button.

5—Save the current resource by selecting Resource | Store As... If you want to name the resource something other than the default "RESOURCE 1" enter a name for it at the "StringID" prompt. Select the "OK" button to close the window and store the resource.

6—Save the current file by selecting File | Save.

7—Test the resource by selecting Resource | Test... and interacting with the objects. When you are done testing it, select the "Exit Test" button.

8—Create the help contexts to be associated with the resource window and its fields:

a) Select Utilities | Help Editor.

b) Select Context | New and enter a name for the context name at the "Context Name" prompt. Select the "OK" button.

c) Enter the title to be displayed on the help window's title bar.

d) Move the cursor to the "message" field and enter the text to be displayed in the help window.

e) Save the context by selecting Context | Save.

f) Repeat steps 'b' through 'e' for each context to be created.

g) Close the help editor by selecting Context | Exit.

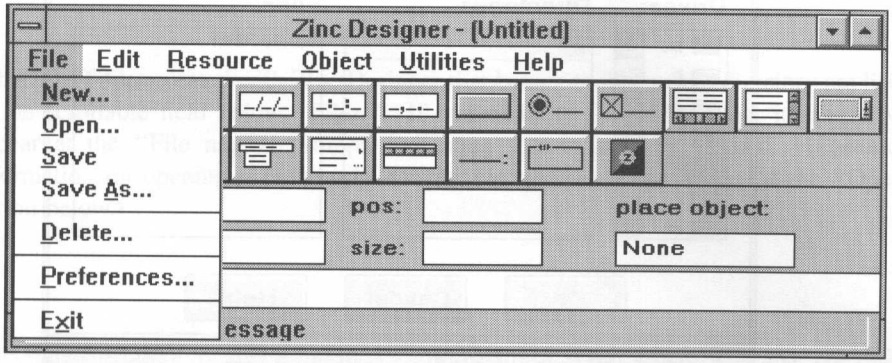
h) Call the editor for each object and select the help context to be associated with it from the "helpContext" combo box list. Select the editor's "OK" button.

9—Repeat steps 5 and 6 to save the new information to the resource and the file.

10—To add other resources to the current file, repeat steps 2 through 9.

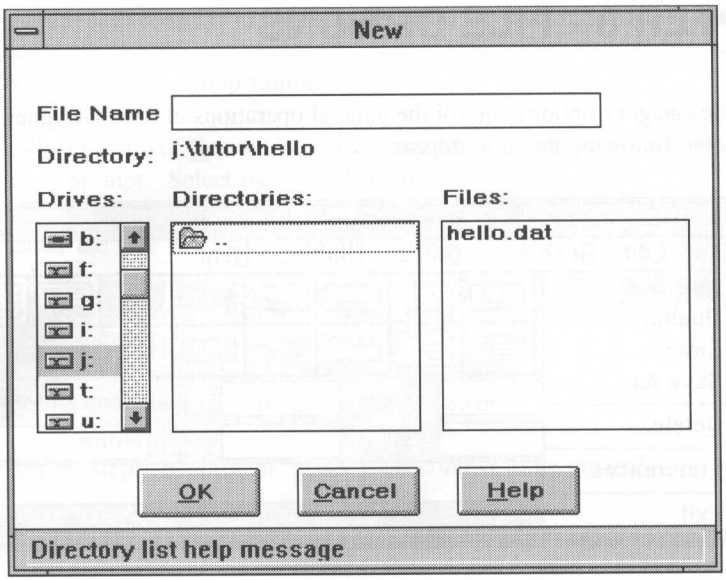
CHAPTER 8 - FILE OPTIONS

The File category options control the general operations of Zinc Designer. Selecting File causes the following menu to appear:



NEW

The "New..." option allows you to create a new file. Selecting it causes a window similar to the following to appear:



File name

If you want to open a new file for an application, enter the name for the new file here. If you do not include it yourself, a "**.DAT**" postfix will be automatically attached to the name when the file is actually created.

Directory

The current directory is shown at the "Directory" prompt. Your file will be saved to this directory. Since this item is not selectable, if you want to make a different directory the current one, it must be done by selecting a new directory from the Directories menu (discussed below).

Drives

This field displays other drives that are available on your system. Selecting a drive causes the files and directories on that drive to be displayed in their respective fields.

Directories

This field displays other available directories of the current drive. ".." represents the parent directory, and, if selected, will display the other sub-directories of the current path, all of which are also selectable.

Files

Other **.DAT** files created with Zinc Designer that belong to the current directory are listed in the scrollable field below "Files." If one of these files is selected, its name will appear at the "File name" prompt, indicating that it is to be opened. (For more information on opening a previously created file, see the explanation for the "Open" option below.)

OK

Selecting this button causes a file to be created which will be given the name entered at the "File name" prompt. If creation of the file is successful, the "New" window will close and the title bar of the control window will be updated to include the name of the current file. If no information has been entered within the "New" window and the "OK" button is selected, you will receive an error message.

Cancel

Selecting this button causes the window to close without executing any changes.

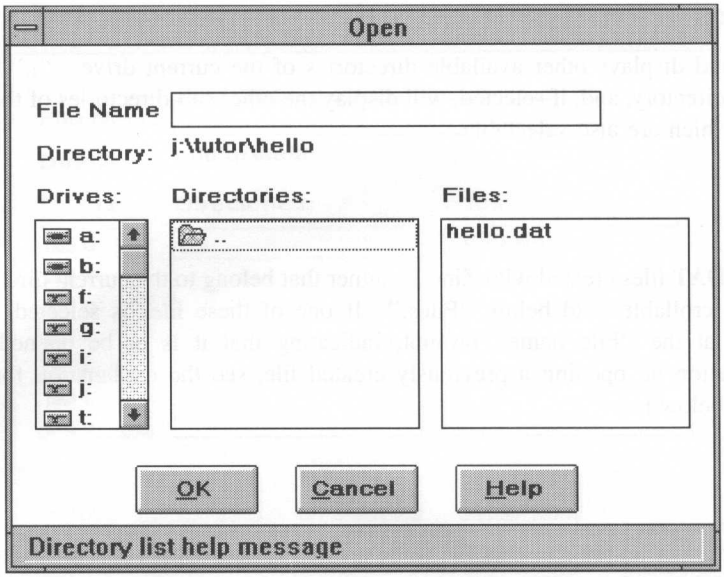
Help

Additional information about creating new files appears when this button is selected.

The help bar at the bottom of the window displays instructions for interaction with the fields of the "New" window.

OPEN

The "Open..." option allows you to open a previously created file. Selecting it causes a window to appear that is similar to the "New" window:



To open an existing file, you can enter the name at the "File name" prompt, or you can select it from the "Files" field, and the name of the file will automatically appear at the prompt.

Directory

The current directory is shown at the "Directory" prompt. Since this item is not selectable, if you want to make a different directory the current one, it must be done by selecting a new directory from the "Directories" menu (discussed below).

Drives

This field displays other drives that are available on your system.

Directories

This field displays other available directories. ".." represents the parent directory, and, if selected, will display the other sub-directories of the current path, all of which are also selectable.

Files

Other **.DAT** files created with Zinc Designer that belong to the current directory are listed in the scrollable field below Files. If one of these files is selected, its name will appear at the File name prompt.

OK

Selecting this button causes the file specified at the "File name" prompt to be opened. If the open procedure is successful, the window will close and the title bar of the control window will be updated to include the name of the current file. If the file entered at the "File name" prompt does not exist, you will receive an error message at this time. If no information has been entered within the "Open" window, you will receive an error message.

Cancel

Selecting this button causes the window to close without executing any changes.

Help

Additional information about opening existing files appears when this button is selected.

The help bar at the bottom of the window displays instructions for interaction with the fields of the "Open" window.

SAVE

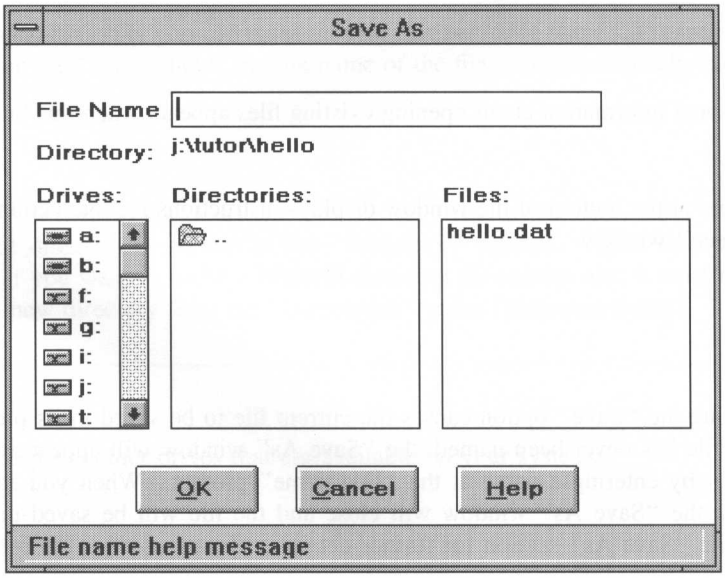
Selecting the "Save" option causes the current file to be saved in its present condition. If the file has never been named, the "Save As" window will appear and allow you to name it by entering a name at the "File name" prompt. When you select the "OK" button, the "Save As" window will close and the file will be saved under that name. (See the "Save As" section for further details on how to save a file for the first time.)

Upon every save operation, Zinc Designer automatically creates three files for the application:

- a "**.DAT**" file, which contains the binary information associated with the objects saved in the application
- a "**.CPP**" file, which contains the definition for *_objectTable*, an array that provides the function read access points for objects saved to disk
- an "**.HPP**" file, which contains the numeric identifications (entered as StringID's) unique to each field
- one or more "**.BK#**" (i.e., backup) file, depending on your choice entered in File | Preferences. (Backup files are created only if a previous **.DAT** file existed.)

SAVE AS

"Save As..." is usually used to either save a file that has not been previously named or to save the current file under another name. Selecting it causes a window to appear that is similar to the "New" and "Open" windows:



Enter a name for the file at the "File name" prompt, or select it from the "Files" field, and the name of the file will automatically appear at the prompt. If you do not include it yourself when entering the name at the prompt, a "**.DAT**" postfix will be automatically attached when the file is actually created. A new file will be created under that name that includes the latest changes to the current application.

Directory

The current directory is shown at the Directory prompt. Since this item is not selectable, if you want to make a different directory the current one, it must be done by selecting a new directory from the Directories menu (discussed below).

Drives

This field displays other drives that are available on your system.

Directories

This field displays other available directories. ".." represents the parent directory, and, if selected, will display the other sub-directories of the current path, all of which are also selectable.

Files

Other **.DAT** files created with Zinc Designer that belong to the current directory are listed in the scrollable field below Files. If one of these files is selected, its name will appear at the File name prompt, and the current application can be saved to the specified file when the "OK" button is selected.

OK

Selecting this button causes the file to be saved under the name entered at the "File name" prompt. If the save operation is successful, the "Save As" window closes.

If you have entered a file name that already exists, a modal window will appear, indicating such. If you select the "Yes" button of this window, the current information replaces the previous information of that file, and both the modal window and the "Save As" windows close. Selecting the "No" button simply closes the modal window and allows you to enter information again in the "Save As" window.

If no information has been entered within the "Save As" window and you select the "OK" button, the window will close and no other action will take place.

Cancel

Selecting this button causes the window to close without executing any changes.

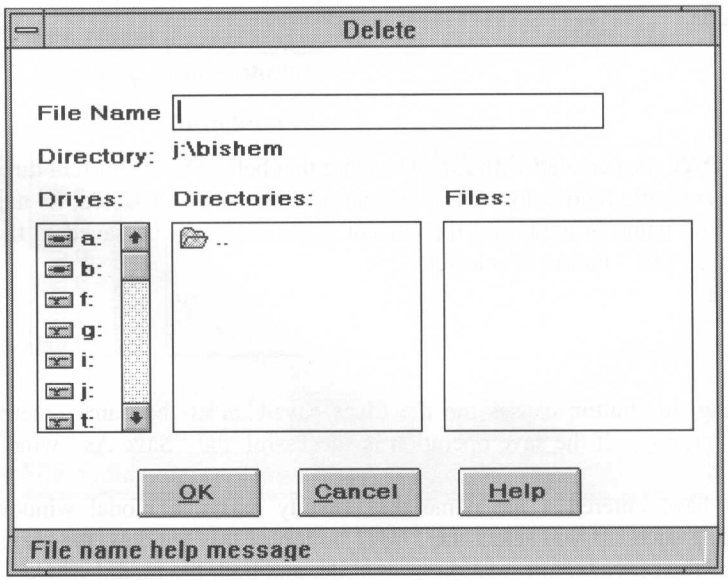
Help

Additional information about saving files appears when this button is selected.

The help bar at the bottom of the window displays instructions for interaction with the fields of the "Save As" window.

DELETE

The "Delete..." option allows you to delete a file. Selecting it causes a window similar to the following to appear:



File name

To delete a file, you can enter the name at the "File name" prompt, or you can select it from the "Files" field, and the name of the file will automatically appear at the prompt.

Directory

The current directory is shown at the Directory prompt. Since this item is not selectable, if you want to make a different directory the current one, it must be done by selecting a new directory from the Directories menu (discussed below).

Drives

This field displays other drives that are available on your system. Selecting one of the drives causes the directories and files on that drive to be displayed in their respective fields.

Directories

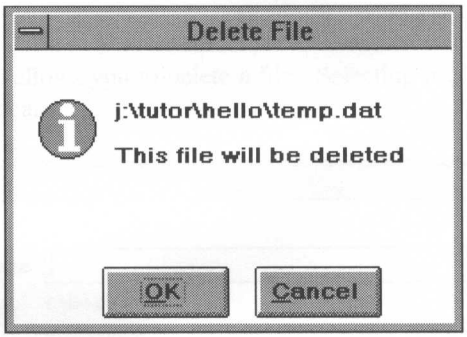
This field displays other available directories. ".." represents the parent directory, and, if selected, will display the other sub-directories of the current path, all of which are also selectable.

Files

Other files created with Zinc Designer that belong to the current directory are listed in the scrollable field below Files. If one of these files is selected, its name will appear at the File name prompt.

OK

Selecting this button causes a modal window to appear which is similar to the following:



The purpose of this window is to make sure that you want to delete the file. If you select the "OK" button, the file indicated at the "File name" prompt is deleted, and both the modal window and the "Delete" window close. If you choose the "Cancel" button, the file is not deleted and just the modal window closes.

If the name of the current file is entered, or if the file entered does not exist, you will receive an error message when the "OK" button is selected.

If no information has been entered within the window, selecting "OK" causes an error message to appear.

Cancel

Selecting this button causes the window to close without executing any changes.

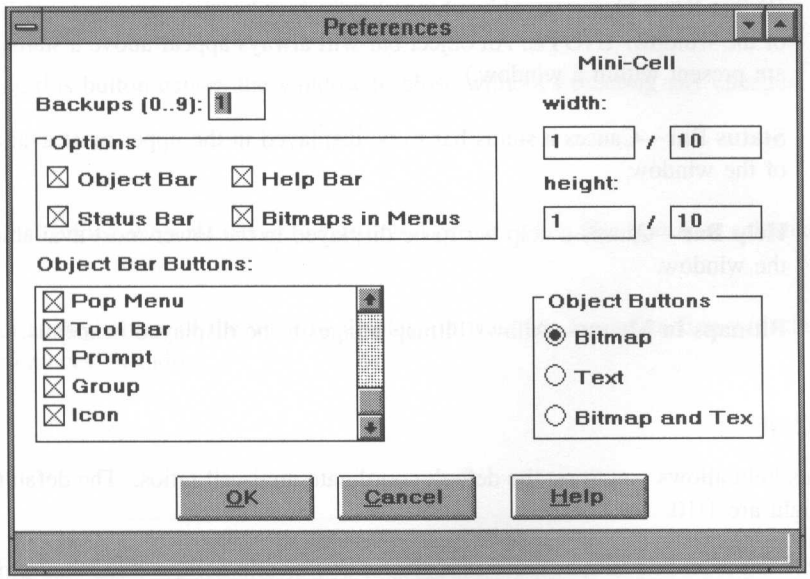
Help

Additional information about deleting files appears when this button is selected.

The help bar at the bottom of the window displays instructions for interaction with the fields of the "Delete" window.

PREFERENCES

The "Preferences..." option allows you to change the default settings of Zinc Designer. Selecting it causes a window similar to the one below to appear:



Backups

Enter in this field the number of backups that you would like the designer to make each

time a save operation is performed. Each backup file will be saved under the same name as the main file but with a postfix that indicates the backup number of the copy. For example, a file with the name of **TEST.DAT** will have a backup copy called **TEST.BK1** if only "1" is entered at the prompt. If any number greater than "1" is entered at the prompt, each time a save operation occurs another backup file will be created, up to the maximum specified. For example, a "3" at the prompt will cause the creation of a **TEST.BK1** file at the first save operation, a **TEST.BK2** file at the second save, and a **TEST.BK3** at the third save. Thereafter, these three backup files would be updated on subsequent saves, with the most recent information being saved in **TEST.BK1** and the oldest information in **TEST.BK3**.

Options

This field presents the options for what can be displayed in Zinc Designer's windows. Each is presented as a check box that toggles, and any number can be selected at one time. The options available are:

Object Bar—Causes an object bar to be displayed in the upper-most available region of the window. (**NOTE**: An object bar will always appear above a status bar if both are present within a window.)

Status Bar—Causes a status bar to be displayed in the upper-most available region of the window.

Help Bar—Causes a help bar to be displayed in the lower-most available region of the window.

Bitmaps in Menus—Allows bitmap images to be displayed in menus.

Mini-Cell

This field allows you to set the default coordinate mini-cell ratios. The default width and height are 1/10.

Object Bar Buttons

This field contains the objects that can be included in an object bar. Selecting one causes it to be represented on the tool bar in the format specified by "Object Buttons" (i.e., bitmap and/or text). Each is presented as a check box that toggles, and any number can be selected at one time.

Object Buttons

This field contains the options for the presentation of object buttons.

Bitmap—Allows only bitmap images to be displayed on an object button.

Text—Allows only text to be displayed on an object button.

Bitmap and Text—Allows both bitmaps and text to be displayed on an object button.

OK

Selecting this button closes the "Preferences" window and causes the information selected to take effect. If no information has been entered within the window, it will close and no other action will take place.

Cancel

Selecting this button causes the window to close without executing any changes.

Help

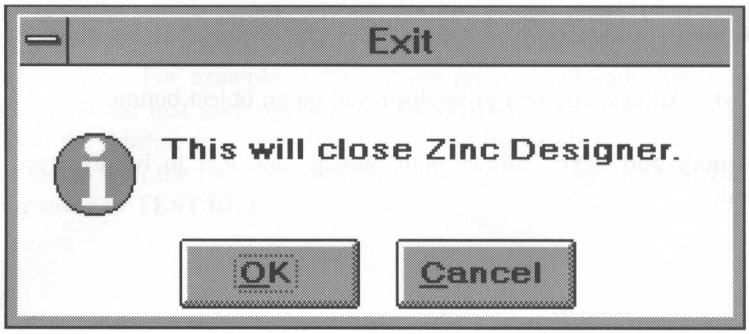
Additional information about default settings appears when this button is selected.

The help bar at the bottom of the window displays instructions for interaction with the fields of the "Preferences" window.

EXIT

Selecting the "Exit" option allows you to exit Zinc Designer. If you have not saved the current file, a modal window will appear that asks whether or not you want to save it before exiting. Selecting the "Yes" button causes the file to be saved and then exits out of the program. Selecting "No" causes the program to exit without saving the current file (i.e., any changes made since the last save operation will be lost). Selecting the "Cancel" button simply closes the modal window.

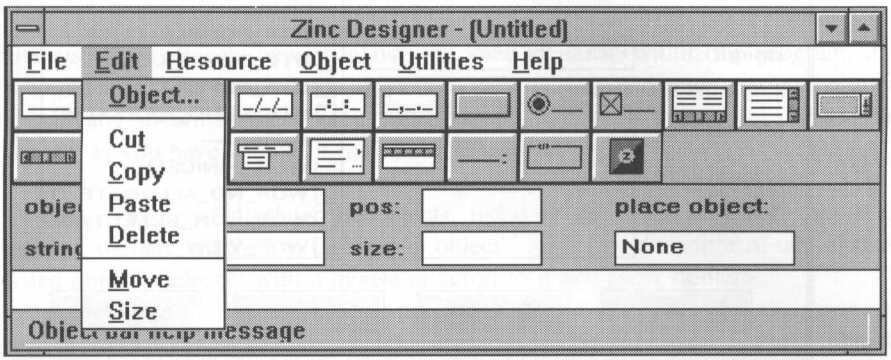
If you have not made any changes within Zinc Designer, selecting "Exit" causes a modal window to appear which is similar to the following:



The purpose of this window is to make sure that you want to exit Zinc Designer. If you select the "OK" button, the program exits. If you choose the "Cancel" button, the program does not exit and the modal window closes.

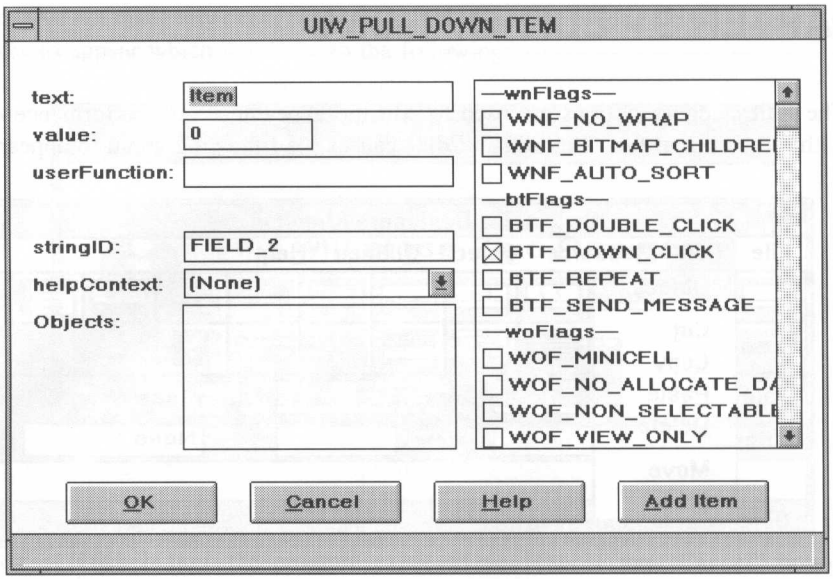
CHAPTER 9 - EDIT OPTIONS

The Edit category options are used to edit the appearance and performance of objects within the current file. Selecting "Edit" causes the following menu to appear:



OBJECT

Each object created with Zinc Designer can be modified through interaction with its object editor. Selecting "Object..." causes the editor for the current object to appear, which is similar to the following:



The object editor controls the general presentation of the object. Since each object has its own specific requirements, the fields of each editor will vary, but all contain one or more of the following fields:

text—This field allows you to enter information to be displayed within the object exactly as you want it to appear in your application. Objects that use the "text" field are: string, text, button, radio button, check box, pull-down item, pop-up item, prompt, and group. Some objects have a field similar to "text," but they use the name of the object in place of "text." These objects are: date, time, bignum, integer, and real.

userFunction and **compareFunction**—If you want to have a user function or a compare function associated with the object, you can enter the name of it in this field. The function must be defined somewhere in your code under the same name that is entered so that Zinc Designer can find it and execute the designated action. (For more information on creating user functions and compare functions, refer to the description of the object's constructor in the *Programmer's Reference*.)

stringID—This field contains the string identification for the object and is present in every object editor (except for horizontal and vertical scroll bars). The default string identification for a resource window is "RESOURCE" plus a unique number corresponding with the order in which it was created. For example, the screen

identification for the first resource window created on the screen would be "RESOURCE_1." The default string identification for an object attached to another object is "FIELD" plus a unique number corresponding with the order in which it was attached to the parent resource. The number given to the first object is actually 0, so, for example, the screen identification for the second object created within a resource window would be "FIELD_1."

Because these objects appear in lists in other locations within the program, it is recommended that you override the default identification and enter a string that more specifically identifies the object. The identification will appear in all locations exactly as you have entered it in the object's editor.

objects—This field displays the objects, listed in the order in which they were created, that are attached to the current object. To access the editor of one of these listed objects, select it with a mouse or scroll to it and press <Enter>.

options and flags field—This field is located on the right side of every object's editor, and it displays flags or options which control the general presentation and operation of the current object. All of these items are listed with check boxes, which display a 'X' when they are currently in effect. To toggle a flag or option from non-current to current or vice versa, select it by either clicking on it with the mouse or by scrolling to it and pressing <Enter>. There is no limit to the number of flags that can be in effect at a given time; however, if two flags are selected that present conflicting information, such as "Center Justify" and "Right Justify," only the flag listed first in the field will have effect.

Other fields that are more specific to individual objects are discussed in chapters 7 through 10.

Each object editor also includes three buttons, which operate in the following manner:

OK—Selecting this button saves the edit information and closes the object editor window. The current object will reflect the editing changes immediately. If no information has been entered within the object editor, its window will close with no other action taking place.

Cancel—Selecting this button causes the window to close without executing any changes.

Help—Additional information about the current object appears when this button is selected.

A help bar is also included in each object editor that displays help on how to interact with the edit window's fields.

CUT

Selecting the Cut option removes the current object from the screen and places it in a global paste buffer.

COPY

Selecting the Copy option copies the current object and places the copy in a global paste buffer.

PASTE

Selecting "Paste" allows you to recall and position on the screen the contents of the global paste buffer (placed there by Cut or Copy procedures). After selecting "Paste," position the cross hair cursor (+) where you would like the paste to occur and press the left mouse button.

DELETE

Selecting "Delete" removes the current object from the screen and deletes it from the file.

MOVE

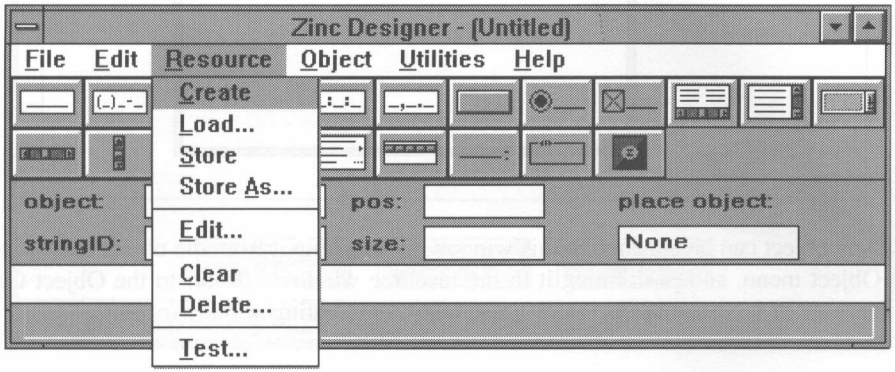
Selecting "Move" allows you to move the current object either by dragging the mouse or by using the arrow keys.

SIZE

Selecting "Size" allows you to size the selected region from the bottom right corner either by dragging the mouse or by using the arrow keys.

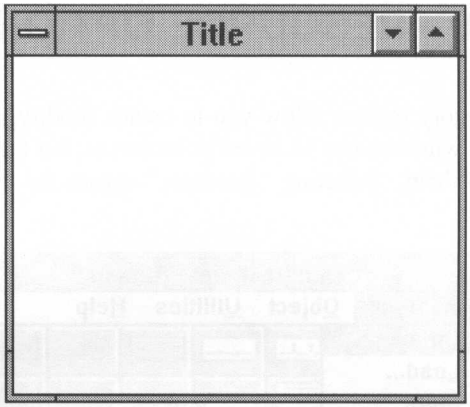
CHAPTER 10 - RESOURCE OPTIONS

The Resource category options allow you to create, modify, and retrieve objects in the current file. Only windows can be saved as resources, but they can have any number of objects attached to them. Selecting "Resource" causes the following menu to appear:



CREATE

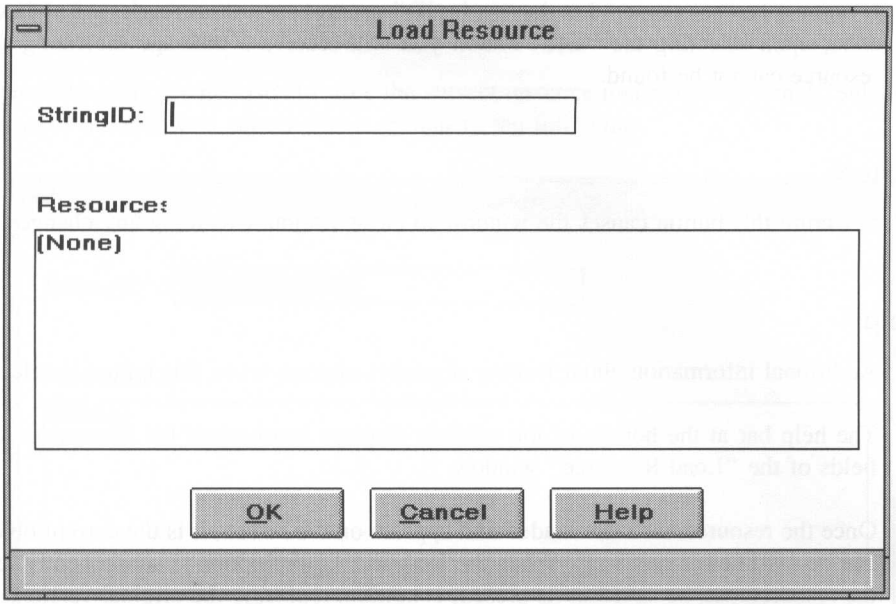
Selecting "Create" automatically places the following window on the screen, complete with a title bar, a system button, and minimize and maximize buttons.



Any object can be attached to this window by selecting it from the object bar, or from the Object menu, and positioning it in the resource window. (Refer to the Object Category chapter of this manual for more information on creating window objects.)

LOAD

"Load..." is used to recall a previously created resource from the current file. Selecting it causes a window similar to the following to appear:



StringID

Enter the string identification of the resource to be loaded, or by selecting it from the "Resources" field, the stringID will automatically be displayed at the "StringID" prompt.

Resources

This field displays the resources that are available in the current file. The resources are listed by their stringID's in alphabetical order. If one of these is selected, its string identification will appear at the "StringID" prompt.

OK

Selecting this button causes the resource designated at the "StringID" prompt to be loaded. If the load operation is successful, the "Load Resource" window closes and the resource window, containing its child objects (if any), appears on the screen in the exact location and condition it was last stored.

If nothing has been entered at the "StringID" prompt, or if the stringID entered does not exist, upon selecting the "OK" button you will receive a message indicating that the resource cannot be found.

Cancel

Selecting this button causes the window to close without executing any changes.

Help

Additional information about loading resources appears when this button is selected.

The help bar at the bottom of the window displays instructions for interaction with the fields of the "Load Resource" window.

Once the resource has been loaded and appears on the screen, it is the current object and can be modified in any way. When the Resource | Store option is subsequently selected, the resource will be saved in its present condition, replacing the original version. (Refer to the Store and Store As sections of this chapter for more information on storing resources.)

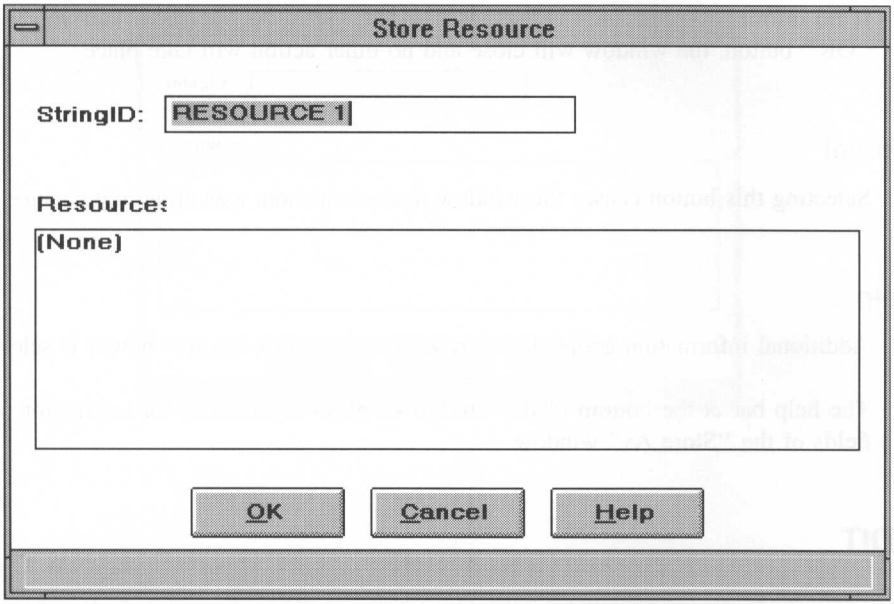
STORE

Selecting the "Store" option causes the current resource to be saved in its present condition to the current file. The name given the resource will be the string identification which appears at the "StringID" prompt of both the resource window's editor and on the control window's status bar. If you have not entered a different name for the resource in its editor or through a "Store As" operation, the stringID given it will be "RESOURCE" plus a unique number corresponding with the order in which it was created. For example, the screen identification for the first resource created in a file would be "RESOURCE_1."

Each time a store operation is performed, the previous contents of the resource are completely replaced by the current information.

STORE AS

"Store As..." is usually used to store the current resource under another name. Selecting it causes a window to appear that is similar to the following:



StringID

Enter a name for the resource at the "StringID" prompt, or, if you want to replace a previously created resource with the current information, select one from the "Resources" field, and the string identification for that resource will automatically appear at the prompt.

Resources

This field displays the resources that are available in the current file. The resources are listed by their stringID's in alphabetical order. If one of these is selected, its string identification will appear at the "StringID" prompt and the current application will replace the previous contents of that resource when the "OK" button is selected.

OK

Selecting this button causes the resource to be stored under the identification entered at the "StringID" prompt. If the save operation is successful, the "Store As" window closes.

If no information has been entered within the "Store As" window and you select the "OK" button, the window will close and no other action will take place.

Cancel

Selecting this button causes the window to close without executing any changes.

Help

Additional information about storing resources appears when this button is selected.

The help bar at the bottom of the window displays instructions for interaction with the fields of the "Store As" window.

EDIT

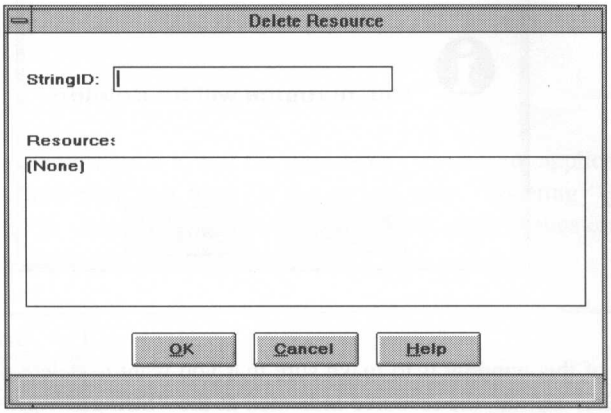
Each object created with Zinc Designer can be modified through interaction with its object editor. Selecting "Edit..." causes the editor of the current object to appear. The object editor controls the general presentation of the object. It can also be called by selecting Edit | Object while the object is current or by clicking twice on an object. (Refer to Chapter 6 of this manual for further information on object editors.)

CLEAR

Selecting "Clear" causes the current resource to be removed from the screen. It does not, however, delete the resource from the file. If you have not stored the current resource immediately before, selecting "Clear" causes a modal window to appear that asks if you want to store it before clearing it from the screen. Selecting "Yes" causes it to be stored and then cleared, selecting "No" causes it to be cleared without storing it first, and selecting "Cancel" simply closes the modal window and the resource is neither stored nor cleared.

DELETE

The "Delete..." option allows you to delete a resource from the current file. Selecting it causes a window similar to the following to appear:



StringID

Enter the string identification of the resource to be deleted, or by selecting it from the "Resources" field, the stringID will automatically be displayed at the prompt.

Resources

This field displays the resources that are available in the current file. The resources are listed by their stringID's in alphabetical order. If one of these is selected, its string identification will appear at the "StringID" prompt.

OK

Selecting this button causes a modal window to appear which is similar to the following:

The purpose of this window is to make sure that you want to delete the resource. If you select the "OK" button, the resource indicated at the "StringID" prompt is deleted from the current file, and both the top modal window and the "Delete Resource" window close. If you choose the "Cancel" button, the resource is not deleted and just the top window closes.

If the name of the current file is entered, or if the file entered does not exist, you will receive an error message when the "OK" button is selected.

If no information has been entered within the window, selecting "OK" causes an error message to appear.

If the delete operation is successful, the "Delete Resource" window closes and the resource window, including its child objects (if any), is removed from the screen and is deleted from the current file.

Cancel

Selecting this button causes the window to close without executing any changes.

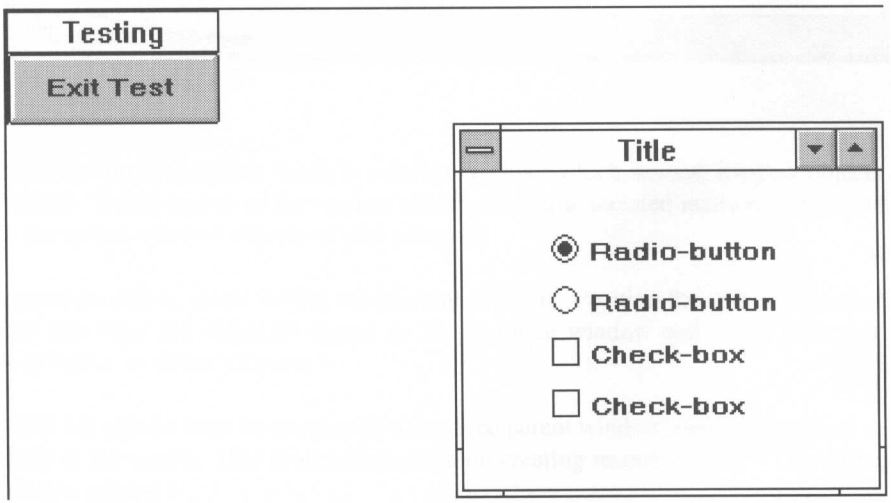
Help

Additional information about deleting resources appears when this button is selected.

The help bar at the bottom of the window displays instructions for interaction with the fields of the "Delete Resource" window.

TEST

The "Test" option allows you to test the objects of your current application resource so that you can see how they will function for an end user. Selecting "Test" causes the control window to be cleared from the screen and moves your application into test mode, which looks something like the following:

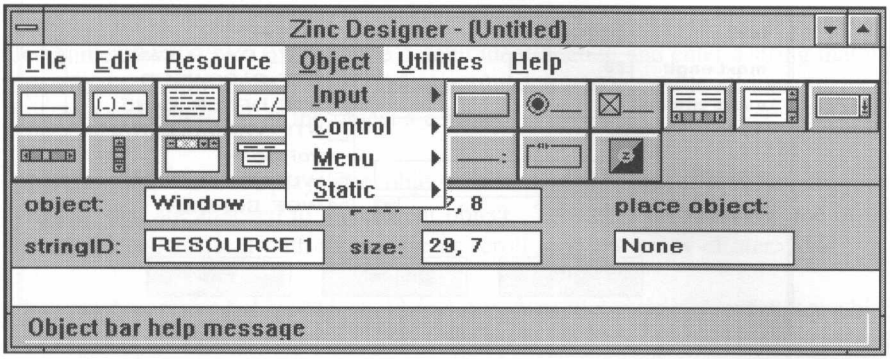


In test mode the objects of your application will look and act as they will for an end user. For example, check box and radio buttons will actually toggle and scroll bars will actually scroll information. No objects can be created or modified while in test mode.

When you have finished testing the resource, select the "Exit Test" button and the screen will return to normal mode. The control window will be displayed again, and you will be able to modify your application in any manner.

CHAPTER 11 - OBJECT OPTIONS

The Object category provides options that allow you to actually create objects. Selecting "Object" causes the following menu to appear:



Each of the options on this menu is a category under which several window objects are classified. Selecting one of the options causes another associated menu to appear, which lists the actual window objects of that category.

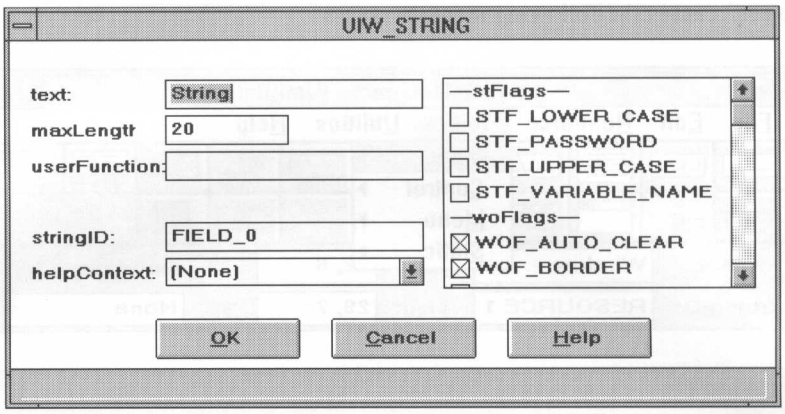
To create an object, select it from the associated menu. Position the cross hair cursor (+) where you want the object to appear on the resource window and either press the left mouse button or press <Enter>.

NOTE: All objects must be attached to a resource parent window; they cannot be attached directly to the screen. (For more information on creating resource windows, see Chapter 10 of this manual.)

The editor of each of these objects can be accessed by any of the following methods:

- Select Edit | Object while the object is current
- Select Resource | Edit while the parent resource window is current; then select the desired object from the edit window's list of objects
- Press <Enter> while the object is current
- Click twice on the object with the mouse

Each editor varies according to the specific object, but the general format of all editors is similar to the following:



The object editor controls the general presentation of the object. Since each object has its own specific requirements, the fields of each editor will vary, but all contain one or more of the following fields:

text—This field allows you to enter information to be displayed within the object exactly as you want it to appear in your application. Objects that use the "text" field are: string, text, button, radio button, check box, pull-down item, pop-up item, prompt, and group. Some objects have a field similar to "text," but they use the name of the object in place of "text." These objects are: date, time, bignum, integer, and real.

userFunction and **compareFunction**—If you want to have a user function or a compare function associated with the object, you can enter the name of it in this field. The function must be defined somewhere in your code under the same name that is entered so that Zinc Designer can find it and execute the designated action. (For more information on creating user functions and compare functions, refer to the description of the object's constructor in the *Programmer's Reference*.)

stringID—This field contains the string identification for the object and is present in every object (editor except for horizontal and vertical scroll bars). The default string identification for a resource window is "RESOURCE" plus a unique number corresponding to the order in which it was created. For example, the screen identification for the first resource window created on the screen would be

"RESOURCE_1." The default string identification for an object attached to another object is "FIELD" plus a unique number corresponding to the order in which it was attached to the parent resource. The number given to the first object is actually 0, so, for example, the screen identification for the second object created within a resource window would be "FIELD_1."

Because these objects appear in lists in other locations within the program, it is recommended that you override the default identification and enter a string that more specifically identifies the object. The identification will appear in all location exactly as you have entered it in the object's editor.

objects—This field displays the objects, listed in the order in which they were created, that are attached to the current object. To access the editor of one of these listed objects, select it with the mouse or scroll to it and press <Enter>.

flags and options field—This field is located on the right side of every object's editor, and it displays flags or options which control the general presentation and operation of the current object. All of these items are listed with check boxes, which display a 'X' when they are currently in effect. To toggle a flag or option from non-current to current or vice versa, select it by either clicking on it with the mouse or by scrolling to it and pressing <Enter>. There is no limit to the number of flags that can be in effect at a given time; however, if two flags are selected that present conflicting information, such as "Center Justify" and "Right Justify," only the flag listed first in the field will have effect.

Other fields that are more specific to individual objects are discussed in chapters 7 through 10.

Each object editor also includes three buttons, which operate in the following manner:

OK—Selecting this button saves the edit information and closes the object editor window. The current object will reflect the editing changes immediately. If no information has been entered within the object editor, its window will close with no other action taking place.

Cancel—Selecting this button causes the window to close without executing any changes.

Help—Additional information about the current object appears when this button is selected.

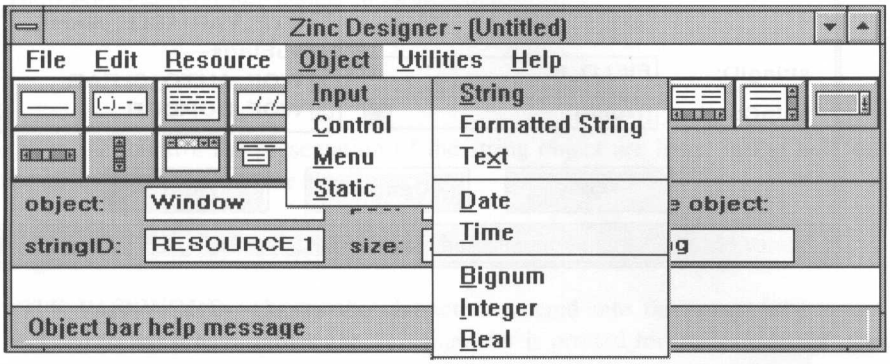
A help bar is also included in each object editor that displays help on how to interact with the edit window's fields.

To test how an object will actually appear and function for the end user, try it in test mode, which is accessed by selection Resource | Test while the parent resource is active. (For more information on test mode, refer to the Test section in Chapter 5 of this manual.)

A description of each window object, grouped according to its category type, is documented in the following four chapters. For more specific information on how these objects are created, refer to the respective chapters of the *Programmer's Reference*.

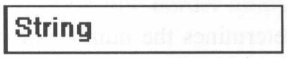
CHAPTER 12 - INPUT OBJECTS

The input category includes objects that are used specifically for date input. Selecting the "Input" option causes the following associated menu to appear:

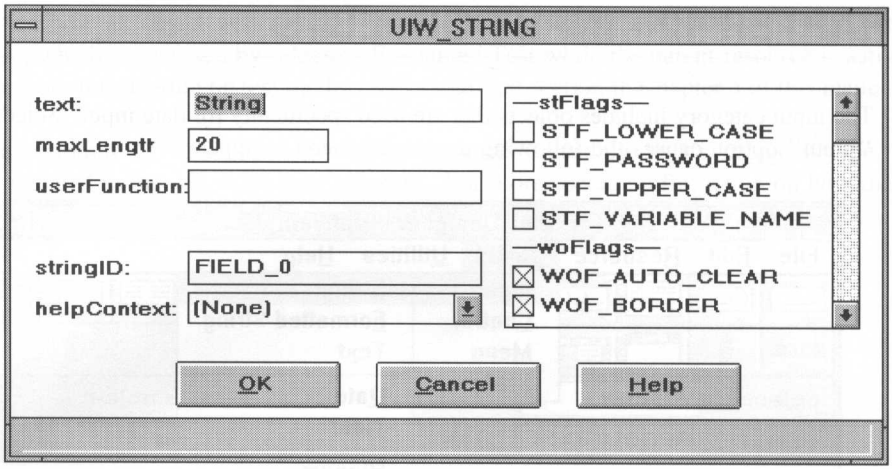


STRING

A string object is used to present and collect alphanumeric string information. Selecting "String" causes the following object to appear:



To modify the string object, call its editor by double clicking the mouse on the object. The following window will appear:



text

Enter text in this field exactly as you want it to appear in the string object. If it contains more characters than the "maxLength" limitation allows, only the number of characters that fall within the limit will be displayed. If the string object is not long enough to display all of the entered text, it can be sized using the mouse or the arrow keys.

maxLength

The number in this field determines the number of characters that the string object will display. The default length is 20. The maximum length is 32,767.

userFunction

If you want to have a user function associated with the string object, enter in this field the name of the function. This user function must be defined somewhere in your code so that Zinc Designer can retrieve it.

stringID

Enter in this field a string that will distinguish the string object from other objects.

helpContext

This field designates the help context to be associated with the string. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the string and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

flags

The flags that control the presentation of the string object are listed in the field on the right half of the window. The flags are:

STF_LOWER_CASE—Converts all character input to lowercase values.

STF_PASSWORD—Causes the characters entered into the string field to not be echoed to the screen; rather, the "." character is printed for each character typed.

STF_UPPER_CASE—Converts all character input to uppercase values.

STF_VARIABLE_NAME—Converts the space character to an underscore value.

WOF_AUTO_CLEAR—Automatically clears the string buffer if the end user positions on the first character of the string field (from another window field) then presses a key (without first having pressed any movement or editing keys).

WOF_BORDER—Draws a single line border around the object in graphics mode. (In text mode, no border is drawn.)

WOF_INVALID—Sets the initial status of the string field to be "invalid." By default, all string information is valid. A programmer may specify a string field as invalid by setting this flag upon creation of the string object or by re-setting the flag through the user function (discussed above). For example, a string field may initially be set to be blank, but the final string edited by the end user must contain some instructional information. In this case the initial string information does not fulfill the programmer's requirements.

WOF_JUSTIFY_CENTER—Center-justifies the string information within the string field.

WOF_JUSTIFY_RIGHT—Right-justifies the string information within the string field.

WOF_MINICELL—Uses mini-cell values to determine the mini-cell heights. Initially, a mini-cell is set to 1/10 of a cell coordinate. This flag allows for more precise positioning of objects and is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the string to not be a form field. If this flag is set the string will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the string object from being selectable. If this flag is set, the end user will not be able to edit, or position on, the string information.

WOF_UNANSWERED—Sets the initial status of the string field to be "unanswered." An unanswered string field is displayed as blank space on the screen.

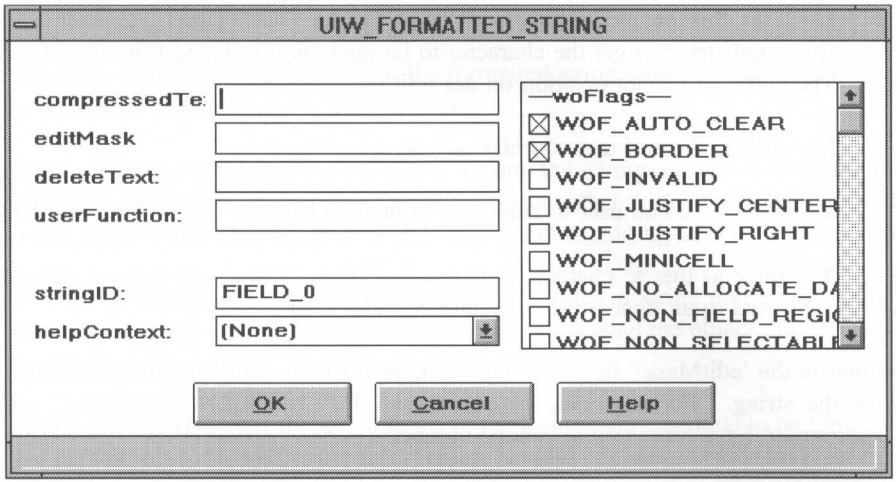
WOF_VIEW_ONLY—Prevents the string from being edited. If this flag is set, the end user will not be able to edit the string information but will be able to browse through the string.

WOAF_NON_CURRENT—The string cannot be made current. If this flag is set, users will not be able to select the string from the keyboard nor with the mouse.

FORMATTED STRING

A formatted string object is used to display and collect information that requires a specific format. For example, telephone numbers and zip codes are best presented as formatted strings. Selecting "Formatted String" causes the following object to appear:

To modify the formatted string object, call its editor. The following window appears:



compressedText

Enter text in this field as you want it to initially appear in the formatted string object. It must conform to the specifications set by the "editMask" and "deleteText" fields. For example, a string "8017858900" would be appropriate for a formatted telephone number.

editMask

This field determines the type of characters that the formatted string will accept. The following characters can be used to define the edit mask:

a—Allows the end user to enter a space (' ') or any letter (i.e., 'a' through 'z' or 'A' through 'Z').

A—Same as the 'a' character option except that a lower-case letter is automatically converted to an upper-case letter.

c—Allows the end user to enter a space (' '), a number (i.e., '0' through '9'), or any alphabetic character (i.e., 'a' through 'z' or 'A' through 'Z').

C—Same as the 'c' character option except that a lower-case character is automatically converted to upper-case.

L—Uses this position as a literal place holder. Using this character causes the formatted string to get the character to be read and displayed from the literal mask. The end user cannot position on nor edit this character.

N—Allows the end user to enter any digit.

x—Allows the end user to enter any printable character (i.e., ' ' through '~').

X—Same as the 'x' character option except that a lower-case letter is automatically converted to an upper-case alphanumeric character.

Enter in the 'editMask" field a string of characters that will define the acceptable format for the string. For example, an edit mask of "LNNNLLNNNLNNNN" would be appropriate for a formatted telephone number.

deleteText

Enter into this field a string of literal characters that will be used whenever a character is deleted from a particular position in the formatted string. For example, a string of "(...) ...-...." would be appropriate for a formatted telephone number.

userFunction

If you want to have a user function associated with the formatted string object, enter in this field the name of the function. This user function must be defined somewhere in your code so that Zinc Designer can retrieve it.

stringID

Enter in this field a string that will distinguish the formatted string object from other objects.

helpContext

This field designates the help context to be associated with the formatted string. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the formatted string and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

flags

The flags that control the presentation of the formatted string object are listed in the field on the right half of the window. The flags are:

WOF_AUTO_CLEAR—Automatically clears the string buffer if the end user positions on the first character of the field (from another window field) then presses a key (without first having pressed any movement or editing keys).

WOF_BORDER—Draws a border around the formatted string object. In graphics mode, setting this option draws a single line border around the object. In text mode, no border is drawn.

WOF_INVALID—Sets the initial status of the formatted string field to be "invalid." By default, all formatted string information is valid. A programmer may specify a formatted string field as invalid by setting this flag upon creation of the string object or by re-setting the flag through the user function (discussed above). For example, a formatted string field for a phone number may initially be set to (000) 000-0000, but the final string edited by the end user must contain some valid phone number. In this case the initial string information does not fulfill the programmer's requirements.

WOF_JUSTIFY_CENTER—Center-justifies the text information within the formatted string field.

WOF_JUSTIFY_RIGHT—Right-justifies the text information within the formatted string field.

WOF_MINICELL—Uses mini-cell values to determine the mini-cell heights. Initially, a mini-cell is set to 1/10 of a cell coordinate. This flag allows for more precise positioning of objects and is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the formatted string to not be a form field. If this flag is set the formatted string will occupy all the remaining space of its parent window.

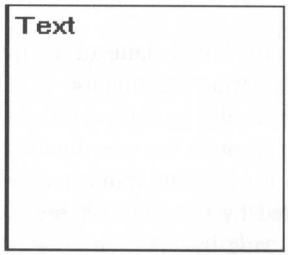
WOF_NON_SELECTABLE—Prevents the formatted string object from being selected. If this flag is set, the end user will not be able to edit or position on the formatted string information.

WOF_UNANSWERED—Sets the initial status of the formatted string field to be "unanswered." An unanswered formatted string field is displayed as blank space on the screen.

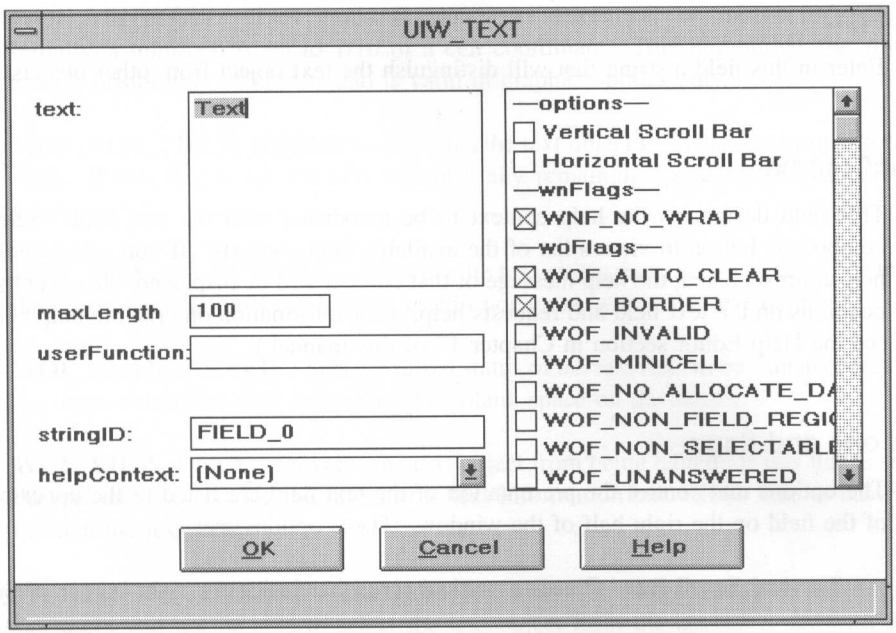
WOAF_NON_CURRENT—The formatted string object cannot be made current. If this flag is set, users will not be able to select the formatted string from the keyboard or with the mouse.

TEXT

A text object is used to present and collect alphanumeric textual information in a multi-line format. Selecting "Text" causes the following box to appear:



To modify the text object, call its editor. The following window will appear:



text

Enter text in this field exactly as you want it to appear in the text object. If it contains more characters than the "maxLength" limitation allows, only the number of characters that fall within the limit will be displayed. If the text object is not long enough to display all of the entered text, it can be sized using the mouse or the arrow keys.

maxLength

The number in this field determines the number of characters that the text object will display. The default length is 100. The maximum length is 32,767.

userFunction

If you want to have a user function associated with the text object, enter in this field the name of the function. This user function must be defined somewhere in your code so that the user table created by Zinc Designer can accesses it.

stringID

Enter in this field a string that will distinguish the text object from other objects.

helpContext

This field designates the help context to be associated with the text field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the text field and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

options and flags

The options that control the presentation of the text field are listed in the upper portion of the field on the right half of the window. These options are:

Vertical Scroll Bar—Places a vertical scroll bar inside the right border of the text field.

Horizontal Scroll Bar—Places a horizontal scroll bar inside the bottom border of the text field.

The flags that control the presentation of the text object are listed in the field in the lower portion of the field on the right half of the window. The flags are:

WNF_NO_WRAP—Disables the default word wrap in the text field.

WOF_AUTO_CLEAR—Automatically clears the text buffer if the end user positions on the first character of the text field (from another window field) and then presses a key (without having previously pressed any movement or editing keys).

WOF_BORDER—Draws a single line border around the text object. In text mode, no border is drawn.

WOF_INVALID—Sets the initial status of the text field to be "invalid." By default, all text information is valid. For example, a text field may initially be set to be blank, but the final text field edited by the end user must contain some instructional text. In this case the initial text information does not fulfill the programmer's requirements.

WOF_MINICELL—Uses mini-cell values to determine the mini-cell heights. Initially, a mini-cell is set to 1/10 of a cell coordinate. This flag allows for more precise positioning of objects and is valid in graphics modes only.

WOF_NON_FIELD_REGION—Prevents the text object from being a normal form field. If this flag is set the text occupies any remaining space within the parent window.

WOF_NON_SELECTABLE—Prevents the text object from being selected. If this flag is set, the user will not be able to edit nor move within the text field.

WOF_UNANSWERED—Sets the initial status of the text field to be "unanswered." An unanswered text field is displayed as blank space on the screen.

WOF_VIEW_ONLY—Prevents the text object from being edited. If this flag is set, the end user will not be able to edit the text information but will be able to browse through the text field.

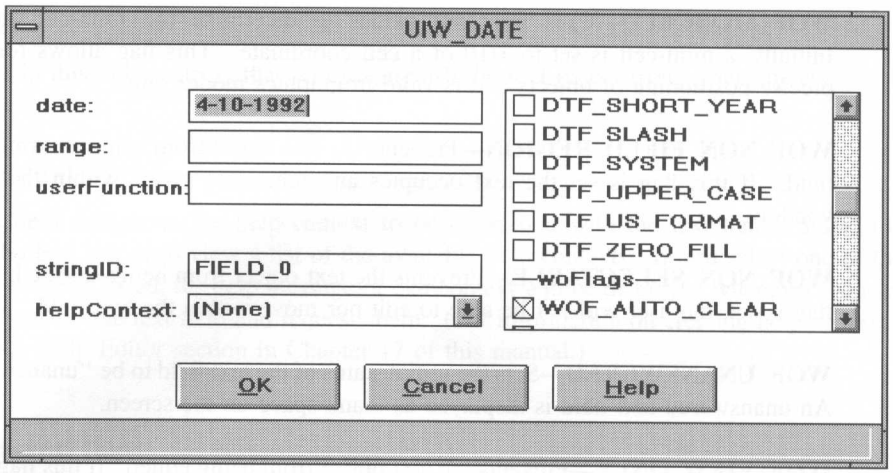
WOAF_NON_CURRENT—The text object cannot be made current. If this flag is set, users will not be able to select the text object from the keyboard nor with the mouse.

DATE

A date field displays and collects date information. Selecting "Date" causes a date field to appear that contains the current date, similar to the figure below:

4-10-1992

To modify the date, call its editor. The following window will appear:



date

Enter in this field the date that you want to appear in the date object. The default format to which this date will be automatically converted is *month-day-year*, with spaces being automatically converted to hyphens (-). (If another flag is set that designates a different separator for the date, such as DTF_SLASH, spaces will be converted accordingly.)

range

If you want to specify a certain range of acceptable dates, enter in this field the valid date ranges. For example, if you want to accept only those dates within the 1992 calendar year, enter the range of "1-1-92..12-31-92." If no range is entered, any date will be accepted.

userFunction

If you want to have a user function associated with the date object, enter in this field the name of the function. This user function must be defined somewhere in your code so that the user table created by Zinc Designer can retrieve it.

stringID

Enter in this field a string that will distinguish the date object from other objects.

helpContext

This field designates the help context to be associated with the date field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the date and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

flags

The flags that control interpretation and presentation of the date object are listed in the field on the right side of the window. These flags are:

DTF_ALPHA_MONTH—Formats the month to be displayed as an ASCII string value.

DTF_DASH—Separates each date variable with a dash, regardless of the default country date separator.

DTF_DAY_OF_WEEK—Adds an ASCII string day-of-week value to the date.

DTF_EUROPEAN_FORMAT—Forces the date to be displayed and interpreted in the European format (i.e., *day/month/year*), regardless of the default country information.

DTF_JAPANESE_FORMAT—Forces the date to be displayed and interpreted in the Japanese format (i.e., *year/month/day*), regardless of the default country information.

DTF_MILITARY_FORMAT—Forces the date to be displayed and interpreted in the U.S. Military format (i.e., *day/month/year* where *month* is a 3 letter abbreviated word), regardless of the default country information.

DTF_SHORT_DAY—Adds a shortened day-of-week to the date.

DTF_SHORT_MONTH—Uses a shortened alphanumeric month in the date.

DTF_SHORT_YEAR—Forces the year to be displayed as a two-digit value.

DTF_SLASH—Separates each date value with a slash, regardless of the default country date separator.

DTF_SYSTEM—Fills a blank date with the system date. For example, if a blank ASCII date were entered by the end user and the DTF_SYSTEM flag were set, the date would be set to the system date.

DTF_UPPER_CASE—Converts the alphanumeric date to upper-case.

DTF_US_FORMAT—Forces the date to be displayed and interpreted in the U.S. format (i.e., *month/day/year*), regardless of the default country information.

DTF_ZERO_FILL—Forces the year, month, and day values to be zero filled when their values are less than 10.

WOF_AUTO_CLEAR—Automatically clears the date buffer if the end user positions on the first character of the date field (from another window field) then presses a key (without first having pressed any movement or editing keys).

WOF_BORDER—Draws a border around the date object. In graphics mode, setting this option draws a single line border around the object. In text mode, no border is drawn.

WOF_INVALID—Sets the initial status of the date field to be "invalid." An invalid date fits in the absolute range determined by the object type (i.e., "1-1-100..12-31-32767") but does not fulfill all the requirements specified by the program. For example, a date may initially be set to 3-12-90 but the final date, edited by the end user, must be in the range "12-1-90..12-31-90." The initial date in this example fits the absolute requirements of a date class object but does not fit into the specified range.

WOF_JUSTIFY_CENTER—Center-justifies the date information within the date field.

WOF_JUSTIFY_RIGHT—Right-justifies the date information within the date field.

WOF_MINICELL—Uses mini-cell values to determine the mini-cell heights. Initially, a mini-cell is set to 1/10 of a cell coordinate. This flag allows for more precise positioning of objects and is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the date object to not be a form field. If this flag is set the date object will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the date object from being selected. If this flag is set, the user will not be able to edit the date information.

WOF_UNANSWERED—Sets the initial status of the date field to be "unanswered." An unanswered date field is displayed as blank space on the screen.

WOF_VIEW_ONLY—Prevents the date object from being edited. If this flag is set, the end user will not be able to edit a date object's information but will be able to browse through the information.

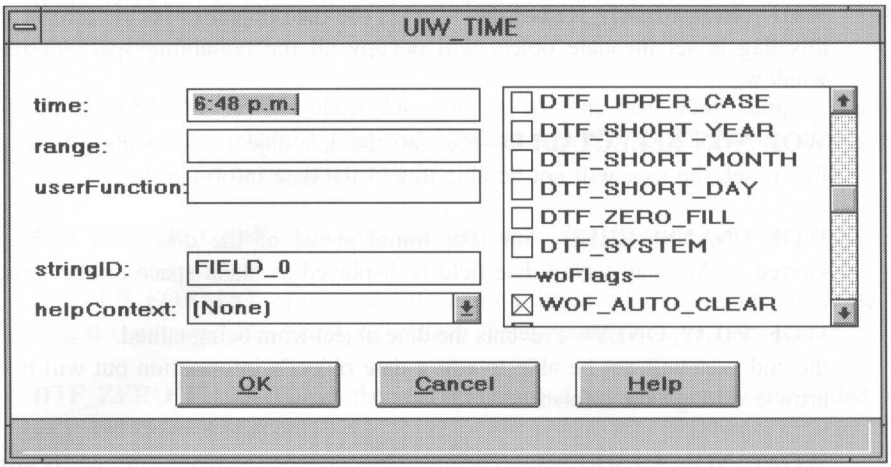
WOAF_NON_CURRENT—The date object cannot be made current. If this flag is set, users will not be able to select the date object from the keyboard nor with the mouse.

TIME

A time field displays and collects time information. Selecting "Time" causes a time field to appear that contains the current time, similar to the figure below:

10:07 p.m.

To modify the time object, call its editor. The following window will appear:



time

Enter in this field the time that you want to appear in the time object. The default format to which this time will be automatically converted is *hour:minutes a.m.* or *hour:minutes p.m.*. A space between numbers will be interpreted as a colon, and necessary periods (for "a.m." and "p.m.") are automatically inserted. Since any hour value under 12 is interpreted as morning, it is necessary to enter "pm" if the hour value is meant to be in post-meridian time and you are using a 12-hour clock. If you enter the time value according to a 24-hour clock, there is no need to enter "a.m." or "p.m."—the object will interpret and convert the value into the default format.

range

If you want to specify a certain range of acceptable time values, enter in this field the valid time ranges. For example, if you want to accept only those times whose values fall in post-meridian time, enter the range of "12:01pm..11:59:59pm." If no range is entered, any time value will be accepted.

userFunction

If you want to have a user function associated with the time object, enter in this field the name of the function. This user function must be defined somewhere in your code so that the user table created by Zinc Designer can retrieve it.

stringID

Enter in this field a string that will distinguish the time object from other objects.

helpContext

This field designates the help context to be associated with the time field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the time and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

flags

The flags that control interpretation, presentation, and operation of the time information are listed in the field on the right. These flags are:

TMF_COLON_SEPARATOR—Separates each time variable with a colon.

TMF_HUNDREDTHS—Includes the hundredths value in the time. (By default the hundredths value is not included.)

TMF_LOWER_CASE—Converts the time to lower-case.

TMF_NO_HOURS—Does not display nor interpret an hour value for the UI_TIME object.

TMF_NO_MINUTES—Does not display nor interpret a minute value for the UIW_TIME class object.

TMF_NO_SEPARATOR—Does not use any separator characters to delimit the time values.

TMF_SECONDS—Includes the seconds value in the time. (By default the seconds value is not included.)

TMF_SYSTEM—Fills a blank time with the system time. For example, if a blank ASCII time value were entered by the end user and the TMF_SYSTEM flag were set, the time would be set to the current system time.

TMF_TWELVE_HOUR—Forces the time to be displayed and interpreted using a 12 hour clock, regardless of the default country information.

TMF_TWENTY_FOUR_HOUR—Forces the time to be displayed and interpreted using a 24 hour clock, regardless of the default country information.

TMF_UPPER_CASE—Converts the time to upper-case.

TMF_ZERO_FILL—Forces the hour, minute, and second values to be zero filled when their values are less than 10.

WOF_AUTO_CLEAR—Automatically clears the time buffer if the end user positions on the first character of the time field (from another window field) and then presses a key (without first having pressed any movement or editing keys).

WOF_BORDER—Draws a border around the time object. In graphics mode, setting this option draws a single line border around the object. In text mode, no border is drawn.

WOF_INVALID—Sets the initial status of the time field to be "invalid." An invalid time fits in the absolute range determined by the object type (i.e., "12:00pm..-11:59:59pm") but does not fulfill all the requirements specified by the program. For example, a time field may initially be set to 8:15am, but the final time, edited by the end user, must be in the range "12:00pm..11:59:59pm." The initial time in this example fits the absolute requirements of a time class object but does not fit into the specified range.

WOF_JUSTIFY_CENTER—Center-justifies the time information within the time field.

WOF_JUSTIFY_RIGHT—Right-justifies the time information within the time field.

WOF_MINICELL—Uses mini-cell values to determine the mini-cell heights. Initially, a mini-cell is set to 1/10 of a cell coordinate. This flag allows for more precise positioning of objects and is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the time object to not be a form field. If this flag is set the time object will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the time object from being selectable. If this flag is set, the user will not be able to edit the time information.

WOF_UNANSWERED—Sets the initial status of the time field to be "un-answered." An unanswered time field is displayed as blank space on the screen.

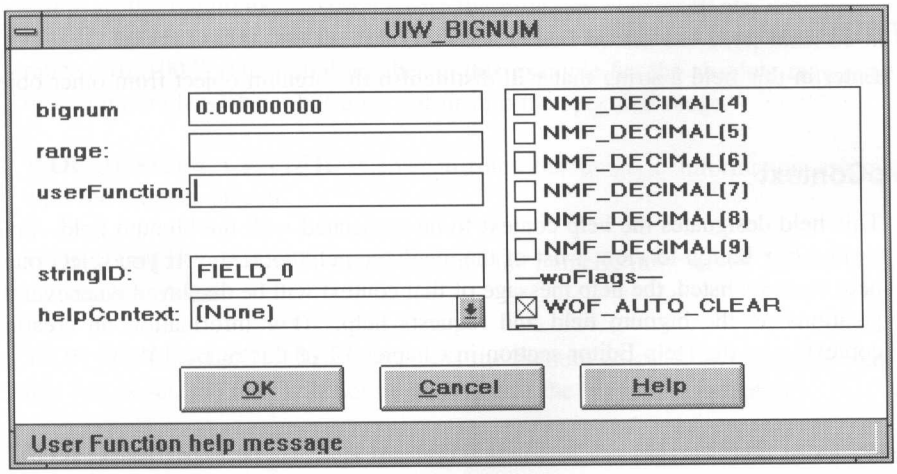
WOF_VIEW_ONLY—Prevents the time string from being modified. This flag will still allow the time field to become current.

WOAF_NON_CURRENT—The time object cannot be made current. If this flag is set, users will not be able to select the time object from the keyboard nor with the mouse.

BIGNUM

A bignum object is used to display and collect numeric information. It can be formatted in various ways, such as for numbers presented as percentages, currency, and credit. Selecting "Bignum" causes the following object to appear:

To modify the bignum object, call its editor. The following window will appear:



bignum

Enter in this field the number that you want to appear in the bignum field. The number will be displayed with the number of decimal places designated by the flags you have set. A bignum object can have up to thirty digits to the left of the decimal place and up to eight digits to the right of the decimal place.

range

If you want to specify a certain range of acceptable bignum values, enter in this field the valid ranges. For example, if you want to accept only numbers between 100 and 100,000, enter the range of "100..100000." If no range is entered, any numeric value will be accepted.

userFunction

If you want to have a user function associated with the bignum object, enter in this field the name of the function. This user function must be defined somewhere in your code so that Zinc Designer can retrieve it.

stringID

Enter in this field a string that will distinguish the bignum object from other objects.

helpContext

This field designates the help context to be associated with the bignum field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the bignum field and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

flags

The flags that control the presentation and operation of the bignum information are listed in the field on the right. These flags are:

NMF_CURRENCY—Displays the number with the country-specific currency symbol.

NMF_CREDIT—Displays the number with the '(' and ')' credit symbols whenever the number is negative.

NMF_COMMAS—Displays the number with commas.

NMF_DECIMAL(decimal)—Displays the number with a decimal point at a fixed location. *decimal* is the number of decimal places to be displayed. Valid *decimal* values range from 0 to 8.

NMF_PERCENT—Displays the number with a percentage symbol.

WOF_AUTO_CLEAR—Automatically clears the numeric buffer if the end user positions on the first character of the bignum field (from another window field) then presses a key (without first having pressed any movement or editing keys).

WOF_BORDER—Draws a border around the bignum object. In graphics mode, setting this option draws a single line border around the object. In text mode, no border is drawn. This is the default argument.

WOF_INVALID—Sets the initial status of the bignum field to be "invalid." Invalid numbers fit in the absolute range determined by the object type but do not fulfill all the requirements specified by the program. For example, a bignum may initially be set to 200, but the final number, edited by the end user, must be in the range "10..100." The initial number in this example fits the absolute requirements of a bignum class object but does not fit into the specified range.

WOF_JUSTIFY_CENTER—Center-justifies the numeric information associated with the number object.

WOF_JUSTIFY_RIGHT—Right-justifies the numeric information associated with the number object.

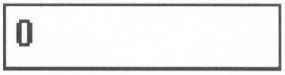
WOF_NON_SELECTABLE—Prevents the bignum object from being selected. If this flag is set, the user will not be able to edit the bignum information.

WOF_UNANSWERED—Sets the initial status of the bignum field to be "unanswered." An unanswered number field is displayed as blank space on the screen.

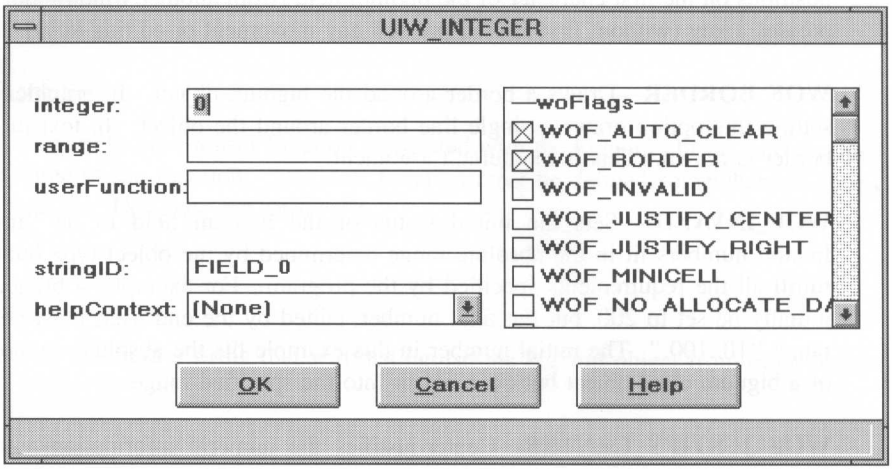
WOF_VIEW_ONLY—Prevents the bignum object from being edited. However, the bignum object may become current.

INTEGER

An integer number object is used to present and collect numeric information for integers. It cannot be formatted. (The bignum object must be used for numbers requiring special formatting capabilities.) Selecting "Integer" causes the following object to appear:



To modify the integer object, call its editor. The following window appears:



integer

Enter in this field the integer that you want to appear in the integer field.

range

If you want to specify a certain range of acceptable integer values, enter in this field the valid ranges. For example, if you want to accept only numbers between 100 and 100,000, enter the range of "100..100000." If no range is entered, any integer value will be accepted.

userFunction

If you want to have a user function associated with the integer object, enter in this field the name of the function. This user function must be defined somewhere in your code so that the user table, created by Zinc Designer, can retrieve it.

stringID

Enter in this field a string that will distinguish the integer object from other objects.

helpContext

This field designates the help context to be associated with the integer field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the integer field and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

flags

The flags that control the presentation and operation of the integer information are listed in the field on the right. These flags are:

WOF_AUTO_CLEAR—Automatically clears the numeric buffer if the end user positions on the first character of the integer field (from another window field) then presses a key (without first having pressed any movement or editing keys).

WOF_BORDER—Draws a border around the integer object. In graphics mode, setting this option draws a single line border around the object. In text mode, no border is drawn. This is the default argument.

WOF_INVALID—Sets the initial status of the integer field to be "invalid." Invalid numbers fit in the absolute range determined by the object type but do not fulfill all the requirements specified by the program. For example, a integer may initially be set to 200, but the final number, edited by the end user, must be in the range "10..100." The initial number in this example fits the absolute requirements of an integer class object but does not fit into the specified range.

WOF_JUSTIFY_CENTER—Center-justifies the numeric information associated with the number object.

WOF_JUSTIFY_RIGHT—Right-justifies the numeric information associated with the integer object.

WOF_MINICELL—Uses mini-cell values to determine the mini-cell heights. Initially, a mini-cell is set to 1/10 of a cell coordinate. This flag allows for more precise positioning of objects and is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the integer field to not be a form field. If this flag is set the integer field will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the integer object from being selected. If this flag is set, the user will not be able to edit the integer information.

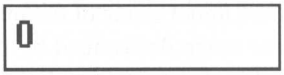
WOF_UNANSWERED—Sets the initial status of the integer field to be "unanswered." An unanswered integer field is displayed as blank space on the screen.

WOF_VIEW_ONLY—Prevents the integer object from being edited. However, the integer object may become current.

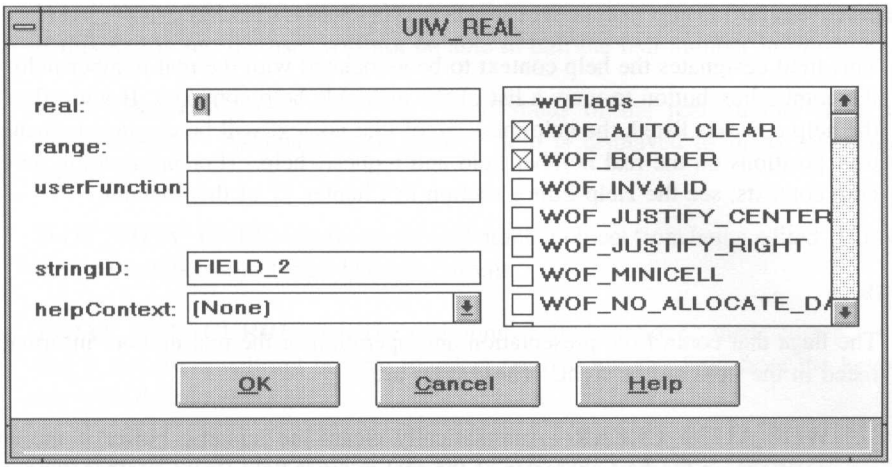
WOAF_NON_CURRENT—The integer field cannot be made current. If this flag is set, users will not be able to select the integer object from the keyboard nor with the mouse.

REAL

A real number object is used to present and collect floating-point numeric information. Decimal numbers will be displayed using decimal notation. When the decimal strings are too large for the input field, they are automatically converted to scientific notation. Selecting "Real" causes the following object to appear:



To modify the real number object, call its editor. The following window appears:



real

Enter in this field the number that you want to appear in the real number field.

range

If you want to specify a certain range of acceptable real number values, enter in this field the valid ranges. For example, if you want to accept only numbers between 100 and 100,000, enter the range of "100..100000." If no range is entered, any real number value will be accepted.

userFunction

If you want to have a user function associated with the real number object, enter in this field the name of the function. This user function must be defined somewhere in your code so that the user table created by Zinc Designer can retrieve it.

stringID

Enter in this field a string that will distinguish the real object from other objects.

helpContext

This field designates the help context to be associated with the real number field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the real number field and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

flags

The flags that control the presentation and operation of the real number information are listed in the field on the right. These flags are:

WOF_AUTO_CLEAR—Automatically clears the numeric buffer if the end user positions on the first character of the real number field (from another window field) then presses a key (without first having pressed any movement or editing keys).

WOF_BORDER—Draws a border around the real number object. In graphics mode, setting this option draws a single line border around the object. In text mode, no border is drawn. This is the default argument.

WOF_INVALID—Sets the initial status of the real number field to be "invalid." Invalid numbers fit in the absolute range determined by the object type but do not fulfill all the requirements specified by the program. For example, a real number may initially be set to 200, but the final number, edited by the end user, must be in the range "10..100." The initial number in this example fits the absolute requirements of a real number class object but does not fit into the specified range.

WOF_JUSTIFY_CENTER—Center-justifies the numeric information associated with the real object.

WOF_JUSTIFY_RIGHT—Right-justifies the numeric information associated with the real object.

WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell coordinates. This flag is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the real number field to not be a form field. If this flag is set the real number field will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the real number object from being selected. If this flag is set, the user will not be able to edit the real number information.

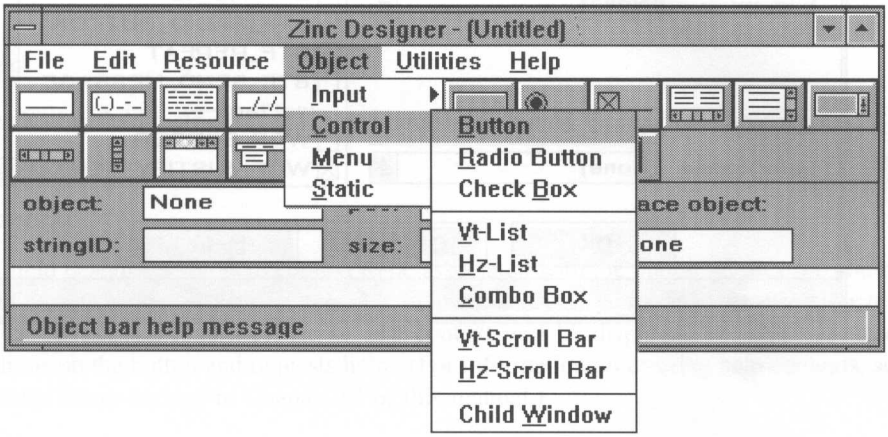
WOF_UNANSWERED—Sets the initial status of the real number field to be "unanswered." An unanswered number field is displayed as blank space on the screen.

WOF_VIEW_ONLY—Prevents the real number object from being edited. However, the real number object may become current.

WOAF_NON_CURRENT—The real number field cannot be made current. If this flag is set, users will not be able to select the real number field from the keyboard nor with the mouse.

CHAPTER 13 - CONTROL OBJECTS

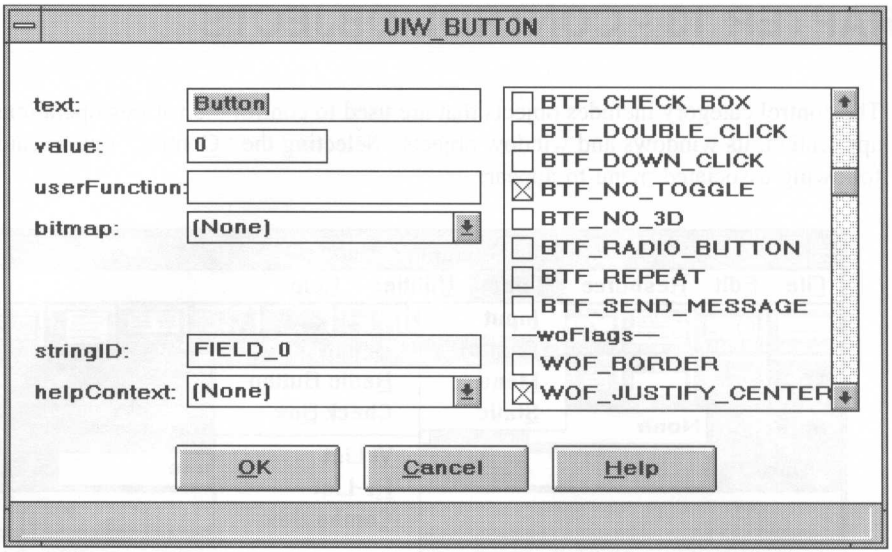
The control category includes objects that are used to control the various operations of an application, its windows and window objects. Selecting the "Control" option causes the following associated menu to appear:



BUTTON

A button is used to provide a selectable option that relates to a window. Selecting "Button" causes the following object to appear:

To modify the button, call its editor. The following window will appear:



text

Enter in this field text exactly as you want it to appear on the button. It will be automatically centered vertically. If the text string is longer than the length of the button, the button must be sized in order to display the entire text.

value

This field allows you to enter in a value that serves as a unique identification for a button. For example, you could associate the value 0 with an "ok" button and a value of 1 with a "cancel" button. This allows you to define one user-function that looks at the button values, instead of several user-functions that are tied to each button object. If the BTF_SEND_MESSAGE flag is set, the value must be an event type.

userFunction

If you want to have a user function associated with the button, enter in this field the name of the function. This user function must be defined somewhere in your code so that the user table, created by Zinc Designer, can retrieve it.

bitmap

This field designates the bitmap image to be associated with the button. Select the combo box button to view a list of the available bitmaps. If you select one of the bitmaps listed, it will be displayed on the button when a bitmap option is current in the File | Preferences window. (For information on creating bitmap images, see the Image Editor section in Chapter 16 of this manual.)

stringID

Enter in this field a string that will distinguish the button object from other objects.

helpContext

This field designates the help context to be associated with the button. Select the combo box button to view a list of the available help contexts. If you select one of the context contexts listed, the help message of that context will be displayed whenever the user positions on the button and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

flags

The flags that control the presentation and operation of the button are listed in the field on the right half of the window. The flags are:

BTF_AUTO_SIZE—Automatically computes the run-time height of the button. If the application is running in text mode, the height is set to 1. If the application is running in graphics mode, the button is approximately 120% of the default cell height.

BTF_CHECK_BOX—Creates a check box that can be toggled when selected. The WOS_SELECTED flag is set when the button is selected. In graphics mode, a square box is drawn that is marked with an 'X' when selected. In text mode the check box is represented by '[ ]' when it is not selected and '[X]' when it is selected. (**NOTE**: A check box can also be created by selecting Object | Control | Check Box or by selecting it from the object bar. For more information on check boxes, see the Check Box section in this chapter.)

BTF_DOUBLE_CLICK—Completes the button action when the button has been selected twice within a period of time specified by *UI_WINDOW_OBJECT::-doubleClickRate*.

BTF_DOWN_CLICK—Completes the button action on a button down-click, rather than on a down-click and release action.

BTF_NO_TOGGLE—Does not toggle the button's WOS_SELECTED status flag. If this flag is set, the WOS_SELECTED window object status flag is not set when the button is selected.

BTF_NO_3D—Causes the button to be displayed without shadowing.

BTF_RADIO_BUTTON—Causes the button to appear and function as a radio button. In graphics mode, a graphical radio button is drawn, while in text mode, it appears as '(•)' when selected or '( )' when not selected. All of the radio buttons in a group, list box, or window are considered to be members of the same group. Only one radio button in a group may be selected at any one time. (**NOTE**: A radio button can also be created by selecting Object | Control | Radio Button or by selecting it from the object bar. For more information on radio buttons, see the Radio Button section in this chapter.)

BTF_REPEAT—Causes the button to be re-selected (i.e., the user function is called) if it remains selected for a period of time greater than that specified by *UI_-WINDOW_OBJECT::repeatRate*.

BTF_SEND_MESSAGE—Causes the event associated with the button's value to be created and put on the event manager when the button is selected. Any temporary windows are removed from the display when this message is sent.

WOF_BORDER—Draws a single-line border around the object in graphics mode. In text mode, it causes a shadow to be displayed on the button.

WOF_JUSTIFY_CENTER—Center-justifies the text within the button.

WOF_JUSTIFY_RIGHT—Right-justifies the text within the button.

WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell cordinates. This flag is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the button object to not be a form field. If this flag is set the button object will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the button object from being selected. If this flag is set, the user will be able to see, but not select, the button.

WOAF_NON_CURRENT—The button object cannot be made current. If this flag is set, users will not be able to select the button object from the keyboard nor with the mouse; however, clicking it with the mouse or pressing the hotkey will cause it to depress and have its user function called.

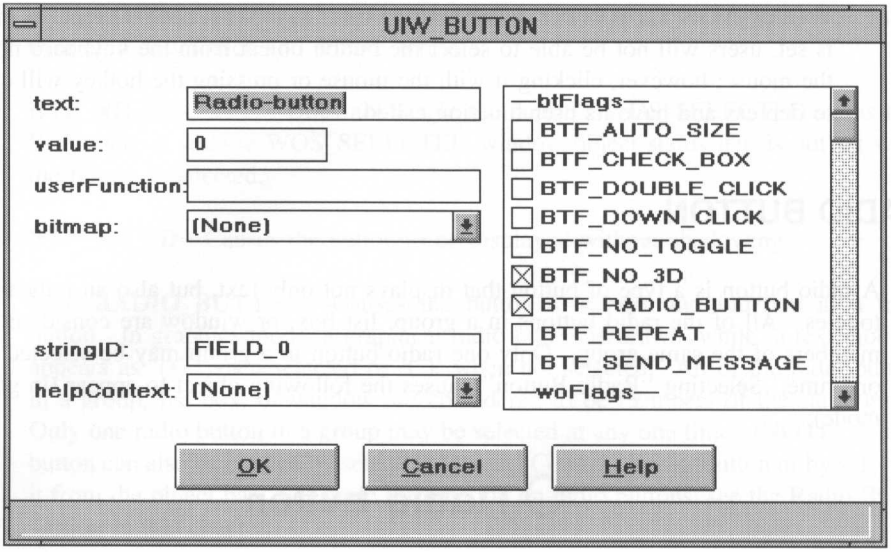
RADIO BUTTON

A radio button is a type of button that displays not only text, but also an indicator that toggles. All of the radio buttons in a group, list box, or window are considered to be members of the same group. Only one radio button in a group may be selected at any one time. Selecting "Radio Button" causes the following object to appear (in graphics mode):

In text mode, the radio button appears as '(•)' when selected or '( )' when not selected.

NOTE: To have multiple radio button groups on the same window use the group object. (For information on creating groups, refer to the Group section in Chapter 10 of this manual.)

To modify the radio button object, call its editor. The following window will appear:



Notice that the editor for the radio button is actually the editor for the standard button object but with the BTF_RADIO_BUTTON flag set. If this flag is toggled, or if the BTF_CHECK_BOX flag is selected, the button will no longer be displayed as a radio button.

text

Enter in this field text exactly as you want it to appear on the radio button. It will be automatically centered vertically. If the text string is longer than the length of the butt - on, the button must be sized in order to display the entire text.

value

This field allows you to enter in a value that serves as a unique identification for a radio button. For example, you could associate the value 0 with an "ok" button and a value of 1 with a "cancel" button. This allows you to define one user-function that looks at the button values, instead of several user-functions that are tied to each button object. If the BTF_SEND_MESSAGE flag is set, the value must be an event type.

userFunction

If you want to have a user function associated with the radio button, enter in this field the name of the function. This user function must be defined somewhere in your code so that the user table, created by Zinc Designer, can retrieve it.

bitmap

This field designates the bitmap image to be associated with the radio button. Select the combo box button to view a list of the available bitmaps. If you select one of the bitmaps listed, it will be displayed on the radio button when a bitmap option is current in the File | Preferences window. (For information on creating bitmap images, see the Image Editor section in Chapter 16 of this manual.)

stringID

Enter in this field a string that will distinguish the radio button object from other objects.

helpContext

This field designates the help context to be associated with the radio button. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the radio button and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

flags

The flags that control the presentation and operation of the radio button are listed in the field on the right half of the window. The flags are:

BTF_AUTO_SIZE—Automatically computes the run-time height of the radio button. If the application is running in text mode, the height is set to 1. If the application is running in graphics mode, the radio button is approximately 120% of the default cell height.

BTF_CHECK_BOX—Creates a check box, instead of a radio button, that can be toggled when selected. The WOS_SELECTED flag is set when the button is selected. In graphics mode, a square box is drawn that is marked with an 'X' when selected. In text mode the check box is represented by '[ ]' when it is not selected

and '[X]' when it is selected. (**NOTE**: A check box can also be created by selecting Object | Control | Check Box or by selecting it from the object bar. For more information on check boxes, see the Check Box section in this chapter.)

BTF_DOUBLE_CLICK—Completes the radio button action when the button has been selected twice within a period of time specified by *UI_WINDOW_OBJECT::-doubleClickRate*.

BTF_DOWN_CLICK—Completes the radio button action on a button down-click, rather than on a down-click and release action.

BTF_NO_TOGGLE—Causes the radio button to not be toggled when it is selected.

BTF_NO_3D—Causes the radio button to be displayed without shadowing.

BTF_RADIO_BUTTON—Causes the button to appear and function as a radio button. In graphics mode, a graphical radio button is drawn, while in text mode, it appears as '(•)' when selected or '( )' when not selected. All of the radio buttons in a group, list box, or window are considered to be members of the same group. Only one radio button in a group may be selected at any one time. This flag is set by default. If it is toggled to not be selected, the radio button becomes a regular button.

BTF_REPEAT—Causes the radio button to be re-selected (i.e., the user function is called) if it remains selected for a period of time greater than that specified by *UI_-WINDOW_OBJECT::repeatRate*.

BTF_SEND_MESSAGE—Causes the event associated with the radio button's value to be created and put on the event manager when the radio button is selected. Any temporary windows are removed from the display when this message is sent.

WOF_BORDER—Draws a single-line border around the object. In text mode, setting this setting is displayed as a shadow on the radio button.

WOF_JUSTIFY_CENTER—Center-justifies the text within the radio button.

WOF_JUSTIFY_RIGHT—Right-justifies the text within the radio button.

WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell cordinates. This flag is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the radio button object to not be a form field. If this flag is set the radio button object will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the radio button object from being selected. If this flag is set, the user will be able to see, but not select, the radio button.

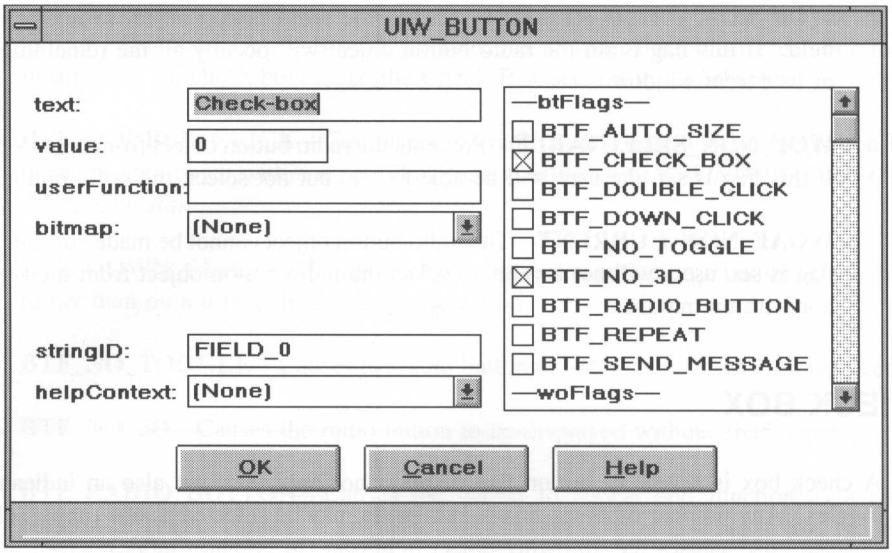
WOAF_NON_CURRENT—The radio button object cannot be made current. If this flag is set, users will not be able to select the radio button object from the keyboard nor with the mouse.

CHECK BOX

A check box is a type of button that displays not only text, but also an indicator that toggles. Any number of check boxes in a group may be selected at one time. Selecting "Check box" causes the following object to appear (in graphics mode):

In text mode the check box is represented by '[ ]' when it is not selected and '[X]' when it is selected.

To modify the check box object, call its editor. The following window will appear:



Notice that the editor for the check box is actually the editor for the standard button object but with the BTF_CHECK_BOX flag set. If this flag is toggled, or if the BTF_RADIO_-BUTTON flag is selected, the button will no longer be displayed as a check box.

text

Enter in this field text exactly as you want it to appear on the check box object. It will be automatically centered vertically. If the text string is longer than the length of the button, the button must be sized in order to display the entire text.

value

This field allows you to enter in a value that serves as a unique identification for a check box object. For example, you could associate the value 0 with an "ok" button and a value of 1 with a "cancel" button. This allows you to define one user-function that looks at the button values, instead of several user-functions that are tied to each button object. If the BTF_SEND_MESSAGE flag is set, the value must be an event type.

userFunction

If you want to have a user function associated with the check box, enter in this field the name of the function. This user function must be defined somewhere in your code so that the user table, created by Zinc Designer, can retrieve it.

bitmap

This field designates the bitmap image to be associated with the check box button. Select the combo box button to view a list of the available bitmaps. If you select one of the bitmaps listed, it will be displayed on the check box button when a bitmap option is current in the File | Preferences window. (For information on creating bitmap images, see the Image Editor section in Chapter 16 of this manual.)

stringID

Enter in this field a string that will distinguish the check box object from other objects.

helpContext

This field designates the help context to be associated with the check box button. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the check box and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

flags

The flags that control the presentation and operation of the check box button are listed in the field on the right half of the window. The flags are:

BTF_AUTO_SIZE—Automatically computes the run-time height of the check box button. If the application is running in text mode, the height is set to 1. If the application is running in graphics mode, the check box button is approximately 120% of the default cell height.

BTF_CHECK_BOX—Creates a check box that can be toggled when selected. The WOS_SELECTED flag is set when the button is selected. In graphics mode, a square box is drawn that is marked with an 'X' when selected. In text mode the check box is represented by '[ ]' when it is not selected and '[X]' when it is selected.

This flag is set by default. If it is toggled to not be selected, the check box becomes a regular button.

BTF_DOUBLE_CLICK—Completes the check box action when the button has been selected twice within a period of time specified by *UI_WINDOW_OBJECT::-doubleClickRate*.

BTF_DOWN_CLICK—Completes the check box action on a button down-click, rather than on a down-click and release action.

BTF_NO_TOGGLE—Causes the radio button to not be toggled when it is selected.

BTF_NO_3D—Causes the check box button to be displayed without shadowing.

BTF_RADIO_BUTTON—Causes the button to appear and function as a radio button, instead of a normal button. In graphics mode, a graphical radio button is drawn, while in text mode, it appears as '(•)' when selected or '( )' when not selected. All of the radio buttons in a group, list box, or window are considered to be members of the same group. Only one radio button in a group may be selected at any one time. (**NOTE**: A radio button can also be created by selecting Object | Control | Radio Button or by selecting it from the object bar. For more information on radio buttons, see the Radio Button section in this chapter.)

BTF_REPEAT—Causes the check box to be re-selected (i.e., the user function is called) if it remains selected for a period of time greater than that specified by *UI_-WINDOW_OBJECT::repeatRate*.

BTF_SEND_MESSAGE—Causes the event associated with the check box's value to be created and put on the event manager when the check box is selected. Any temporary windows are removed from the display when this message is sent.

WOF_BORDER—Draws a single-line border around the object. In text mode, setting this setting is displayed as a shadow on the check box button.

WOF_JUSTIFY_CENTER—Center-justifies the text within the check box button.

WOF_JUSTIFY_RIGHT—Right-justifies the text within the check box button.

WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell cordinates. This flag is valid in graphics modes only.

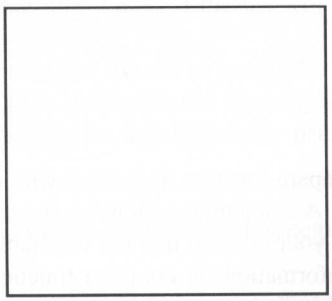
WOF_NON_FIELD_REGION—Causes the check box object to not be a form field. If this flag is set the check box object will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the check box object from being selected. If this flag is set, the user will be able to see, but not select, the check box.

WOAF_NON_CURRENT—The check box object cannot be made current. If this flag is set, users will not be able to select the check box object from the keyboard nor with the mouse.

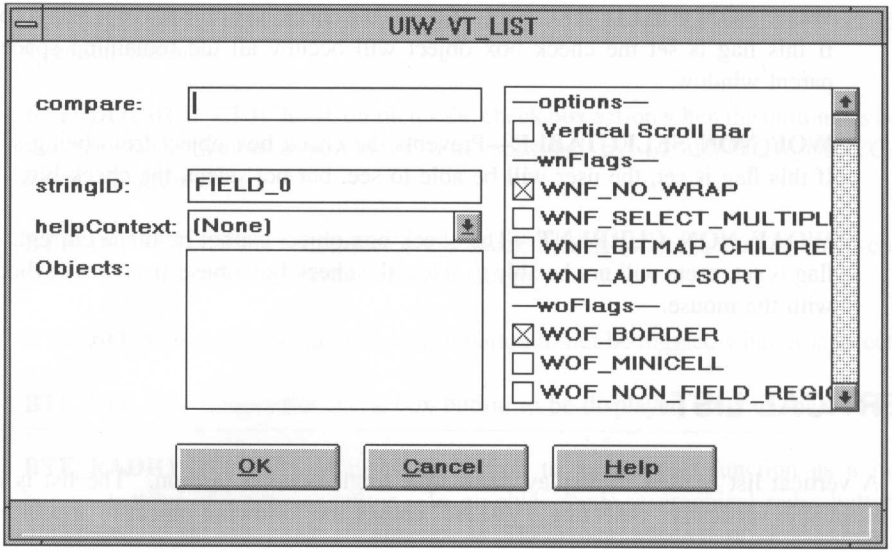
VERTICAL LIST

A vertical list is used to display items in a single-column fashion. The list is is only scrollable vertically. Selecting "Vt-List" causes the following object to appear:



Notice that the list is initially empty. A vertical list is actually a framework to which other objects can be attached. For example, a list of strings could be added to a vertical list by repeatedly selecting the string object from the menu or the toolbar and placing the string within the list. These objects will be aligned in a single-column fashion automatically. When more items are added to the lsit than can be displayed, a vertical scroll bar is automatically added.

To modify the vertical list object, call its editor. The following window will appear:



compare

If you want to have a compare function associated with the vertical list, enter in this field the name of the function. A compare function determines the order of list items and must be defined somewhere in your code so that the user table, created by Zinc Designer, can retrieve it. (For more information on compare functions for vertical lists, refer to the UI_LIST and UIW_VT_LIST chapters of the *Programmer's Reference*.)

stringID

Enter in this field a string that will distinguish the vertical list object from other objects.

helpContext

This field designates the help context to be associated with the vertical list. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the vertical list and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

Objects

This field displays the objects, listed by their string identifications, that are currently attached to the vertical list.

options and flags

The options and flags that control the presentation and operation of the vertical list are listed in the field on the right half of the window. The flags are:

Vertical Scroll Bar—Places a vertical scroll bar inside the right border of the text field.

WNF_AUTO_SORT—Assigns a compare function to alphabetize the strings within the list. If this flag is used, the compare function value should be NULL.

WNF_BITMAP_CHILDREN—Used to denote that some of the list's sub objects contain bitmaps. This flag must be set if the list will contain non-string objects.

WNF_NO_WRAP—Causes the list not to wrap when scrolling. By default, if the highlight is positioned on the last item in the list and the down key is pressed, the list will wrap and position itself on the first item in the list. The **WNF_NO_WRAP** flag disables this feature.

WNF_SELECT_MULTIPLE—Allows multiple items within the vertical list to be selected.

WOF_BORDER—In graphics mode, this flag draws a single line border around the list box. In text mode, no border is drawn.

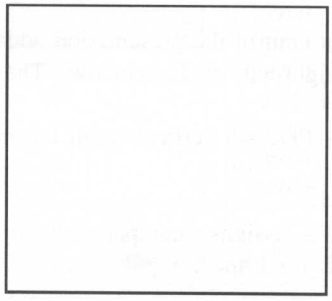
WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell cordinates. This flag is valid in graphics modes only.

WOF_NON_FIELD_REGION—The list box is not a form field. If this flag is set the list box will occupy any remaining space within the parent window.

WOF_NON_SELECTABLE—Prevents the list from being selected. If this flag is set, the user will not be able to position on the list.

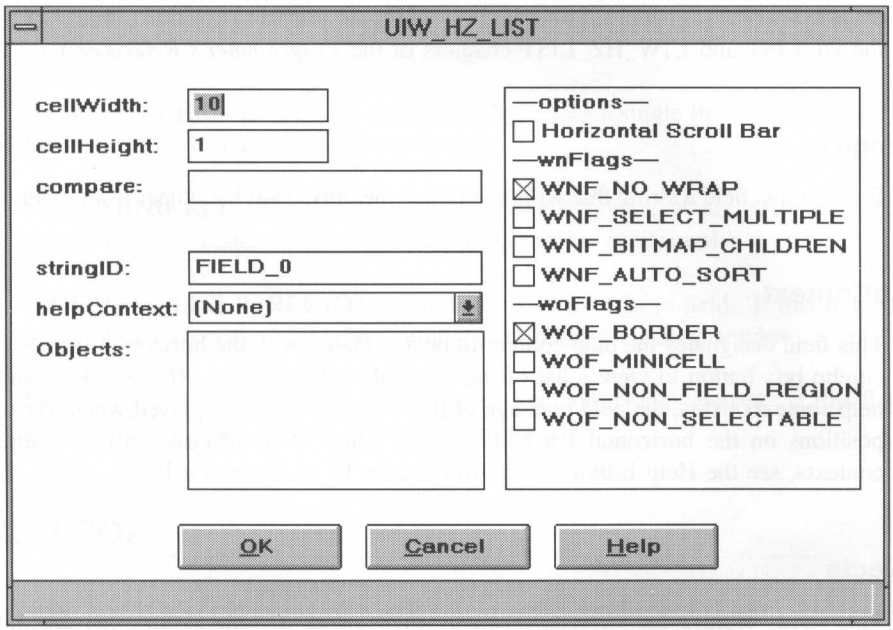
HORIZONTAL LIST

A horizontal list is used to display related information in a multiple-column fashion within a window. The list is only scrollable horizontally. Selecting "Hz-List" causes the following object to appear:



Notice that the list is initially empty. A horizontal list is actually a framework to which other objects can be attached. For example, a list of strings could be added to a horizontal list by repeatedly selecting the string object from the menu or the toolbar and placing it within the list. The items will be aligned automatically in rows and columns. When more items are added to the list than can be displayed, a horizontal scroll bar is automatically added.

To modify the horizontal list object, call its editor. The following window will appear:



cellWidth

Enter in this field a number to specify the maximum cell width of a single list item. If the list is wider than the specified width, it will be displayed with multiple columns. The default width is "10."

cellHeight

Enter in this field a number to specify the maximum cell height of a single list item. If the list is taller than the specified height, it will be displayed with multiple rows. The default height is "1."

compare

If you want to have a compare function associated with the horizontal list, enter in this field the name of the function. A compare function determines the order of list items and must be defined somewhere in your code so that the user table, created by Zinc Designer,

can retrieve it. (For more information on compare functions for horizontal lists, refer to the UI_LIST and UIW_HZ_LIST chapters of the *Programmer's Reference*.)

stringID

Enter in this field a string that will distinguish the horizontal list object from other objects.

helpContext

This field designates the help context to be associated with the horizontal list. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the horizontal list and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

Objects

This field displays the objects, listed by their string identifications, that are currently attached to the horizontal list.

options and flags

The options and flags that control the presentation and operation of the horizontal list are listed in the field on the right half of the window. The flags are:

Horizontal Scroll Bar—Places a horizontal scroll bar inside the bottom border of the text field.

WNF_AUTO_SORT—Assigns a compare function to alphabetize the strings within the list. If this flag is used, the compare function value should be NULL.

WNF_BITMAP_CHILDREN—Used to denote that some of the list's sub objects contain bitmaps. This flag must be set if the list will contain non-string objects.

WNF_NO_WRAP—Causes the list not to wrap when scrolling. By default, if the highlight is positioned on the last item in the list and the arrow key is pressed, the list will wrap and position itself on the first item in the list. The **WNF_NO_WRAP** flag disables this feature.

WNF_SELECT_MULTIPLE—Allows multiple items within the horizontal list to be selected.

WOF_BORDER—In graphics mode, this flag draws a single line border around the list box. In text mode, no border is drawn.

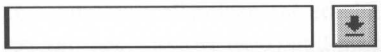
WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell cordinates. This flag is valid in graphics modes only.

WOF_NON_FIELD_REGION—The list box is not a form field. If this flag is set the list box will occupy any remaining space within the parent window.

WOF_NON_SELECTABLE—Prevents the list from being selected. If this flag is set, the user will not be able to position on the list.

COMBO BOX

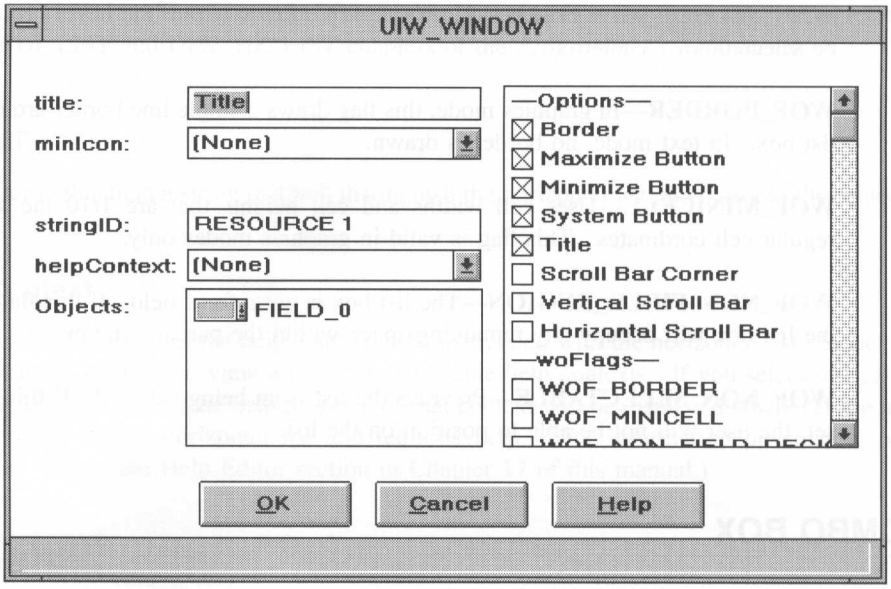
A combo box is a combination of a string field and a scrollable list box. It is used to display a list of selectable items. When one of the items is selected, it appears in the string field. Selecting "Combo Box" causes the following object to appear:



The scrollable list is displayed when the button to the right of the string field is selected. When one of the items of that list is selected, it is copied into the string field and the list box disappears. If the string field is editable, the user can enter text and the item in the list that most closely matches the characters typed will be highlighted. The user can then select the item to copy it back into the string field.

Objects are added to the combo box's list by selecting them from the menu or object bar and placing them on the combo box object. By default, they will be automatically aligned in a single column in the order in which they were created.

To modify the combo box object, call its editor. The following window will appear:



compare

If you want to have a compare function associated with the combo box, enter in this field the name of the function. A compare function determines the order of list items and must be defined somewhere in your code so that the user table, created by Zinc Designer, can retrieve it. (For more information on compare functions for combo boxes, refer to the UI_LIST and UIW_COMBO_BOX chapters of the *Programmer's Reference*.)

stringID

Enter in this field a string that will distinguish the combo box object from other objects.

helpContext

This field designates the help context to be associated with the combo box. Select the help context field's combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the combo box and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

Objects

This field displays the objects, listed by their string identifications, that are currently attached to the combo box.

options and flags

The flags that control the presentation and operation of the combo box are listed in the field on the right half of the window. The flags are:

Vertical Scroll Bar—Places a vertical scroll bar inside the right border of the combo box list field.

WNF_AUTO_SORT—Assigns a compare function to alphabetize the strings within the list. If this flag is used, the compare function value should be NULL.

WNF_BITMAP_CHILDREN—Should be set when items other than strings are added to the combo box.

WNF_NO_WRAP—Does not allow the user to scroll to the top of the combo box list by cursoring down at the bottom of the list.

WOF_AUTO_CLEAR—Automatically clears the edit buffer if the end-user positions on the combo box (from another window field) and presses a non-movement key.

WOF_BORDER—Draws a border around the combo box object. In graphics mode, setting this option draws a single-pixel border around the object. In text mode, no border is drawn.

WOF_JUSTIFY_CENTER—Center-justifies the string associated with the combo box's string field.

WOF_JUSTIFY_RIGHT—Right-justifies the string associated with the combo box's string field.

WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell cordinates. This flag is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the combo box to not be a form field. If this flag is set the combo box will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the combo box object from being selected. If this flag is set, the user will be able to see, but not select, the combo box.

WOF_UNANSWERED—Sets the initial status of the combo box to be "unanswered,"which is displayed as blank space on the screen.

WOF_VIEW_ONLY—Prevents the combo box from being edited. If this flag is set, the end-user will not be able to edit a combo box's information but will be able to browse through the information.

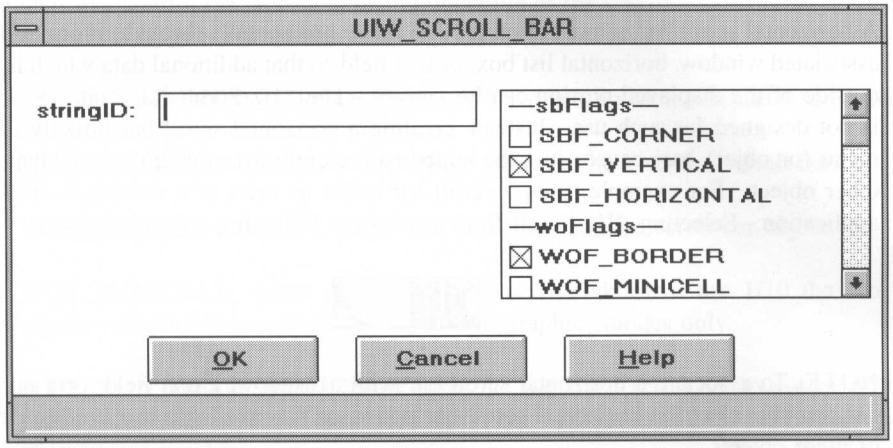
WOAF_NON_CURRENT—The combo box cannot be made current. If this flag is set, users will not be able to select the combo box from the keyboard nor with the mouse.

VERTICAL SCROLL BAR

A vertical scroll bar is most often used to move vertically through information in an associated window, vertical list box, or text field so that additional data which is hidden outside of the displayed portion can be viewed. The "Vt-Scroll Bar" option, however, is not designed for such use. Instead, creating a vertical scroll bar directly from the menu (or object bar) causes it to be added to the current resource, independent of any other object. For example, such a scroll bar could be used as a volume control for an application. Selecting "Vt-Scroll Bar" causes the following object to appear:

NOTE: To associate a vertical scroll bar with a window, a text field, or a vertical list, simply select the vertical scroll bar option or flag available in the editor for each of these objects.

To modify the scroll bar, call its editor. The following window will appear:



Notice that the editor for the vertical scroll bar is actually an editor for a generic scroll bar object but with the SBF_VERTICAL flag set. If this flag is toggled, or if another SBF flag is set, the scroll bar will no longer be displayed as a vertical one.

flags

The flags that control the presentation and operation of the vertical scroll bar are listed in the center field of the window. The flags are:

SBF_HORIZONTAL—Defines the scroll bar object to be a horizontal scroll bar.

SBF_VERTICAL—Defines the scroll bar object to be a vertical scroll bar.

WOF_BORDER—Draws a single line border around the scroll bar object.

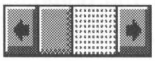
WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell cordinates. This flag is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the vertical scroll bar to not be a form field. If this flag is set the vertical scroll bar will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the vertical scroll bar from being selected. If this flag is set, the user will not be able to position on the scroll bar.

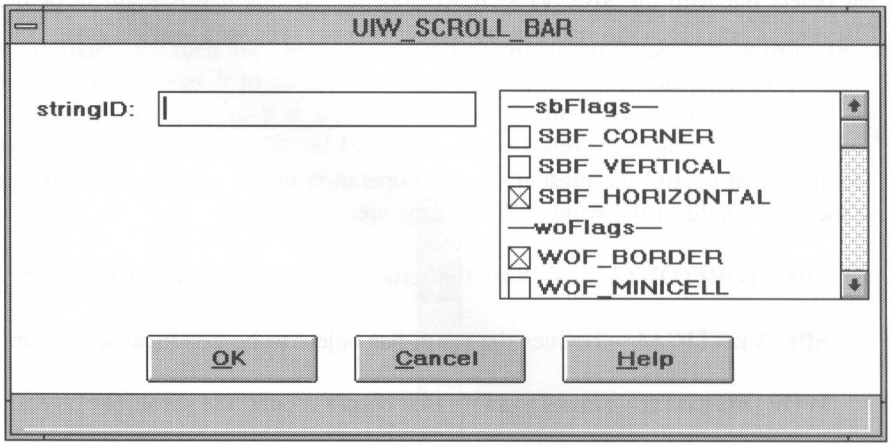
HORIZONTAL SCROLL BAR

A horizontal scroll bar is most often used to move horizontally through information in an associated window, horizontal list box, or text field so that additional data which is hidden outside of the displayed portion can be viewed. The "Hz-Scroll Bar" option, however, is not designed for such use. Instead, creating a horizontal scroll bar directly from the menu (or object bar) causes it to be added to the current resource, independent of any other object. For example, such a scroll bar could be used as a volume control for an application. Selecting "Hz-Scroll Bar" causes the following object to appear:



NOTE: To associate a horizontal scroll bar with a window, a text field, or a horizontal list, simply select the horizontal scroll bar option or flag available in the editor for each of these objects.

To modify the scroll bar, call its editor. The following window will appear:



Notice that the editor for the horizontal scroll bar is actually an editor for a generic scroll bar object but with the SBF_HORIZONTAL flag set. If this flag is toggled, or if another SBF flag is set, the scroll bar will no longer be displayed as a horizontal one.

flags

The flags that control the presentation and operation of the horizontal scroll bar are listed in the center field of the window. The flags are:

SBF_HORIZONTAL—Defines the scroll bar object to be a horizontal scroll bar.

SBF_VERTICAL—Defines the scroll bar object to be a vertical scroll bar.

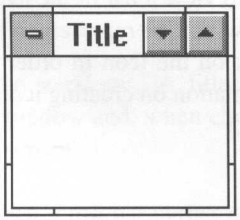
WOF_BORDER—Draws a single line border around the scroll bar object.

WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell cordinates. This flag is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the horizonal scroll bar to not be a form field. If this flag is set the horizontal scroll bar will occupy all the remaining space of its parent window.

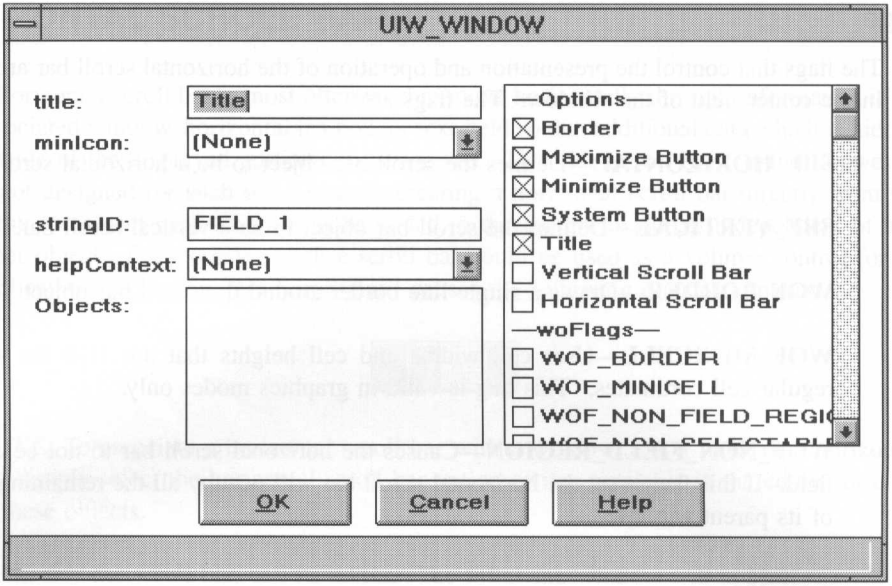
CHILD WINDOW

A window is used as a controlling structure for displaying and interacting with other objects. This object is known as a child window in order to distinguish it from the main resource window to which it must be attached. Selecting "Child Window" causes the following object to appear:



By default, the window is created with a title, a system button, a maximize button, and a minimize button. Other objects can be added by simply selecting them from the menu or object bar and placing them on the window.

To modify the child window object, call its editor. The following window will appear:



minIcon

This field designates the icon to be associated with the window when it is minimized. Select the combo box button to view a list of the available icon images. If you select one of the icons listed, the window will be represented by it when in a minimized state. The end user will be able to click on the icon in order to restore the window to its original size on the screen. (For information on creating icon images, see the Image Editor section in Chapter 16 of this manual.)

stringID

Enter in this field a string that will distinguish the child window from other objects.

helpContext

This field designates the help context to be associated with the child window. Select the field's combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever

the user positions on the child window and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

Objects

This field displays the objects, listed by their string identifications, that are currently attached to the child window.

options and flags

The options that control the presentation of the child window are listed in the upper portion of the field on the right half of the window. The options are:

Border—Draws a three-dimensional border around the outer permimeter of the window. (**NOTE**: This is an actual UIW_BORDER object, unlike the WOF_-BORDER flag, which basically just outlines the window field.)

Maximize Button—Attaches a maximize button to the window that will enlarge the window to its maximum size on the screen when selected.

Minimize Button—Attaches a minimize button to the window that will reduce the window to its minimum size on the screen when selected.

System Button—Attaches a system button to the window. When selected, a system button displays the following selectable options: Restore, Move, Size, Minimize, Maximize, and Close.

Title—Attaches a title to the window. A title is used to display short textual information about its parent window and, when clicked on with a mouse, allows you to move the window on the screen.

Scroll Bar Corner—Attaches a scroll bar corner to the window. A scroll bar corner is the grey area between a horizontal and a vertical scroll bar.

Vertical Scroll Bar—Attaches a vertical scroll bar to the window, allowing the information within the window to be scrolled vertically. The scroll bar will automatically occupy the furthest available position inside the window's right border.

Horizontal Scroll Bar—Attaches a horizontal scroll bar to the window, allowing the information within the window to be scrolled horizontally. The scroll bar will

automatically occupy the furthest available position inside the window's bottom border.

The flags that control the presentation and operation of the child window are listed in the lower portion of the field on the right half of the window. The flags are:

WOF_BORDER—Draws a single line border around the window. If the application program is running in text mode, no border is drawn.

WOF_MINICELL—Uses mini-cell values to determine the mini-cell heights. Initially, a mini-cell is set to 1/10 of a cell coordinate. This flag allows for more precise positioning of objects and is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the window to not be a form field. If this flag is set the window will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the window from being selected. If this flag is set, the user will not be able to position, nor edit, on the window.

WOAF_DIALOG_OBJECT—Creates the window as a dialog box. A dialog box is a temporary window used to display or receive information from the user. Using this flag will cause a dialog style border to be displayed.

WOAF_LOCKED—Prevents the user from removing the window from the screen display.

WOAF_MDI_OBJECT—Creates the window as an MDI window. If the MDI window is added to the window manager, it becomes an MDI parent (i.e., it can contain MDI child objects.) If the MDI window is added directly to an MDI window, it will become an MDI child object. MDI child windows can be moved or sized, but will remain entirely within the MDI parent window.

WOAF_MODAL—Prevents any other window from receiving event information from the window manager. A modal window receives all event information until it is removed from the screen display.

WOAF_NO_DESTROY—Prevents the window manager from calling the window's destructor. If this flag is set, the window can be removed from the screen display, but the programmer must call the destructor associated with the window to actually destroy it.

WOAF_NO_MOVE—Prevents the end user from changing the screen location of the window during an application.

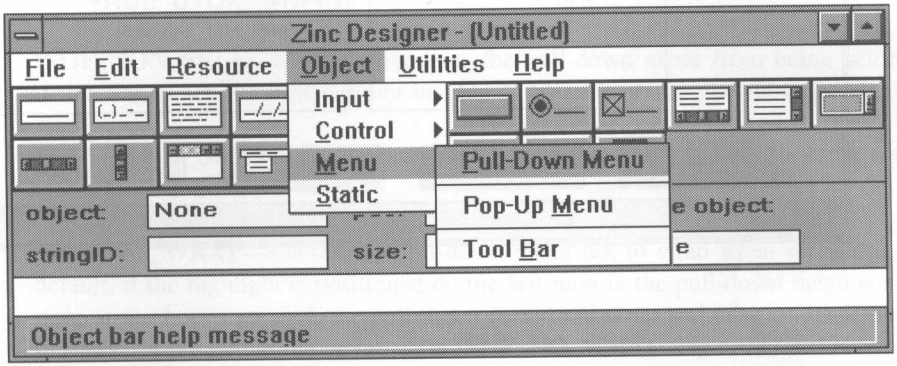
WOAF_NON_CURRENT—The window cannot be made current. If this flag is set, users will not be able to select the window from the keyboard nor with the mouse.

WOAF_NO_SIZE—Prevents the end user from changing the size of the window during an application.

WOAF_TEMPORARY—Causes the window to only occupy the screen temporarily. Once another window is selected from the screen, the temporary window is removed from the window manager (i.e., erased from the display). Once removed, a temporary window will be destroyed if the **WOAF_NO_DESTROY** flag is not set.

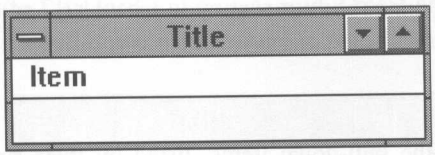
CHAPTER 14 - MENU OBJECTS

The menu category includes objects used specifically for creating menus that display selectable options. Selecting "Menu" causes the following associated menu to appear:



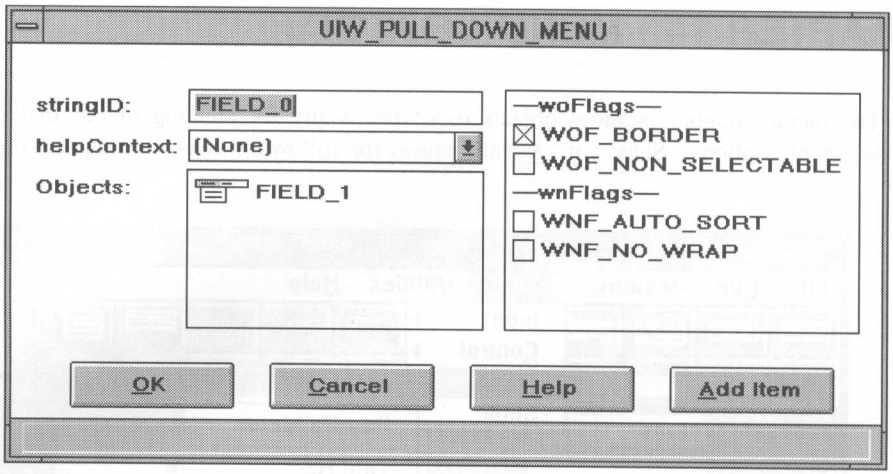
PULL-DOWN MENU

A pull-down menu acts as a structure for selectable menu items that appear in a single horizontal line. It automatically occupies the length of the top portion of the window to which it is attached. Selecting the "Pull-Down Menu" option and attaching it to a window causes the following object to appear:



A multi-level selectable menu is created by adding pull-down items and pop-up items to the pull-down menu. The pull-down menu object is automatically created with one pull-down item attached to it. (For information on adding more items, see "Add Item" of this section.)

To modify the pull-down menu, call its editor. The following window will appear:



stringID

Enter in this field a string that will distinguish the pull-down menu object from other objects.

helpContext

This field designates the help context to be associated with the pull-down menu. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the pull-down menu and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

Objects

This field displays the pull-down items, listed by their string identifications, that are currently attached to the pull-down menu. Select one of these items, and its editor will appear. (Refer to the Pull-down Item section below for more information on pull-down items.)

flags

The flags that control the presentation of the pull-down menu object are listed in the field on the right half of the window. The flags are:

WOF_BORDER—In graphics mode, this flag draws a single line border around the pull-down menu. In text mode, no border is drawn.

WOF_NON_SELECTABLE—Prevents the pull-down menu from being selected. If this flag is set, the user will not be able to position on the pull-down menu.

WNF_AUTO_SORT—Assigns a compare function to alphabetize the items within the pull-down menu.

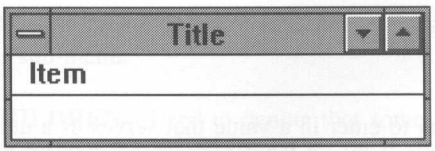
WNF_NO_WRAP—Causes the pull-down menu not to wrap when scrolling. By default, if the highlight is positioned on the last item in the pull-down menu and the right-arrow key is pressed, the pull-down menu will wrap and position itself on the first item in the pull-down menu. The **WNF_NO_WRAP** flag disables this feature.

Add Item

Selecting this button causes a pull-down item to be added to the pull-down menu. (For more information on pull-down items, refer to the Pull-Down Item section below.)

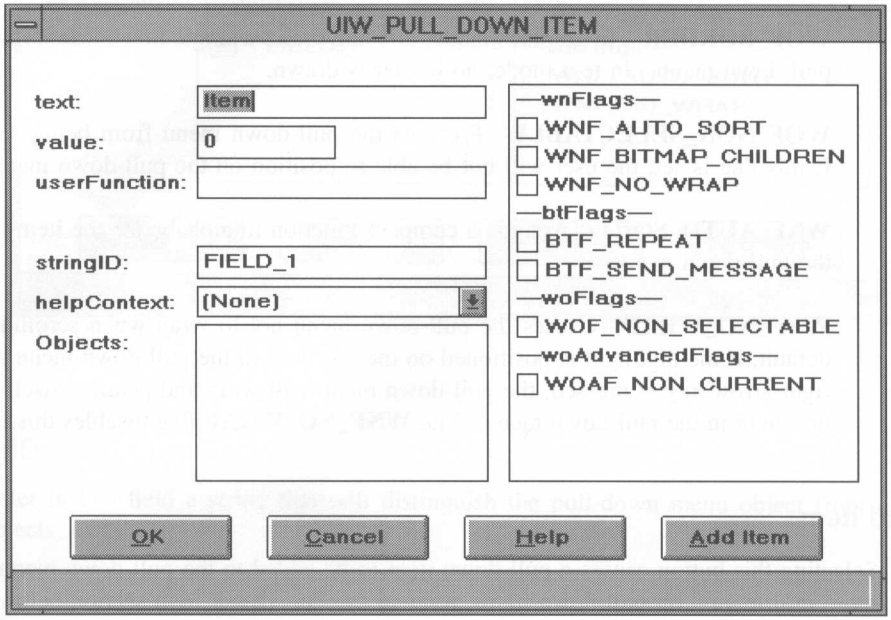
PULL-DOWN ITEM

A pull-down item serves as the first level of selection in a pull-down menu. It can only be created by selecting the "Add Item" button contained in the pull-down menu's editor. The figure below shows a pull-down menu with one pull-down item attached to it:



The multi-level effect of a pull-down menu is further achieved by adding pop-up items to the pull-down item. (For more information, see "Add Item" of this section.)

To modify the pull-down item, including adding pop-up items to it, the editor must be called. This can only be done by selecting the item from the "Objects" field of the pull-down menu's editor. Upon doing so, the following window appears:



text

Enter in this field text exactly as you want it to appear on the pull-down item. It will be automatically centered vertically.

value

This field allows you to enter in a value that serves as a unique identification for a pull-down item. This allows you to define one user-function that looks at the pull-down item values, instead of several user-functions that are tied to each pull-down item object. If the BTF_SEND_MESSAGE flag is set, the value must be an event type.

userFunction

If you want to have a user function associated with the pull-down item, enter in this field the name of the function. This user function must be defined somewhere in your code so that the user table, created by Zinc Designer, can retrieve it.

stringID

Enter in this field a string that will distinguish the pull-down item object from other objects.

helpContext

This field designates the help context to be associated with the pull-down item. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the pull-down item and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

Objects

This field displays the pop-up items, listed by their string identifications, that are currently attached to the pull-down item. Select one of these items, and its editor will appear. (Refer to the Pop-Up Item section below for more information on pop-up items.)

flags

The flags that control the presentation and operation of the pull-down item are listed in the field on the right half of the window. The flags are:

WNF_AUTO_SORT—Assigns a compare function to alphabetize the items within the pull-down item's sub-menu.

WNF_BITMAP_CHILDREN—Used to denote that some of the pull-down item's pop-up items contain bitmaps. This flag must be set if the pull-down item's sub-menu will contain non-string objects.

WNF_NO_WRAP—Causes the pull-down item's menu not to wrap when scrolling. By default, if the highlight is positioned on the last item in the pull-down item's sub-menu and the down key is pressed, the menu will wrap and position itself on the first

item in the pull-down item's sub-menu. The **WNF_NO_WRAP** flag disables this feature.

BTF_REPEAT—Causes the pull-down item to be re-selected (i.e., the user function is called) if it remains selected for a period of time greater than that specified by *UI_WINDOW_OBJECT::repeatRate*.

BTF_SEND_MESSAGE—Causes the event associated with the pull-down item's value to be created and put on the event manager when the pull-down item is selected. Any temporary windows are removed from the display when this message is sent.

WOF_NON_SELECTABLE—Prevents the pull-down item object from being selected. If this flag is set, the user will be able to see, but not select, the pull-down item.

WOAF_NON_CURRENT—The item cannot be made current. If this flag is set, users will not be able to select the item from the keyboard nor with the mouse.

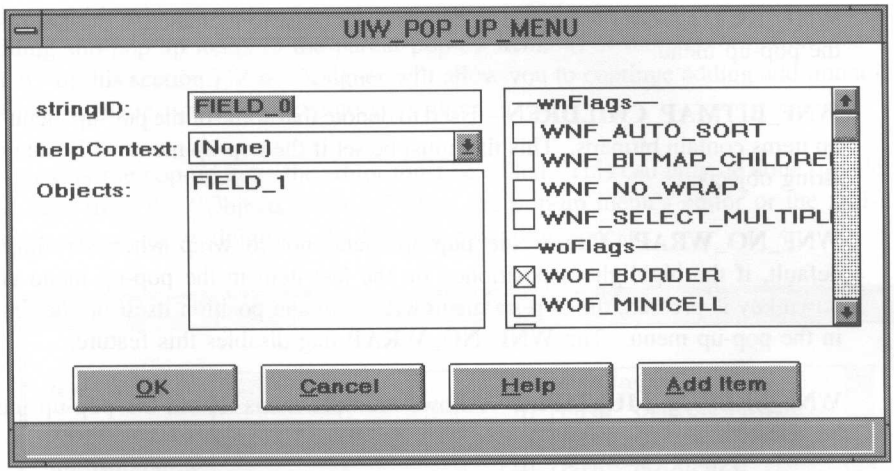
Add Item

Selecting this button causes a pop-up item to be added to the pull-down item. (For more information on pop-up items, refer to the Pop-Up Item section below.)

POP-UP MENU

A pop-up menu acts as a structure for selectable menu items which are presented in rows and columns. A selectable menu is created by adding pop-up items to the pop-up menu. The pull-down menu object is automatically created with one pull-down item attached to it.

To modify the pop-up menu, call its editor. The following window will appear:



stringID

Enter in this field a string that will distinguish the pop-up menu object from other objects.

helpContext

This field designates the help context to be associated with the pop-up menu. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the pop-up menu and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

Objects

This field displays the pop-up items, listed by their string identifications, that are currently attached to the pop-up menu. Select one of these items, and its editor will appear. (Refer to the Pop-Up Item section below for more information on pop-up items.)

flags

The flags that control the presentation of the pop-up menu object are listed in the field on the right half of the window. The flags are:

WNF_AUTO_SORT—Assigns a compare function to alphabetize the items within the pop-up menu.

WNF_BITMAP_CHILDREN—Used to denote that some of the pop-up menu's pop-up items contain bitmaps. This flag must be set if the pop-up menu will contain non-string objects.

WNF_NO_WRAP—Causes the pop-up menu not to wrap when scrolling. By default, if the highlight is positioned on the last item in the pop-up menu and the down key is pressed, the pop-up menu will wrap and position itself on the first item in the pop-up menu. The **WNF_NO_WRAP** flag disables this feature.

WNF_SELECT_MULTIPLE—Allows multiple items within the pop-up menu to be selected.

WOF_BORDER—In graphics mode, this flag draws a single line border around the pop-up menu. In text mode, no border is drawn.

WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell coordinates. This flag is valid in graphics modes only.

WOF_NON_FIELD_REGION—The pop-up menu is not a form field. If this flag is set the pop-up menu will occupy any remaining space within the parent window.

WOF_NON_SELECTABLE—Prevents the pop-up menu from being selected. If this flag is set, the user will not be able to position on the pop-up menu.

Add Item

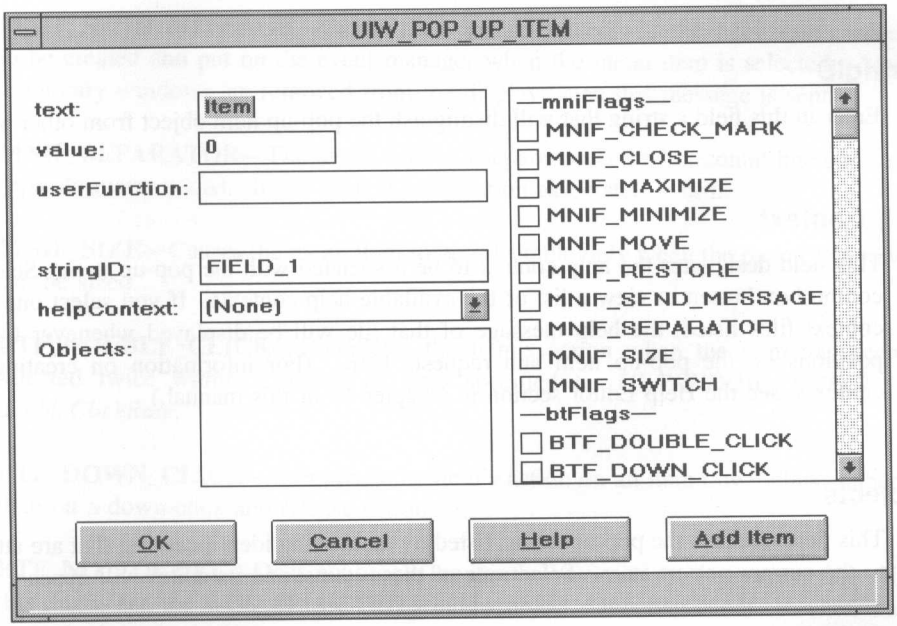
Selecting this button causes a pop-up item to be added to the pop-up menu. (For more information on pop-up items, refer to the Pop-Up Item section below.)

POP-UP ITEM

A pop-up item is used to display and select options associated with a list of menu items. It can be attached to a pop-up menu, to a pull-down item (as the second level of selection within a pull-down menu), or to another pop-up item. It can only be created by selecting the "Add Item" button contained in either the pop-up menu's editor or the pull-down item's editor.

The multi-level effect of a pop-up menu or a pull-down menu is further achieved by adding sub-pop-up items to the parent pop-up item. (For more information, see "Add Item" of this section.) Zinc Designer will allow you to continue adding additional levels as long as there is available memory for them.

To modify the pop-up item, the editor must be called. This can only be done by selecting the item from the "Objects" field of either the pop-up menu's editor or the pull-down item's editor. Upon doing so, the following window appears:



text

Enter in this field text exactly as you want it to appear on the pop-up item. It will be automatically centered vertically.

value

This field allows you to enter in a value that serves as a unique identification for a pop-up item. This allows you to define one user-function that looks at the pop-up item values,

instead of several user-functions that are tied to each pop-up item object. If the MNIF_-SEND_MESSAGE flag is set, the value must be an event type.

userFunction

If you want to have a user function associated with the pop-up item, enter in this field the name of the function. This user function must be defined somewhere in your code so that the user table, created by Zinc Designer, can retrieve it.

stringID

Enter in this field a string that will distinguish the pop-up item object from other objects.

helpContext

This field designates the help context to be associated with the pop-up item. Select the combo box button to view a list of the available help contexts. If you select one of the context files listed, the help message of that file will be displayed whenever the user positions on the pop-up item and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

Objects

This field displays the pop-up items, listed by their string identifications, that are attached to the current pop-up item. Select one of these items, and its editor will appear.

flags

The flags that control the presentation and operation of the pop-up item are listed in the field on the right half of the window. The flags are:

MNIF_CHECK_MARK—Marks the first position of the menu item's string information with a check-mark if the item has been selected (i.e., the WOS_SELECTED status flag is set).

MNIF_MAXIMIZE—Causes the menu item to be selectable only when the parent window can be maximized.

MNIF_MINIMIZE—Causes the menu item to be selectable only when the parent window can be minimized.

MNIF_MOVE—Causes the menu item to be selectable only when the parent window can be moved.

MNIF_RESTORE—Causes the menu item to be selectable only when the parent window is in a maximized or minimized state.

MNIF_SEND_MESSAGE—Causes the event associated with the menu item's value to be created and put on the event manager when the menu item is selected. Any temporary windows are removed from the display when this message is sent.

MNIF_SEPARATOR—The menu item is a separator (i.e., a horizontal line used to separate menu items). It has no text information associated with it.

MNIF_SIZE—Causes the menu item to be selectable only when the parent window can be sized.

BTF_DOUBLE_CLICK—Completes the item's action when the item has been selected twice within a period of time specified by *UI_WINDOW_OBJECT::-doubleClickRate*.

BTF_DOWN_CLICK—Completes the item's action on an item down-click, rather than on a down-click and release action.

BTF_NO_TOGGLE—Does not toggle the item's WOS_SELECTED status flag. If this flag is set, the WOS_SELECTED window object status flag is not set when the menu item is selected.

BTF_SEND_MESSAGE—Causes the event associated with the menu item's value to be created and put on the event manager when the menu item is selected. Any temporary windows are removed from the display when this message is sent.

WOF_BORDER—Draws a single line border around the pop-up item. In text mode, no border is drawn.

WOF_JUSTIFY_CENTER—Center-justifies the text information associated with the pop-up item.

WOF_JUSTIFY_RIGHT—Right-justifies the text information associated with the pop-up item.

WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell coordinates. This flag is valid in graphics modes only.

WOF_NON_FIELD_REGION—Causes the pop-up item to not be a form field. If this flag is set the item will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the pop-up item from being selected.

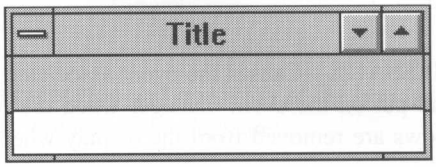
WOAF_NON_CURRENT—Prevents the item from being made current. If this flag is set, users will not be able to select the item from the keyboard nor with the mouse.

Add Item

Selecting this button causes an additional pop-up item to be added to the current pop-up item.

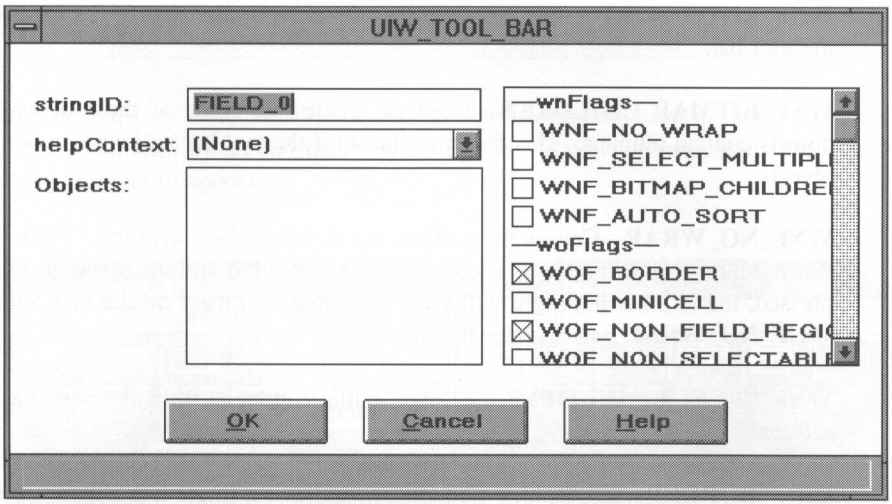
TOOL BAR

A tool bar is used as a controlling structure for a set of selectable window objects. It differs from the pull-down menu in that a variety of objects can be added to it—not just string items. The tool bar will automatically occupy the upper-most area available in a window, positioning itself directly below the pull-down menu, if one exists. Multiple tool bars may be added to a window. Selecting "Tool Bar" and attaching it to a window causes the following object to appear:



An object can be added to the tool bar by selecting the desired object from the control window's menu or object bar and placing it on the resource window's tool bar. The control window's object bar itself is an example of a group of bitmapped buttons that have been attached to a tool bar.

To modify the tool bar, call its editor. The following window will appear:



stringID

Enter in this field a string that will distinguish the tool bar from other objects.

helpContext

This field designates the help context to be associated with the tool bar. Select the combo box button to view a list of the available help contexts. If you select one of the context files listed, the help message of that file will be displayed whenever the user positions on the tool bar and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 17 of this manual.)

Objects

This field displays the objects, listed by their string identifications, that are currently attached to the tool bar.

flags

The flags that control the presentation and operation of the tool bar are listed in the field on the right half of the window. The flags are:

WNF_AUTO_SORT—Assigns a compare function to alphabetize the items within the tool bar.

WNF_BITMAP_CHILDREN—Used to denote that some of the tool bar's sub-objects contain bitmaps. This flag must be set if the tool bar will contain non-string objects.

WNF_NO_WRAP—Causes the tool bar not to wrap when scrolling. By default, if the highlight is positioned on the last item in the tool bar and the down-arrow key is pressed, the tool bar highlight will wrap and reposition itself on the first item. The **WNF_NO_WRAP** flag disables this feature.

WNF_SELECT_MULTIPLE—Allows multiple items within the tool bar to be selected.

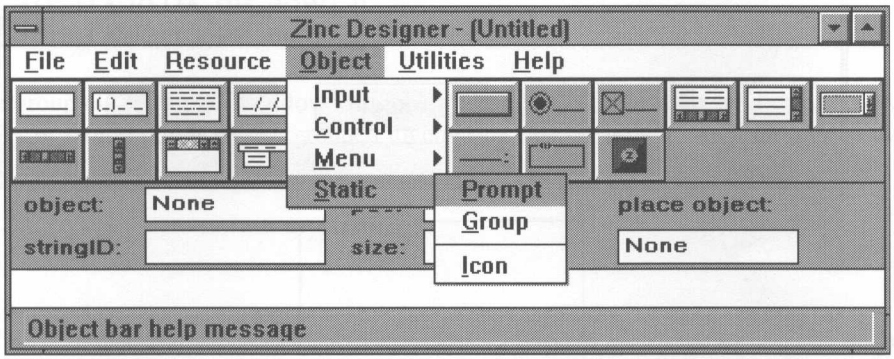
WOF_BORDER—In graphics mode, this flag draws a single line border around the tool bar. In text mode, no border is drawn.

WOF_NON_FIELD_REGION—The tool bar is not a form field. If this flag is set the tool bar will occupy any remaining space within the parent window.

WOF_NON_SELECTABLE—Prevents the tool bar from being selected. If this flag is set, the user will not be able to position on the tool bar.

CHAPTER 15 - STATIC OBJECTS

The static category includes window objects that are generally not designed to be edited nor interacted with by an end user. Selecting the "Static" option causes the following associated menu to appear:

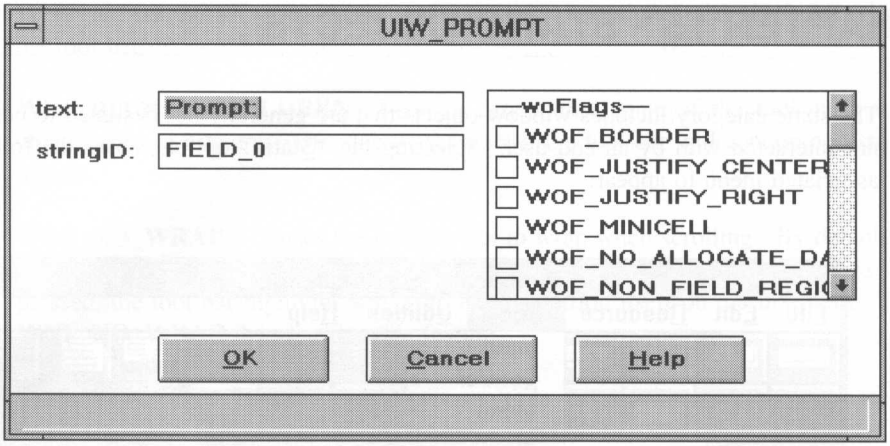


PROMPT

A prompt object is used to provide lead information for another window object. Selecting "Prompt" causes the following object to appear:

Prompt

To modify the prompt object, call its editor. The following window will appear:



text

Enter in this field a text string exactly as you want it to appear in the prompt. It will be automatically centered vertically. If either the WOF_JUSTIFY_CENTER or the WOF_-JUSTIFY_RIGHT flag is set and text string is longer than the length of the prompt field, the field must be sized in order to display the entire text.

stringID

Enter in this field a string that will distinguish the prompt object from other objects.

flags

The flags that control the presentation of the prompt are listed in the field on the right half of the window. The flags are:

WOF_BORDER—Draws a single-line border around the object in graphics mode. In text mode, it causes a shadow to be displayed on the prompt.

WOF_JUSTIFY_CENTER—Center-justifies the text within the prompt.

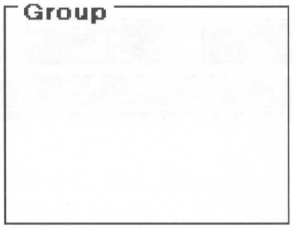
WOF_JUSTIFY_RIGHT—Right-justifies the text within the prompt.

WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell cordinates. This flag is valid in graphics modes only.

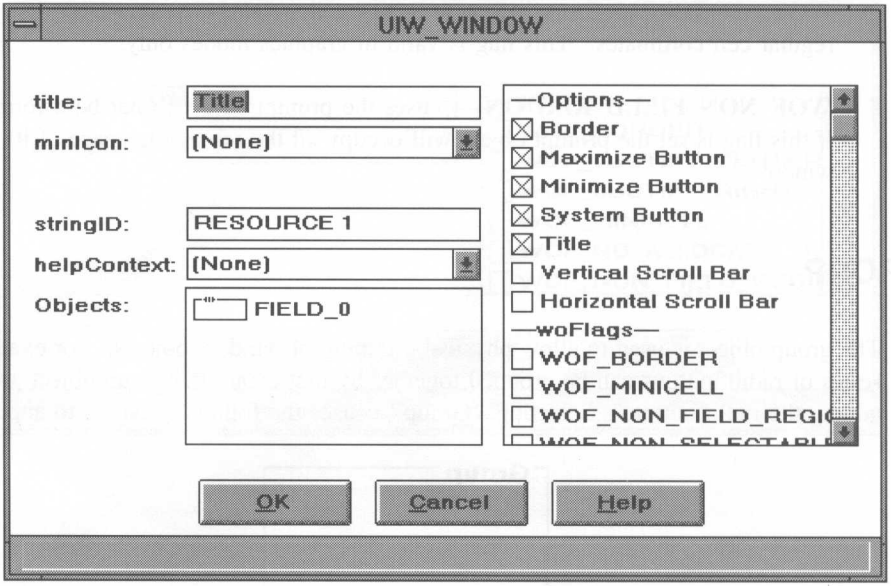
WOF_NON_FIELD_REGION—Causes the prompt object to not be a form field. If this flag is set the prompt object will occupy all the remaining space of its parent window.

GROUP

The group object is used to allow physical grouping of window objects. For example, a series of radio buttons can be grouped together by first creating a group object and then adding the radio buttons. Selecting "Group" causes the following object to appear:



To modify the group object, call its editor. The following window will appear:



text

Enter in this field text exactly as you want it to appear in the upper left corner of the group object's border. If the text string is longer than the width of the group box, only the portion that fits will be displayed.

stringID

Enter in this field a string that will distinguish the group object from other objects.

Objects

This field displays the objects, listed by their string identifications, that are currently attached to the group object.

flags

The flags that control the presentation and operation of the group are listed in the field on the right half of the window. The flags are:

WNF_BITMAP_CHILDREN—Used to denote that some of the group's sub-objects contain bitmaps. This flag must be set if the group will contain non-string objects.

WNF_SELECT_MULTIPLE—Allows multiple items within the group to be selected.

WOF_BORDER—In graphics mode, this flag draws a single line border around the group box. In text mode, no border is drawn.

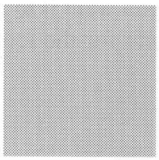
WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell cordinates. This flag is valid in graphics modes only.

WOF_NON_FIELD_REGION—The group box is not a form field. If this flag is set the group box will occupy any remaining space within the parent window.

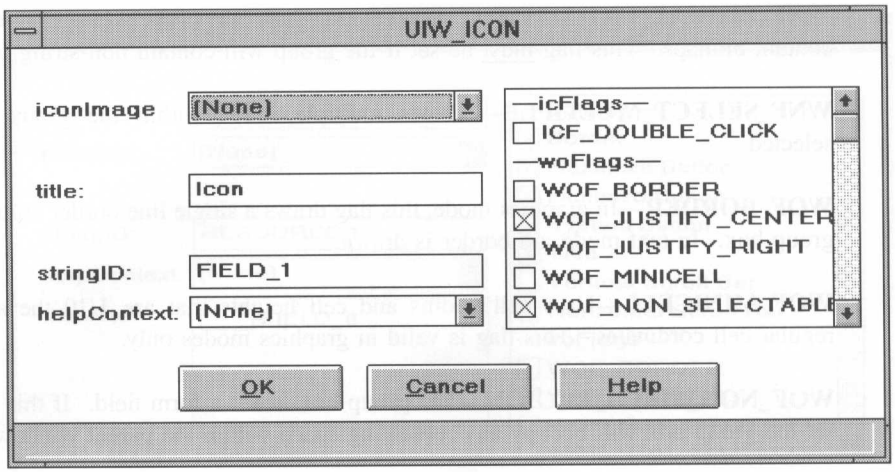
WOF_NON_SELECTABLE—Prevents the group from being selected. If this flag is set, the user will not be able to position on the group.

ICON

An icon is used to display a 32x32 pixel bitmap image to the screen. It is part of the static category because it is often present in an application as an indicator of some sort that cannot be interacted with; however, an icon can also be created for interaction purposes, such as a question mark icon that displays help when selected. Selecting "Icon" causes the following object to appear:



To modify the icon, call its editor. The following window will appear:



iconImage

This field designates the image to be associated with the icon. Select the combo box button to view a list of the available images. If you select one of the images listed, it will be displayed on the icon. (For information on creating bitmap images, see the Image Editor section in Chapter 16 of this manual.)

title

Enter in this field text exactly as you want it to appear in the the rectangular region below the icon. If you do not want a title for the icon, delete the default string "Icon."

stringID

Enter in this field a string that will distinguish the icon object from other objects.

helpContext

This field designates the help context to be associated with the icon. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user

positions on the icon and requests help. (For information on creating help contexts, see the Help Editor section in Chapter 16 of this manual.)

flags

The flags that control the presentation and operation of the icon are listed in the field on the right half of the window. The flags are:

ICF_DOUBLE_CLICK—Completes the icon action on a mouse double-click, rather than on a single-click and release action.

WOF_BORDER—Draws a single-line border around the icon bitmap and another border around the icon's title.

WOF_JUSTIFY_CENTER—Center-justifies the string information within the icon. This flag only has effect if the icon is attached to a list.

WOF_JUSTIFY_RIGHT—Right-justifies the string information within the icon. This flag only has effect if the icon is attached to a list.

WOF_NON_SELECTABLE—Indicates that the icon object cannot be selected. If this flag is set, the user will not be able to select the icon.

CHAPTER 16 - UTILITIES OPTIONS

The utilities category provides options that allow you to create images and help utilities to be used throughout your application. Selecting "Utilities" causes the following menu to appear:

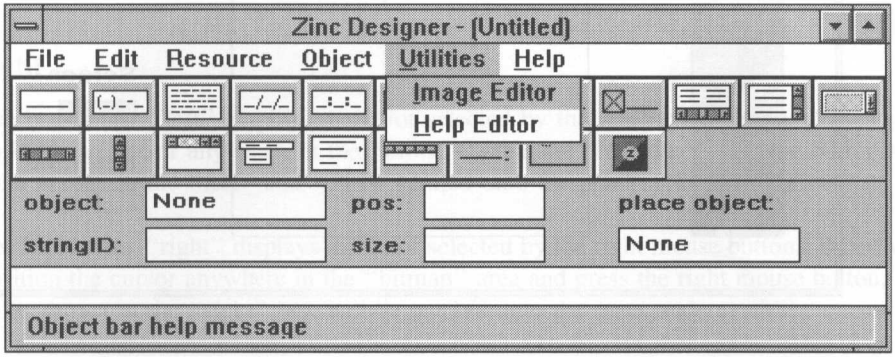
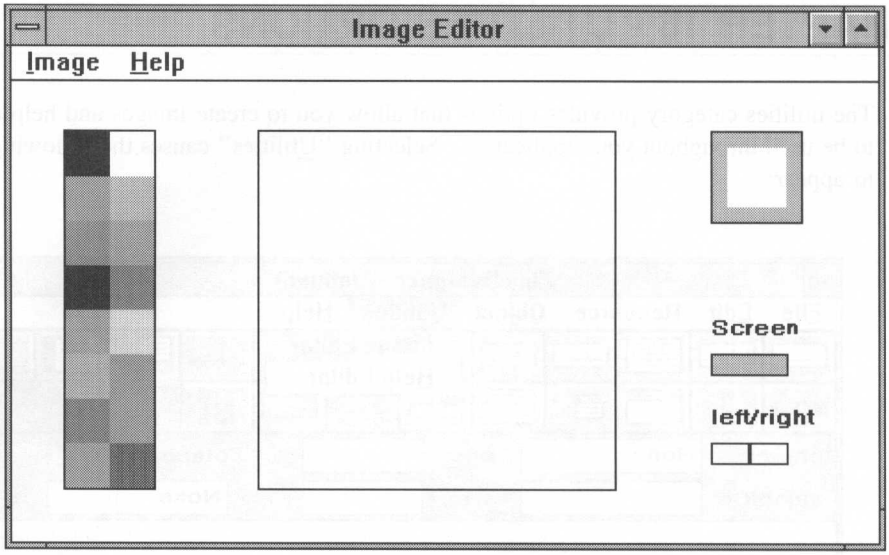


IMAGE EDITOR

The image editor allows you to create icon and bitmap images that can be assigned to other objects in your application. This option only has effect in graphics mode. Selecting "Image Editor" causes the following window to appear:



colors field

The available colors are displayed in this field. To select a color, click on it with the mouse.

drawing field

This field is the drawing area where you create your icon. You can paint one pixel at a time by positioning on it and pressing a mouse button, or you can paint in continuous motion by holding down a mouse button and dragging the cursor.

image field

This area displays the image in its actual size as you create it.

Screen

This field is used when you want to have part of your image to be transparent (i.e., to show through to the screen behind it). Whichever mouse button is used to click on this field will have the ability to draw the transparent region. For example, if you want to create a window icon that displays the area underneath it, you could draw the frame for the window with a color from the color field and then click on the "Screen" field and fill the frame in.

left/right

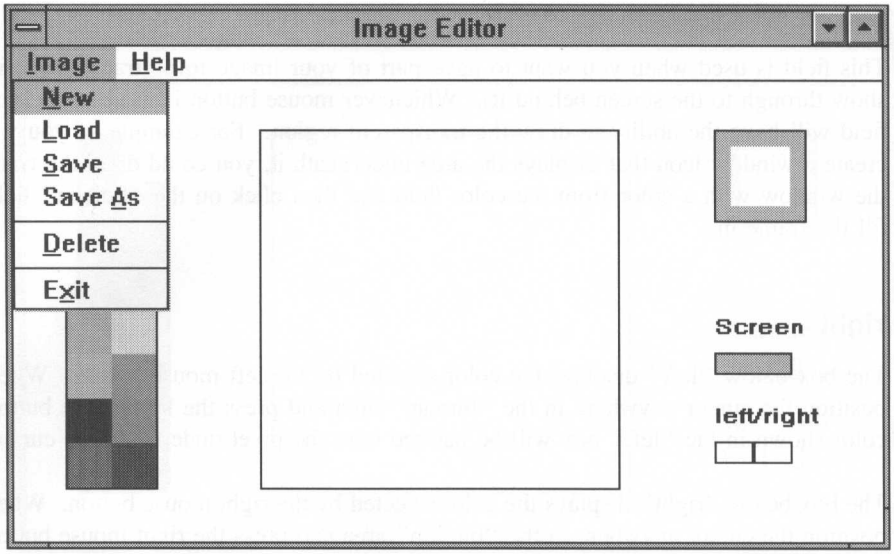
The box below "left" displays the color selected by the left mouse button. When you position the cursor anywhere in the "bitmap" area and press the left mouse button, the color shown in the "left" box will be painted onto the pixel underneath the cursor.

The box below "right" displays the color selected by the right mouse button. When you position the cursor anywhere in the "bitmap" area and press the right mouse button, the color shown in the "right" box will be painted onto the pixel underneath the cursor.

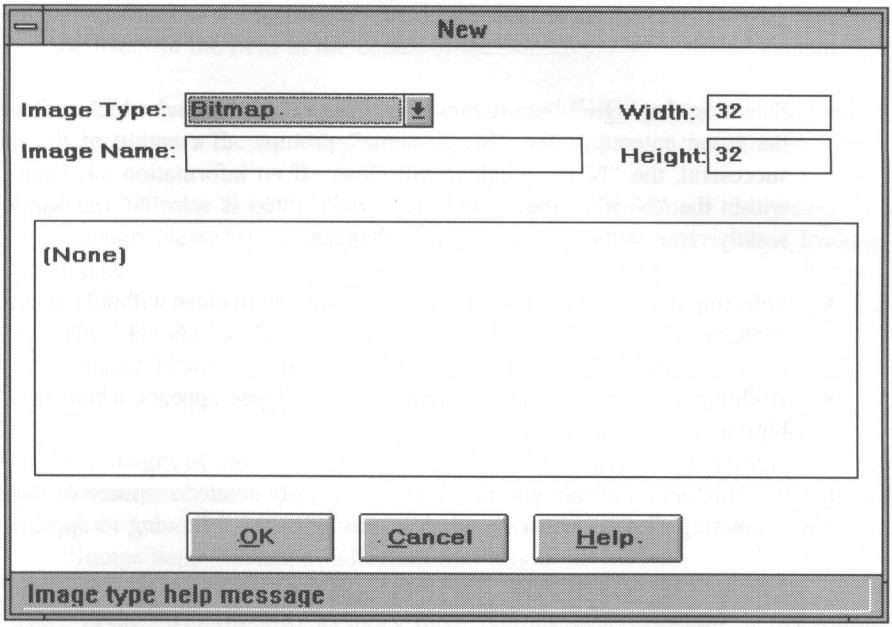
If either the right or the left mouse buttons have selected the "Screen" field instead of a color from the colors field, the appropriate box of the "left/right" field will be grey.

Image options

The image options of the window's pull-down menu control the general operations of the image editor. Selecting "Image" causes the following menu to appear:



New—This option allows you to create a new image. Selecting it causes a window similar to the following to appear:



Interaction with the fields of the "New" window is accomplished as follows:

- The "Image Type" field designates the type of image—bitmap or icon—to be created. Select the combo box button or press the <down arrow>; then select one of these two options. If the image will be designed for use with an icon, select "Icon." For use with all other objects, select "Bitmap."

- If you want to create a new image, enter the name for the new image in the "Image Name" field.

- Enter in the "Width" field the desired pixel width for the image. If the image type is icon, the width must be 32.

- Enter in the "Height" field the desired pixel height for the image. If the image type is icon, the width must be 32.

- Other images of the current type (i.e., bitmap or icon) that have been created with the image editor in the current application file are listed in the field in the center of the window. If one of these images is selected, its name will appear at the "Image name" prompt, indicating that it is to be loaded. (For more information on loading a previously created image, see the explanation for the "Load" option below.)

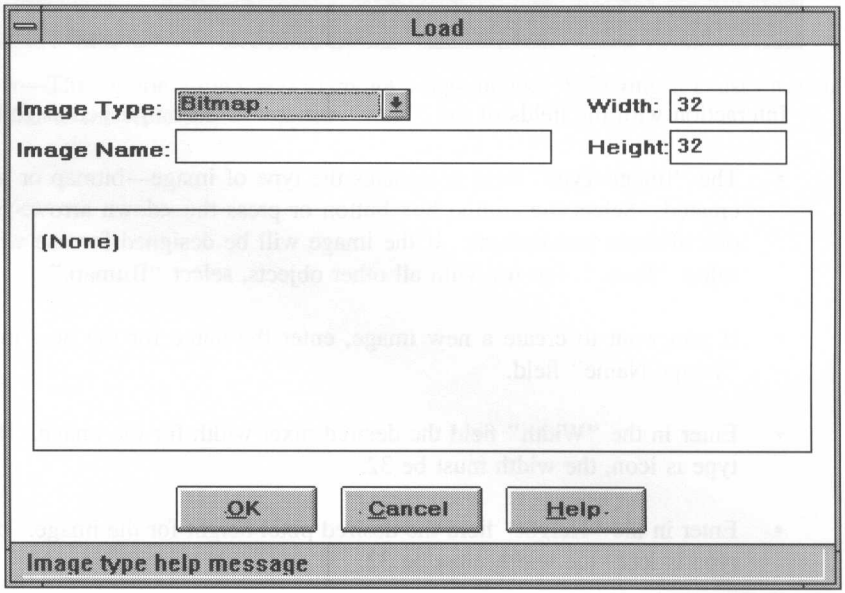
The "New" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes an image to be created which will be given the name entered at the "Image name" prompt. If creation of the image is successful, the "New" window will close. If no information has been entered within the "New" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about creating new images appears when the "Help" button is selected.

Load—This option allows you to recall a previously created resource of the current file. Selecting "Load" causes a window similar to the following to appear:



Interaction with the fields of the "Load" window is accomplished as follows:

- The "Image Type" field designates the type of the image—bitmap or icon—to be loaded. Select the combo box button or press the <down arrow> and select

one of these two options to view the available images of that type, which will be listed in the field in the center of the window.

- Enter in the "Image Name" field the name of the image that is to be loaded, or select it from the list below and the name will appear at this prompt.

- The "Width" field displays the pixel width for the image designated at the "Image Name" prompt. This number cannot be changed when loading an image.

- The "Height" field displays the pixel height for the image designated at the "Image Name" prompt. This number cannot be changed when loading an image.

- All images of the current type (i.e., bitmap or icon) that have been created with the image editor in the current application file are listed in the field in the center of the window. If one of these images is selected, its name will appear at the "Image name" prompt, indicating that it is to be loaded.

The "Load" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes the image designated at the "Image name" prompt to be loaded. If the image is successfully loaded, the "Load" window will close. If no information has been entered within the "Load" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

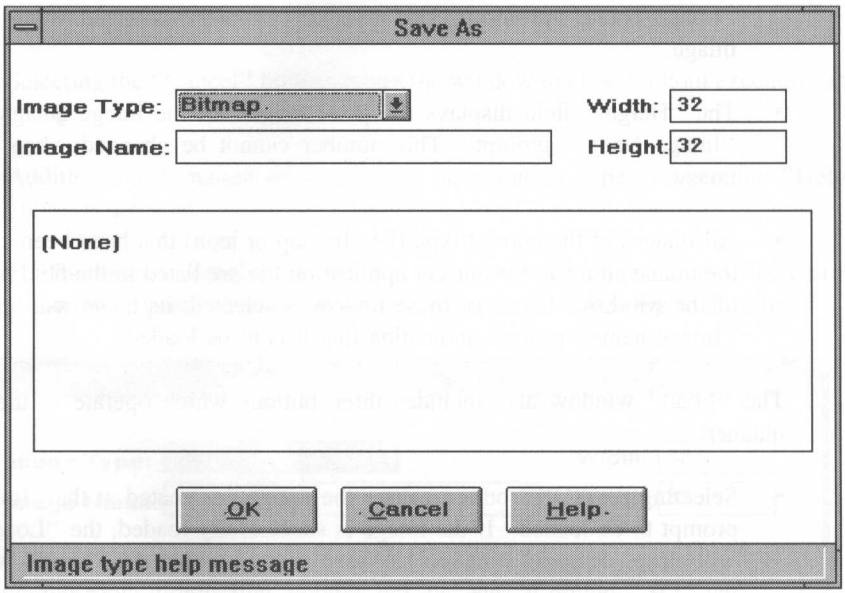
- Additional information about loading images appears when the "Help" button is selected.

Once the image has been loaded and appears in the drawing area, it can be modified in any way. When the Image | Save option is subsequently selected, the image will be saved in its present condition, replacing the original version. (Refer to the Save and Save As options of this section for more information on saving images.)

Save—Selecting this option causes the current image to be saved in its present condition. If the image has never been named, the "Save As" window will appear and allow you to name it by entering a name at the "Image Name" prompt. When you select the "OK" button, the "Save As" window will close and the image will

be saved under that name. (See the "Save As" section for further details on how to save a image for the first time.)

Save As—This option allows you to either save an image that has not been previously named or to save the current image under another name. Selecting "Save As" causes the following window to appear:



Interaction with the fields of the "Save As" window is accomplished as follows:

- The "Image Type" field designates the type of image—bitmap or icon—to be saved. Select the combo box button or press the <down arrow>; then select one of these two options. If the image will be designed for use with an icon, select "Icon." For use with all other objects, select "Bitmap."
- Enter the name for the image in the "Image Name" field.
- Enter in the "Width" field the desired pixel width for the image. If the image type is icon, the width must be 32.
- Enter in the "Height" field the desired pixel height for the image. If the image type is icon, the width must be 32.

- Other images of the current type (i.e., bitmap or icon) that have been created with the image editor in the current application file are listed in the field in the center of the window. If one of these images is selected, its name will appear at the "Image name" prompt, and the current image will replace the previous information of the specified image when the "OK" button is selected.

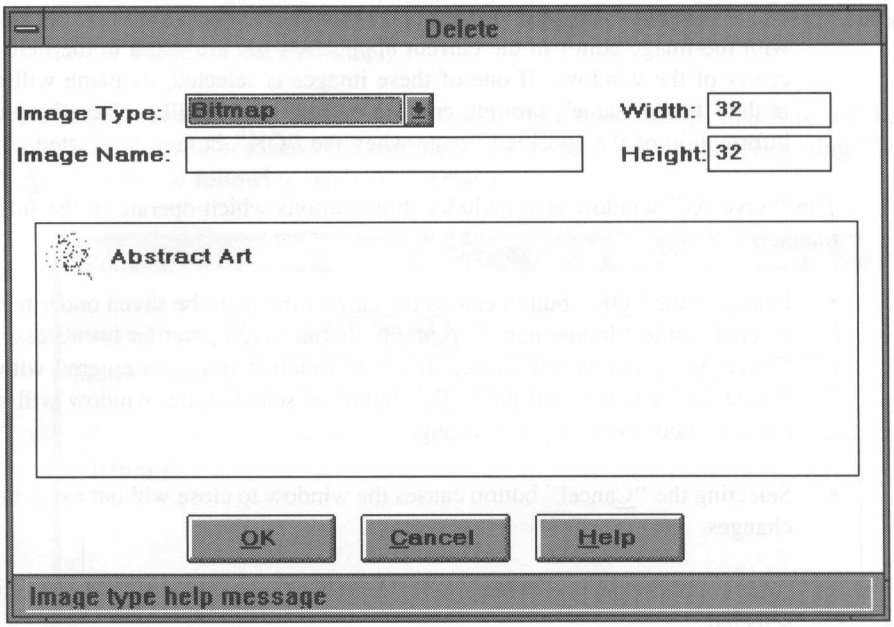
The "Save As" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes the current image to be saved under the name entered at the "Image name" prompt. If the save operation is successful, the "Save As" window will close. If no information has been entered within the "Save As" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about saving images appears when the "Help" button is selected.

Delete—This option allows you to delete an image. Selecting "Delete" causes a window similar to the following to appear:



Interaction with the fields of the "Delete" window is accomplished as follows:

- The "Image Type" field designates the image type of the image—bitmap or icon—to be deleted. Select the combo box button or press the <down arrow> and select one of these two options to view the available images of that type.

- Enter in the "Image Name" field the name of the image to be deleted, or select it from the list in the center of the window and the name will appear at this prompt.

- The "Width" field displays the pixel width for the image designated at the "Image Name" prompt. This field is not editable.

- The "Height" field displays the pixel height for the image designated at the "Image Name" prompt. This field is not editable.

- All images of the current type (i.e., bitmap or icon) that have been created with the image editor in the current application file are listed in the field in the center of the window. If one of these images is selected, its name will appear at the "Image Name" prompt, indicating that it is to be deleted.

The "Delete" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes the image designated at the "Image name" prompt to be deleted. If the image is successfully deleted, the "Delete" window will close. If no information has been entered within the "Delete" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about deleting images appears when the "Help" button is selected.

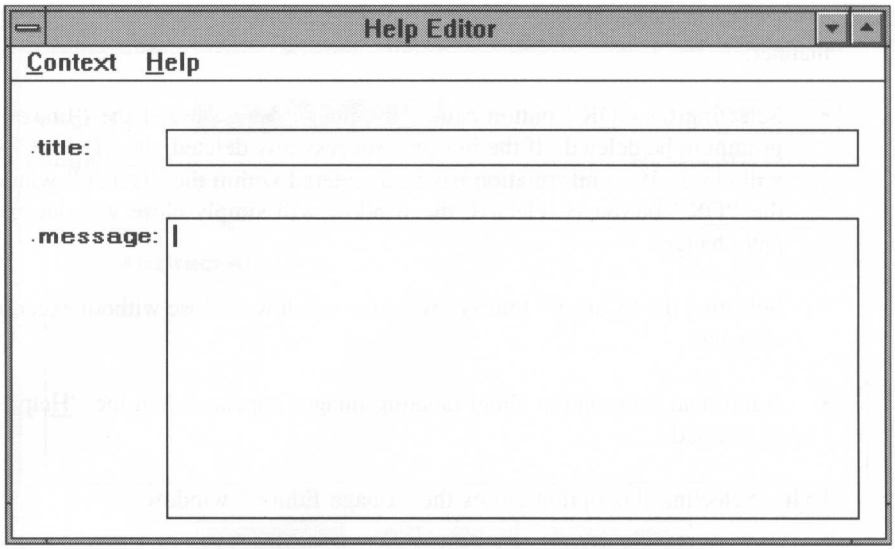
Exit—Selecting this option closes the "Image Editor" window.

Help option

The help option, when selected, displays help on creating images using the image editor.

HELP EDITOR

The help editor allows you to create help contexts to be used throughout your application. Selecting "Help Editor" causes the following window to appear:



title

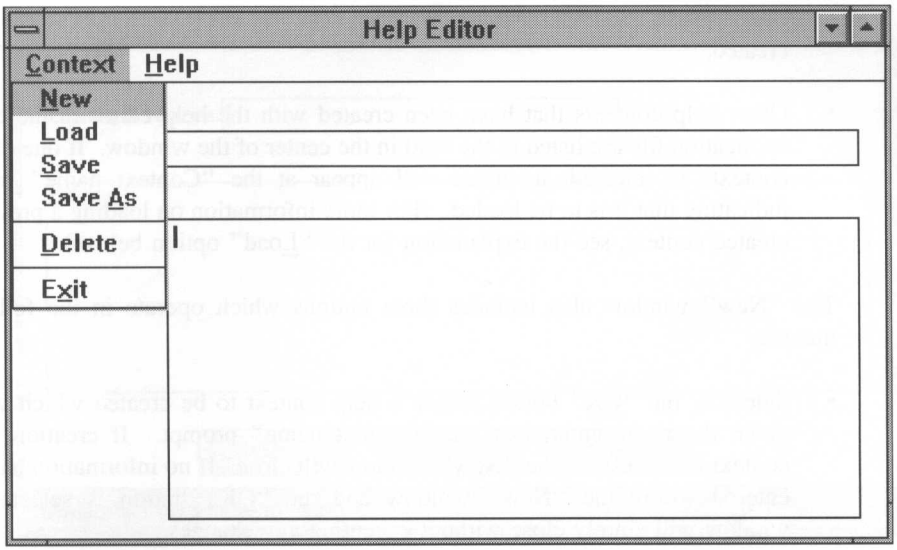
Enter in this field a title to be displayed on the help window's title bar.

message

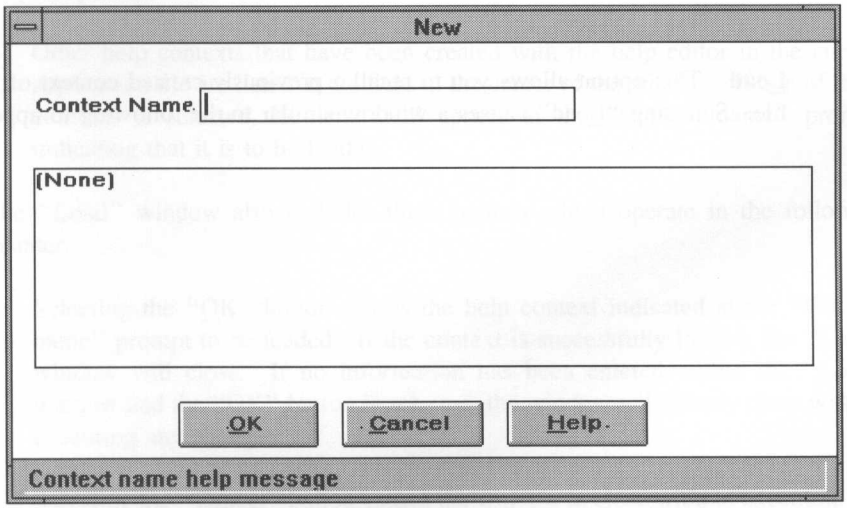
Enter in this field the text to be displayed as help information within the help window.

Context options

The context options of the window's pull-down menu control the general operations of the help editor. Selecting "Context" causes the following menu to appear:



New—This option allows you to create a new help context. Selecting it causes a window similar to the following to appear:



Interaction with the fields of the "New" window is accomplished as follows:

- Enter in the "Context Name" field a name for the new help context to be created.

- Other help contexts that have been created with the help editor in the current application file are listed in the field in the center of the window. If one of these contexts is selected, its name will appear at the "Context name" prompt, indicating that it is to be loaded. (For more information on loading a previously created context, see the explanation for the "Load" option below.)

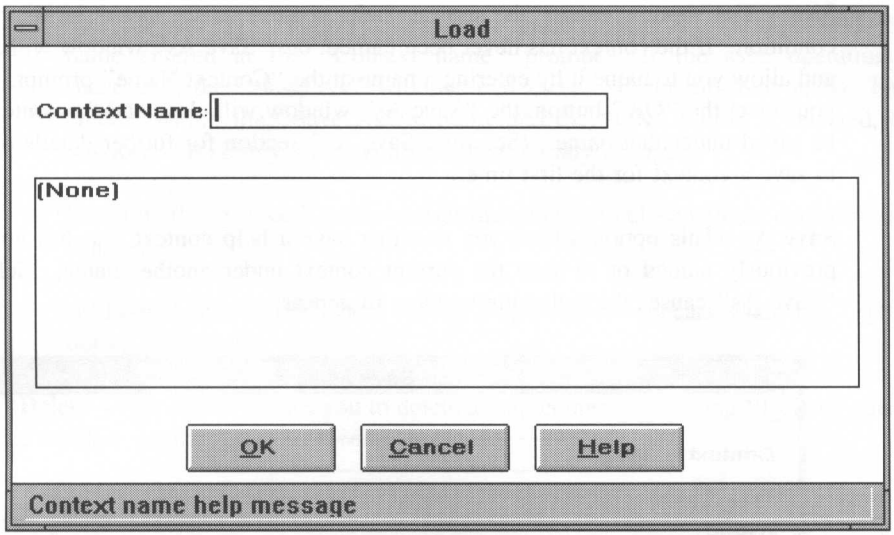
The "New" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes a help context to be created which will be given the name entered at the "Context name" prompt. If creation of the context is successful, the "New" window will close. If no information has been entered within the "New" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about creating new contexts appears when the "Help" button is selected.

Load—This option allows you to recall a previously created context of the current file. Selecting "Load" causes a window similar to the following to appear:



Interaction with the fields of the "Load" window is accomplished as follows:

- Enter in the "Context Name" field the name of the context to be loaded, or select it from the list in the center of the window and the name will appear at this prompt.

- Other help contexts that have been created with the help editor in the current application file are listed in the field in the center of the window. If one of these contexts is selected, its name will appear at the "Context name" prompt, indicating that it is to be loaded.

The "Load" window also includes three buttons which operate in the following manner:

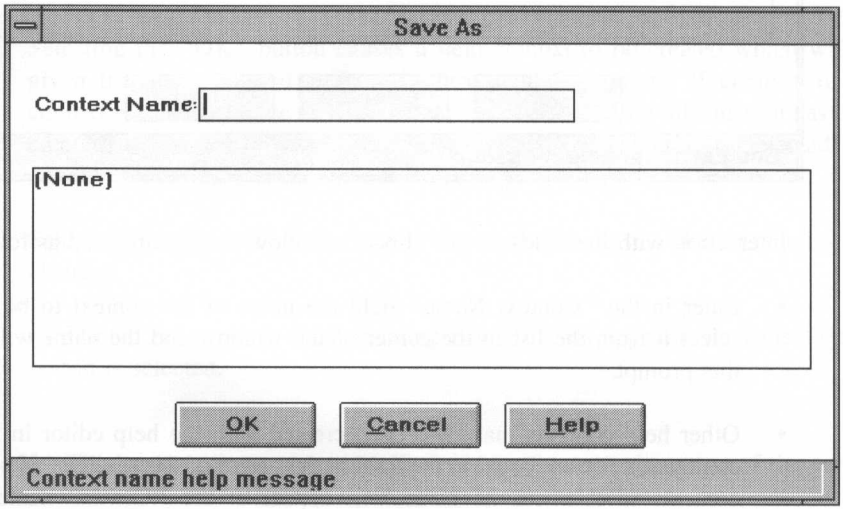
- Selecting the "OK" button causes the help context indicated at the "Context name" prompt to be loaded. If the context is successfully loaded, the "Load" window will close. If no information has been entered within the "Load" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about loading contexts appears when the "Help" button is selected.

Save—This option causes the current help context to be saved in its present condition. If the context has never been named, the "Save As" window will appear and allow you to name it by entering a name at the "Context Name" prompt. When you select the "OK" button, the "Save As" window will close and the context will be saved under that name. (See the "Save As" section for further details on how to save a context for the first time.)

Save As—This option allows you to either save a help context that has not been previously named or to save the current context under another name. Selecting "Save As" causes the following window to appear:



Interaction with the fields of the "Save As" window is accomplished as follows:

- Enter in the "Context Name" field a name for the current context or select one from the list below and the name will appear at this prompt.
- Other help contexts that have been created with the help editor in the current application file are listed in the field in the center of the window. If one of these contexts is selected, its name will appear at the "Context name" prompt, indicating that the current context is to replace the previous information.

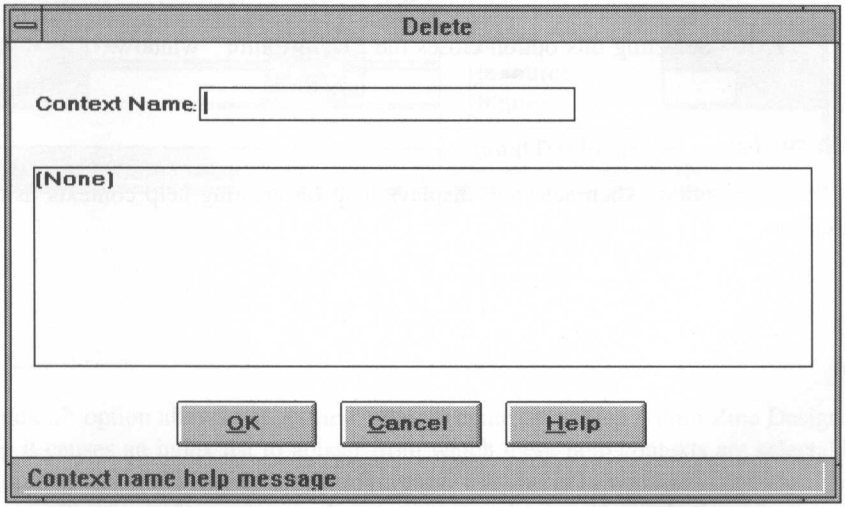
The "Load" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes the current help context to be saved under the name entered at the "Context name" prompt. If the save operation is successful, the "Save As" window will close. If no information has been entered within the "Save As" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about saving help contexts appears when the "Help" button is selected.

Delete—This option allows you to delete a help context. Selecting "Delete" causes a window similar to the following to appear:



Interaction with the fields of the "Delete" window is accomplished as follows:

- Enter in the "Context Name" field the name of the help context to be deleted.

- Other help contexts that have been created with the help editor in the current application file are listed in the field in the center of the window. If one of these images is selected, its name will appear at the "Context name" prompt, indicating that it is to be deleted.

The "Delete" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes the help context designated at the "Context name" prompt to be deleted. If the context is successfully deleted, the "Delete" window will close. If no information has been entered within the "New" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about deleting contexts appears when the "Help" button is selected.

Exit—Selecting this option closes the "Help Editor" window.

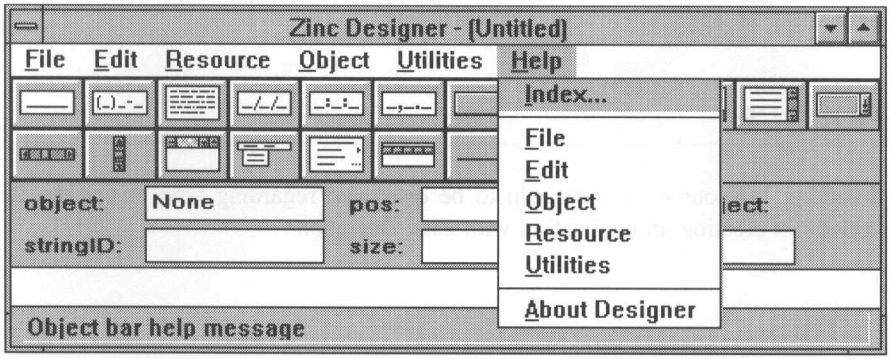
Help option

The help option, when selected, displays help on creating help contexts using the help editor.

CHAPTER 17 - HELP OPTIONS

The Help category is available so that you can receive help at any time during Zinc Designer's execution. The various options represent the different areas within Zinc Designer where help information is available.

Selecting "Help" causes the following menu to appear:



INDEX

The "Index..." option allows you to view all help contexts created within Zinc Designer. Selecting it causes an index list to appear from which these help contexts are selectable. When you select a specific help context from the list, the help window associated with it appears. Pressing <Esc> causes the window to close, and another option can be selected from the index list.

FILE

Selecting "File" causes help to be displayed regarding how to use the File options in creating an application with Zinc Designer.

EDIT

Selecting "Edit" causes help to be displayed regarding how to use the Edit options in creating an application with Zinc Designer.

OBJECT

Selecting "Object" causes help to be displayed regarding how to use the Object options in creating an application with Zinc Designer.

RESOURCE

Selecting "Resource" causes help to be displayed regarding how to use the Resource options in creating an application with Zinc Designer.

UTILITIES

Selecting "Utilities" causes help to be displayed regarding how to use the Utilities options in creating an application with Zinc Designer.

ABOUT DESIGNER

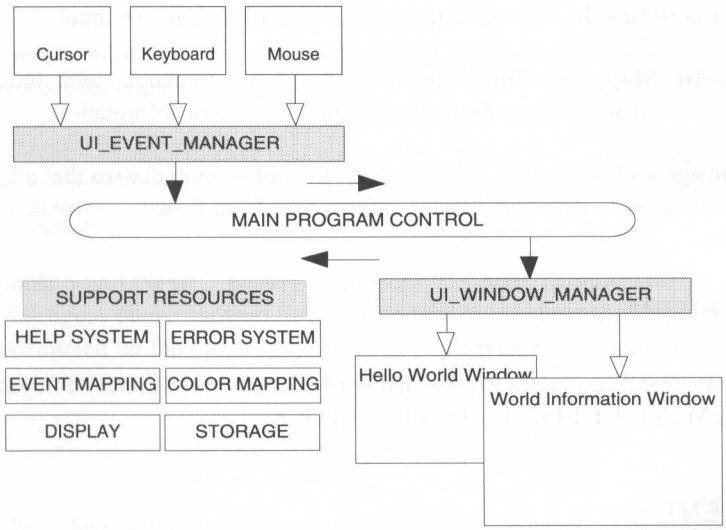
Selecting "About Designer" causes information to be displayed regarding the general contents and specifics of Zinc Designer (e.g., the current version number and copyright information).

SECTION IV
ADVANCED CONCEPTS

CHAPTER 18 – ZINC LIBRARY CLASSES

The purpose of the first two chapters in this section is to get you familiar with the class objects and C++ features used throughout Zinc Interface Library. This section examines the library from a conceptual level, whereas all other sections show you how to create actual applications using the library.

Most of the information contained in this section is based on the concepts illustrated by the general Zinc Interface Library model:



These concepts, as well as others that are fundamental to the operation of Zinc Interface Library, will be discussed in this chapter. They include:

Base Elements—These are the core classes used within Zinc Interface Library. This core consists of three classes that support the concepts of lists, list elements, and list blocks (an array of list elements).

Event Manager—This group of classes consists of input devices—such as the keyboard, mouse, and cursor—the event manager, and support classes used by the event manager and input devices.

Window Manager—This group consists of all window objects—such as buttons, title bars, and text—the window manager, and support classes used by the window manager and window objects.

Help System—This class uses a presentation window to show context-sensitive help.

Error System—This class uses a presentation window to show run-time errors.

Screen Display—These classes support the low-level screen output, which includes the management of screen regions on the display.

Event Mapping—This consists of a set of structures, with an accompanying function, that performs the logical mapping of keyboard and mouse input.

Palette Mapping—This consists of a set of structures, with an accompanying function, that performs the logical mapping of color information.

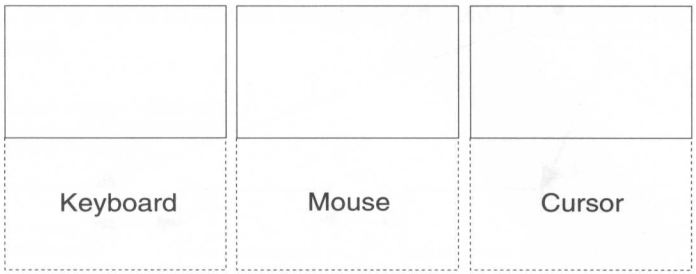
Storage and Retrieval—This contains a set of support classes that allow you to store information to disk and retrieve information from disk.

Base Classes

There are three base classes from which most Zinc Interface Library components are built: UI_ELEMENT, UI_LIST and UI_LIST_BLOCK.

UI_ELEMENT

The UI_ELEMENT class serves as the base class to input devices and window objects. It allows derived class objects to be grouped together in a list, even though their internal definitions and operations may be different. Classes derived from the UI_ELEMENT base class can be viewed in the following manner:



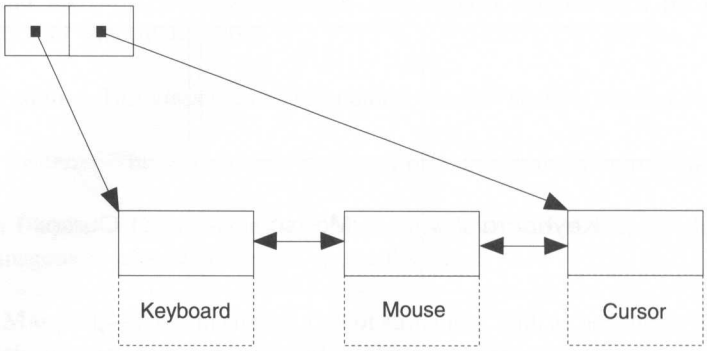
The two major elements associated with the UI_ELEMENT class are its **Previous( )** and **Next( )** member functions.

```
class EXPORT UI_ELEMENT
{
public:
    UI_ELEMENT *Next(void);
    UI_ELEMENT *Previous(void);
```

The **Previous( )** and **Next( )** member functions are used to move within a list. For example, the figure above showed three input devices, keyboard, mouse and cursor. If we were positioned on the mouse object, a call to **Previous( )** would return a pointer to the keyboard object, whereas a call to **Next( )** returns a pointer to the cursor object.

UI_LIST

The UI_LIST class is used to group a set of related elements. The following picture shows how elements derived from the UI_ELEMENT base class can be linked together in a list:



There are several major elements associated with the UI_LIST class: its **First( )**, **Next( )**, **Add( )**, and **Subtract( )** member functions, along with its + and – operator overloads.

```
class EXPORT UI_LIST
{
    friend class EXPORT UI_LIST_BLOCK;

public:
    // Members described in UI_LIST reference chapter.
    UI_LIST(int (*_compareFunction)(void *element1, void *element2) =
        NULL) : first(NULL), last(NULL), current(NULL),
        compareFunction(_compareFunction);
    virtual ~UI_LIST(void);
    UI_ELEMENT *Add(UI_ELEMENT *newElement);
    UI_ELEMENT *Add(UI_ELEMENT *element, UI_ELEMENT *newElement);
    int Count(void);
    UI_ELEMENT *Current(void);
    virtual void Destroy(void);
    UI_ELEMENT *First(void);
    UI_ELEMENT *Get(int index);
    UI_ELEMENT *Get(int (*findFunction)(void *element1, void *matchData),
        void *matchData);
    int Index(UI_ELEMENT const *element);
    UI_ELEMENT *Last(void);
    void SetCurrent(UI_ELEMENT *element);
    void Sort(void);
    UI_ELEMENT *Subtract(UI_ELEMENT *element);
    UI_LIST &operator+(UI_ELEMENT *element);
    UI_LIST &operator-(UI_ELEMENT *element);

protected:
    UI_ELEMENT *first, *last, *current;
}
```

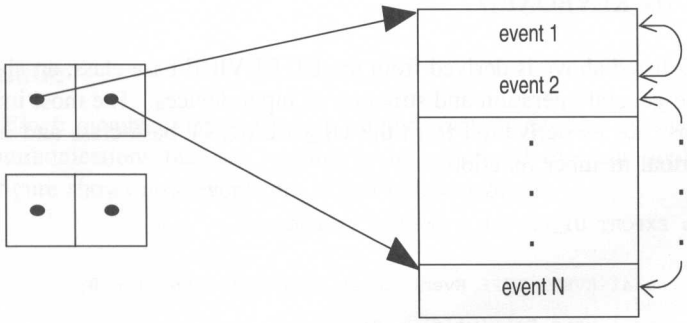
The **First( )** and **Last( )** member functions are used to get the first or last list element. For example, if you were to call **First( )** with the list shown above, a pointer to the keyboard object would be returned. A call to **Last( )** however, would result in a pointer to the cursor object being returned.

The **Add( )** and **Subtract( )** member functions, along with the **+** and **–** operator overloads are used to add or subtract list elements to and from the current list object. For example, the list figure above could be created using either the **Add( )** member function or the **+** operator overload.

```
eventManager->Add(keyboard);
eventManager->Add(mouse);
eventManager->Add(cursor);

    or

*eventManager
    + keyboard
    + mouse
    + cursor;
```

UI_LIST_BLOCK

The UI_LIST_BLOCK class works just like a list, but it uses the memory efficiencies of an array by not only keeping pointers to objects in use (the box shown as *list* in the figure below), but also by maintaining a list of free list elements (the box shown as *freeList*). The following figure shows how the UI_QUEUE_BLOCK class stores event information in the form of a list block:



These three element and list base classes, along with their conceptual figures, are used in the library documentation to aid in the presentation of design concepts.

Event Manager

The event manager class (UI_EVENT_MANAGER) serves as the message center for all internal Zinc communication as well as for all user-entered input information. There are two major elements associated with this class—its public UI_LIST derivation and its *queueBlock* member variable.

```
class EXPORT UI_EVENT_MANAGER : public UI_LIST
{
protected:
    UI_QUEUE_BLOCK queueBlock;
```

Input devices

The UI_LIST part of the event manager contains a programmer-specified set of input devices that feed user-input into your application. These devices can either be polled devices, such as a keyboard, or interrupt driven devices, such as a mouse.

The following device classes are defined by Zinc Interface Library:

UID_CURSOR
UID_MOUSE
UID_KEYBOARD

Each class listed above is derived from the UI_DEVICE base class, an abstract class that defines the general operation and structure of input devices. The most important aspects of this class are its derivation from the UI_ELEMENT base class and its **Event( )** and **Poll( )** virtual member functions.

```
class EXPORT UI_DEVICE : public UI_ELEMENT
{
public:
    virtual EVENT_TYPE Event(const UI_EVENT &event) = 0;
protected:
    virtual void Poll(void) = 0;
```

The class derivation from UI_ELEMENT allows input devices to be added to the event manager's list of input devices. The following code shows how three input devices (keyboard, mouse, and cursor) can all be added to the event manager's list of input devices:

```
// Initialize the event manager and add three devices to it.
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
*eventManager
    + new UID_KEYBOARD
```

```
+ new UID_MOUSE
+ new UID_CURSOR;
```

The **Event( )** function is used to communicate changes to a device's mode of operation. The type of message is contained in *event.rawCode* for the DOS version and in *event.message* for the MS Windows version. Here are some sample messages that can be sent to input devices:

D_OFF—Tells the device to stop feeding input information into the event manager's input queue. No further input information will be received until a D_ON message is received.

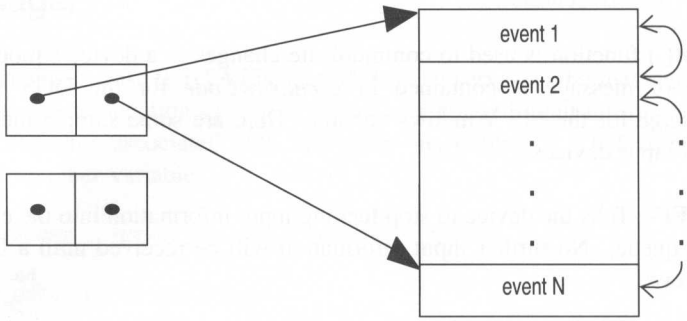
D_POSITION—Changes the position of a device. For example, if the device receiving this message were a cursor, the position of the blinking cursor would be changed to the screen position given by *event.position*.

DM_WAIT—Changes the mouse pointer to be an hour-glass. This message is only understood by the UID_MOUSE class.

The **Poll( )** function allows each device to feed input information to the event manager 's input queue. For example, the UID_KEYBOARD class uses the **Poll( )** routine to look at the keyboard BIOS to see if any keys have been pressed by the user. If a key has been pressed, the **UID_KEYBOARD::Poll( )** routine removes the key from the keyboard BIOS and places it in the event manager's input queue.

The input queue

The *queueBlock* member variable stores all information entered by the user and all internal communications that are waiting to be processed by your application. The following figure shows how events are stored by *queueBlock*:



There are three major components to the input queue: the UI_EVENT structure, the UI_-QUEUE_ELEMENT class, and the UI_QUEUE_BLOCK class.

The UI_EVENT structure contains the actual input information. The definition of this structure is:

```
struct UI_EVENT
{
    EVENT_TYPE type;                    // The type of event.
    union
    {
        MSG message;
        RAW_CODE rawCode;
    };
    union
    {
        UI_KEY key;
        UI_REGION region;
        UI_POSITION position;
        UI_SCROLL_INFORMATION scroll;
        void *data;
    };
};
```

The type of information contained in this structure depends on the type of class object that generates the message. For example, the UID_KEYBOARD class sets the following event information:

- *event.type* always contains the value E_KEY. This lets all receiving objects know that *event.key* contains any related keyboard information.

- *event.rawCode* contains the keyboard's raw scan-code.

- *event.key* contains other keyboard information (i.e., the shift state and low-eight bits of the scan-code).

The UI_QUEUE_ELEMENT and UI_QUEUE_BLOCK classes store event information in a list block. The UI_QUEUE_ELEMENT class is derived from UI_ELEMENT and contains the actual event information:

```
class EXPORT UI_QUEUE_ELEMENT : public UI_ELEMENT
{
public:
    UI_EVENT event;
```

The UI_QUEUE_BLOCK class is derived from UI_LIST_BLOCK and is used to store UI_QUEUE_ELEMENT class objects. The use of these classes allows the input queue to buffer event information before it is processed within your application. The use of a list block keeps the library from allocating and destroying memory every time it receives or dispatches a message.

Window Manager

The window manager class (UI_WINDOW_MANAGER) controls the presentation and operation of all windows and window objects that are displayed on the screen. There are two major elements associated with this class—its public UIW_WINDOW derivation and its virtual **Event( )** member function.

```
class EXPORT UI_WINDOW_MANAGER : public UIW_WINDOW
{
public:
    virtual EVENT_TYPE Event(const UI_EVENT &event);
```

Window objects

The UIW_WINDOW part of the window manager contains the set of windows currently active on the screen. Any window object can be attached to the screen, but normally only window class objects (UIW_WINDOW) are attached.

The following window objects are defined by Zinc Interface Library:

UIW_BIGNUM	**UIW_POP_UP_MENU**
UIW_BORDER	**UIW_PROMPT**
UIW_BUTTON	**UIW_PULL_DOWN_ITEM**
UIW_COMBO_BOX	**UIW_PULL_DOWN_MENU**
UIW_DATE	**UIW_REAL**
UIW_FORMATTED_STRING	**UIW_SCROLL_BAR**
UIW_GROUP	**UIW_STRING**
UIW_HZ_LIST	**UIW_SYSTEM_BUTTON**

UIW_ICON	**UIW_TEXT**
UIW_INTEGER	**UIW_TIME**
UIW_MAXIMIZE_BUTTON	**UIW_TITLE**
UIW_MINIMIZE_BUTTON	**UIW_TOOL_BAR**
UIW_POP_UP_ITEM	**UIW_VT_LIST**
	UIW_WINDOW

Each class listed above is derived from the UI_WINDOW_OBJECT base class. This class defines the general operation and structure of window objects. The most important aspects of this class are its derivation from the UI_ELEMENT base class and its **Event( )** virtual member function.

```
class EXPORT UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    virtual EVENT_TYPE Event(const UI_EVENT &event);
```

The class derivation from UI_ELEMENT allows window objects to be added to the window manager's list of window objects.

```
// Initialize the window manager and add two windows to it.
UI_WINDOW_MANAGER *windowManger = new UI_WINDOW_MANAGER(display,
    eventManager);
UIW_WINDOW *window1 = new UIW_WINDOW(0, 0, 40, 10);
.
.
.
UIW_WINDOW *window2 = new UIW_WINDOW(5, 5, 40, 10);
.
.
.
*windowManager
    + window1
    + window2;
```

The **Event( )** function is used to send logical or system information to a specific window. Here are some sample messages that can be interpreted by window objects:

S_CREATE—Tells the window object to initialize its internal information, such as its size and position within a parent window. The S_CREATE message is always succeeded by an S_CURRENT, S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE message. The S_CREATE message is sent to all of the window objects associated with a window whenever the window is attached to the window manager.

S_DISPLAY_ACTIVE—Tells the window object to display itself according to an active state. The compliment message is S_DISPLAY_INACTIVE.

L_BEGIN_SELECT—Begins the selection process of a window or window object. For example, if the end user presses the left mouse button, the selection of an object

is initiated. When the mouse button is released (L_END_SELECT), the selection process is completed.

Event member functions

The **Event( )** member function dispatches run-time event information from the event manager to windows. For example, if an application were running with two overlapping windows, the window manager would automatically route normal event information to the top window, but pass a mouse click to the window affected by the mouse's position.

There are two types of messages that the window manager and window objects understand:

Logical Events—These events are generated by input devices, then interpreted by the receiving object. For example, a mouse click inside a window would be interpreted as L_BEGIN_SELECT (i.e., begin a selection process); whereas the same mouse click inside a text field would be interpreted as L_BEGIN_MARK (i.e., begin marking a region of the text).

System Events—These events are generated by the window manager, or by window objects as the result of a previous event. For example, the title object generates an S_MOVE message when the user presses the left mouse button inside its border. This message is later received by the window manager, which in turn moves the window on the screen.

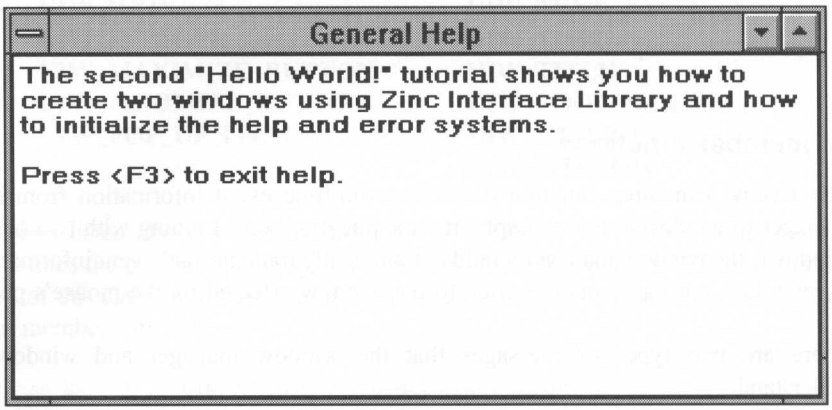
Help System

The help system is used to give end users help during an application. It brings up a help window whenever help is requested.

The help system contains one important virtual function, **DisplayHelp( )**.

```
class EXPORT UI_HELP_SYSTEM
{
public:
    .
    .
    .
    virtual void DisplayHelp(UI_WINDOW_MANAGER *windowManager,
        UI_HELP_CONTEXT helpContext = NO_HELP_CONTEXT);
```

A UIW_WINDOW is used to present information to the screen. A picture of this window is shown below:



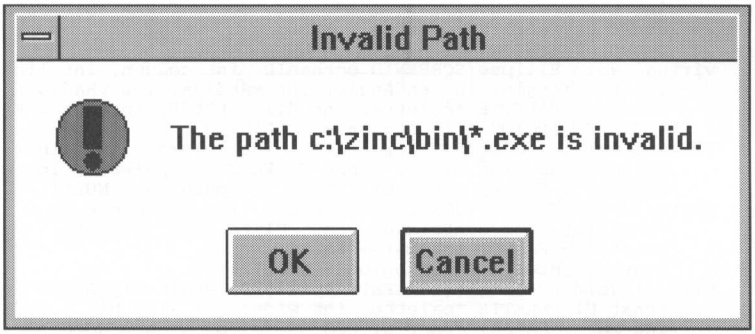
The help window system's **DisplayHelp( )** member function provides context sensitive help information during an application. Each help context contains a title (shown on the title bar), and a help message (shown in the text portion of the window). The *helpContext* argument is used as an identifier to a unique title/message pair.

Error System

The error system brings up a window to display error information to end users whenever an error is detected. The error system contains one important virtual function, **ReportError( )**.

```
class EXPORT UI_ERROR_SYSTEM
{
public:
    virtual UIS_STATUS ReportError(UI_WINDOW_MANAGER *windowManager,
        UIS_STATUS errorStatus, char *format, ...);
```

A UIW_WINDOW is used to present error information to the screen. A picture of this window is shown below (where an invalid pathname was entered within an application):



The error system's **ReportError( )** member function is used to display information about the type of error encountered during an application. This function takes **printf( )** style arguments that are used in the text portion of the window.

Screen Displays

Screen display classes are used to present information to the screen. The following displays are defined by Zinc Interface Library:

UI_BGI_DISPLAY
UI_FG_DISPLAY
UI_MSC_DISPLAY
UI_MSWINDOWS_DISPLAY
UI_TEXT_DISPLAY

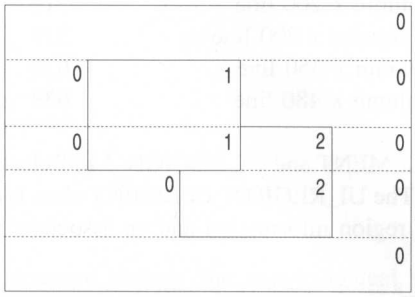
Each class listed above is derived from the UI_DISPLAY base class. This class defines the general operation and structure of display classes. The most important aspects of this class are its derivation from the UI_REGION_LIST base class and the virtual member functions that perform various drawing operations.

```
class EXPORT UI_DISPLAY : public UI_REGION_LIST
{
public:
    // Members described in UI_DISPLAY reference chapter.
    int installed;
    const int isText;
    int cellWidth;
    int cellHeight;
    int columns;
    int lines;
    USHORT onCursorValue;
    USHORT offCursorValue;
    UI_EVENT_MANAGER *eventManager;

    virtual ~UI_DISPLAY(void);
    virtual void Bitmap(SCREENID screenID, int column, int line,
```

```
        int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette = NULL,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual void Ellipse(SCREENID screenID, int column, int line,
        int startAngle, int endAngle, int xRadius, int yRadius,
        const UI_PALETTE *palette, int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL) = 0;
    virtual void Polygon(SCREENID screenID, int numPoints,
        const int *polygonPoints, const UI_PALETTE *palette,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual void Rectangle(SCREENID screenID, const UI_REGION &region,
        const UI_PALETTE *palette, int width = 1, int fill = FALSE,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion) = 0;
    virtual void RegionConvert(UI_REGION &region, USHORT *oldFlags,
        USHORT newFlags) = 0;
    virtual void RegionDefine(SCREENID screenID, const UI_REGION &region);
    virtual void RegionDefine(SCREENID screenID, int left, int top,
        int right, int bottom);
    virtual void RegionMove(const UI_REGION &oldRegion, int newColumn,
        int newLine, SCREENID oldScreenID = ID_SCREEN,
        SCREENID newScreenID = ID_SCREEN) = 0;
    virtual void Text(SCREENID screenID, int left, int top,
        const char *text, const UI_PALETTE *palette, int length = -1,
        int fill = TRUE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN) = 0;
    virtual int TextWidth(const char *string, SCREENID screenID =
        ID_SCREEN) = 0;

protected:
    // Members described in UI_DISPLAY reference chapter.
    UI_DISPLAY(int isText, int cellWidth, int cellHeight);
};
```

Region lists

The class derivation from UI_REGION_LIST allows the display to keep track of regions on the screen. Whenever an object is placed on the screen, a region is reserved so that the object can paint information. As objects are placed on the screen, the regions are split up to allow the painting of background regions without destroying the presentation of higher level objects. This process actually performs the "clipping" of screen regions according to an object's identification. For example, the following picture shows how a screen may be split when two windows are attached to it. The figure shows the region list equivalent where the screen background is represented by the 0 values and the two windows are represented by the values 1 and 2.



Region lists have three main components: a UI_REGION structure, UI_REGION_-ELEMENT class objects, and a UI_REGION_LIST class.

The UI_REGION structure contains the actual region to reserve. The definition of this structure is:

```
struct EXPORT UI_REGION
{
public:
    // Members described in UI_REGION reference chapter.
    int left, top, right, bottom;

#ifdef _WINDOWS
    void Assign(const RECT &rect);
#endif

    int Encompassed(const UI_REGION &region);
    int Overlap(const UI_REGION &region);
    int Overlap(const UI_POSITION &position);
    int Touching(const UI_POSITION *position);
    int Overlap(const UI_REGION &region, UI_REGION &result);
    int operator==(const UI_REGION &region);
    int operator!=(const UI_REGION &region);
    UI_REGION &operator++(void);
    UI_REGION &operator--(void);
    UI_REGION &operator+=(int offset);
    UI_REGION &operator-=(int offset);
};
```

The screen coordinates are defined according to the mode of operation, with the top-left corner always being the coordinates 0, 0. Here are some sample right-bottom coordinates, based on the type of display mode:

Display mode	Right	Bottom
Text 80 column x 25 line	79	24
Text 40 column x 25 line	39	24
Text 80 column x 43 line	79	42
Text 80 column x 50 line	79	49

CGA 320 column x 200 line	319	199
MCGA 320 column x 200 line	319	199
EGA 640 column x 350 line	639	349
VGA 640 column x 480 line	639	479

The UI_REGION_ELEMENT and UI_REGION_LIST classes are used to store the region information in a list. The UI_REGION_ELEMENT class is derived from UI_ELEMENT. It contains the actual region information and an associated screen identification:

```
class EXPORT UI_REGION_ELEMENT : public UI_ELEMENT
{
public:
    SCREENID screenID;
    UI_REGION region;
```

The first time a window is attached to the window manager it is assigned a unique value that is stored in its *screenID* member variable. In addition, the screen is re-defined to contain the window's region. This area is represented by a new UI_REGION_-ELEMENT, where *screenID* is assigned the same value as the window's screen identification, and *region* is assigned the same area occupied by the window. The *region* variable is used later by display functions to clip the boundaries of an object before any screen painting is performed. For example, if two windows were attached to the screen and information were painted to the background window, the background information would be clipped so that the painted regions would not overlap the front window.

Virtual display functions

Virtual display member functions are used to define an abstract method of drawing information to the screen. For example, all display classes have the **Rectangle( )** member function. In text mode, a rectangle is drawn with either a single or a double line. In graphics mode, however, the same routine draws a single or double pixel rectangle. This abstract method of drawing is the key to creating single source applications that run in DOS text, DOS graphics, and MS Windows screen modes.

Event Mapping

There are two structures used by the **MapEvent( )** function to convert raw input information into logical messages: UI_EVENT and UI_EVENT_MAP.

The UI_EVENT structure was discussed earlier in this chapter. It contains the raw event information entered by users during an application.

```
struct EXPORT UI_EVENT
{
    EVENT_TYPE type;
    union
    {
        MSG message;
        RAW_CODE rawCode;
    }
    .
    .
    .
};
```

The UI_EVENT_MAP structure defines the raw-to-logical mapping of events. Its definition is shown below:

```
struct EXPORT UI_EVENT_MAP
{
    OBJECTID objectID;
    LOGICAL_EVENT logicalValue;
    EVENT_TYPE eventType;
    RAW_CODE rawCode;

    static LOGICAL_EVENT MapEvent(UI_EVENT_MAP *mapTable,
        const UI_EVENT &event,
        OBJECTID id1 = ID_WINDOW_OBJECT, OBJECTID id2 = ID_WINDOW_OBJECT,
        OBJECTID id3 = ID_WINDOW_OBJECT, OBJECTID id4 = ID_WINDOW_OBJECT,
        OBJECTID id5 = ID_WINDOW_OBJECT);
};
```

Whenever an event is received from the system, it is interpreted by the receiving object using **UI_WINDOW_OBJECT::LogicalEvent( )**. **LogicalEvent( )** calls **MapEvent( )** using a specified *mapTable* to match a logical value. If *event.type* and *event.rawCode* match a particular map-entry's *eventType* and *rawCode*, the entry's *logicalValue* is returned.

NOTE: There are two pre-defined event map tables used in Zinc Interface Library: *eventMapTable* and *hotKeyMapTable*. *eventMapTable* is used by all UI_WINDOW-_OBJECT class objects and the window manager to determine the logical interpretation of raw events. *hotKeyMapTable* is used by all high-level windows to determine sub-object hot key equivalents. It is only used when an <Alt> key is pressed to get the logical interpretation of the hot key.

Palette Mapping

There are two structures used by the **MapPalette( )** function to provide color palette information for window objects: UI_PALETTE and UI_PALETTE_MAP.

The UI_PALETTE structure contains the color combinations for text and graphic displays.

```
struct EXPORT UI_PALETTE
{
    // --- Text mode ---
    UCHAR fillCharacter;                // Fill character.
    COLOR colorAttribute;               // Color attribute.
    COLOR monoAttribute;                // Mono attribute.

    // --- Graphics mode ---
    LOGICAL_PATTERN fillPattern;        // Fill pattern.
    COLOR colorForeground;              // EGA/VGA colors.
    COLOR colorBackground;
    COLOR bwForeground;                 // Black & White colors (2 color).
    COLOR bwBackground;
    COLOR grayScaleForeground;          // Monochrome colors (3+ color).
    COLOR grayScaleBackground;
};
```

The UI_PALETTE_MAP structure defines the raw-to-logical mapping of palettes. Its definition is shown below:

```
struct UI_PALETTE_MAP
{
    OBJECTID objectID;
    LOGICAL_PALETTE logicalPalette;
    UI_PALETTE palette;

    static UI_PALETTE *MapEvent(UI_PALETTE_MAP *mapTable,
        LOGICAL_PALETTE logicalPalette, OBJECTID id1 = ID_WINDOW_OBJECT,
        OBJECTID id2 = ID_WINDOW_OBJECT, OBJECTID id3 = ID_WINDOW_OBJECT,
        OBJECTID id4 = ID_WINDOW_OBJECT, OBJECTID id5 = ID_WINDOW_OBJECT);
};
```

Whenever a window object paints information to the screen, it gets the color palette using **UI_WINDOW_OBJECT::MapPalette( )**. **MapPalette( )** uses a specified *mapTable* to match a logical value with a palette. If the logical value matches a particular map-entry's *logicalValue*, the entry's *palette* is returned.

NOTE: There are three pre-defined palette map tables used in Zinc Interface Library: *normalPaletteMapTable*, *helpPaletteMapTable*, and *errorPaletteMapTable*. *normalPaletteMapTable* is used by all normal window objects. *helpPaletteMapTable* is used by the UI_HELP_SYSTEM window and *errorPaletteMapTable* is used by the UI_ERROR_SYSTEM window.

CHAPTER 19 – C++ FEATURES

In this chapter we will look at Zinc Interface Library to examine the many C++ features used to make the product powerful and easy to use. The following general concepts are discussed:

1—The way Zinc defines class objects. This includes the definition of basic C++ classes, derived classes, abstract classes, and friend classes.

2—Several methods used to construct class objects. These methods include using the **new** operator, having the constructor called automatically when the scope of a class object is reached, and the various types of construction that are used by Zinc Interface Library.

3—The types of delete operations that can be used to destroy a class object. These methods correspond to the construction methods described, as well as some implementation details Zinc Interface Library uses in conjunction with virtual destructors.

4—Definition of the types of member variables used by Zinc Interface Library. This includes scope variable definition as well as the use of static member variables.

5—Definition of the types of member functions used by Zinc Interface Library. This includes the many features provided by C++ including the use of default arguments, virtual member functions, overloaded functions, pointers to member functions, operator overloads, and static member functions.

Class Definitions

Design issues

Zinc classes are designed to be consistent and easy to understand. This is accomplished, in part, by presenting each class in a similar manner. The following rules apply to all Zinc class definitions:

1—First, all classes have a preceding comment that tells where the class' member functions and static variables can be found. For example, the UI_LIST class has the following lead information.

// ----- UI_LIST ----------
// ----- member functions found in LIST.CPP ----------

class EXPORT UI_LIST

.
.
.

2—Second, all class definitions are preceded by the reserved word **class** and the prefix "UI_". The reserved word class tells the compiler that the definition not only has structural information, but also contain unique information that constitutes a class, such as member functions, single and multiple inheritance, pointers to member functions, etc. The prefix "UI_" is used to indicate a "User Interface" type class. This allows you to have other classes (such as list and list elements) without worrying that your definition conflicts with that used by Zinc Interface Library. Some sample class definitions are given below:

```
class EXPORT UI_ELEMENT
    .
    .
    .
class EXPORT UI_DEVICE : public UI_ELEMENT
    .
    .
    .
class EXPORT UIW_WINDOW : public UI_WINDOW_OBJECT, public UI_LIST
    .
    .
    .
```

3—Third, the order of member access control is first public, then protected and last private. The reason public is defined first is because it is the main part of the clas s you are concerned with. If private members were first, you would have to wade through undocumented variables and functions before you got to the information you really needed.

Public members can be accessed by any other functions and are documented in alphabetical order in the *Programmer's Reference*. Protected members can only be accessed by the class itself, derived class objects and objects that are given the special friend class status. These members are also documented in alphabetical order in the *Programmer's Reference*. Private members can only be accessed by the class itself or by a class granted special friend access. Derived classes cannot access the

private members of another class (unless they are friend classes). Private classes are not documented in any of the Zinc Interface Library manuals.

The UI_DEVICE class shows how this member access order is followed:

```
class EXPORT UID_KEYBOARD : public UI_DEVICE
{
public:
    // Members described in UID_KEYBOARD reference chapter.
    static int breakHandlerSet;

    UID_KEYBOARD(DS_STATE initialState = D_ON);
    virtual ~UID_KEYBOARD(void);
    virtual DS_STATE Event(const UI_EVENT &event);

protected:
    // Members described in UID_KEYBOARD reference chapter.
    int enhancedBIOS;

    virtual void Poll(void);

private:
    // Private members are not documented.
    unsigned char breakState;

#ifndef _WINDOWS
    int EnhancedBios(void);
    void KeyGet(UI_EVENT &event);
    int KeyWaiting(void);
#endif
};
```

4—Finally, member variables and functions are placed in separate logical groups. Member variables are grouped according to a logical order that may consist of byte boundary alignment, first use, most common usage, or a number of other factors. Member functions however, are organized in alphabetical order with the constructor and destructor being placed first. The UIW_BUTTON class shows how this grouping is accomplished.

```
class EXPORT UIW_BUTTON : public UI_WINDOW_OBJECT
{
public:
    // Members described in UIW_BUTTON reference chapter.
    USHORT btFlags;
    USHORT value;
    void (*userFunction)(void *object, UI_EVENT &event);

    UIW_BUTTON(int left, int top, int width, char *text,
        BTF_FLAGS btFlags = BTF_NO_TOGGLE | BTF_AUTO_SIZE,
        WOF_FLAGS woFlags = WOF_JUSTIFY_CENTER,
        USER_FUNCTION userFunction = NULL, EVENT_TYPE value = 0,
        char *bitmapName = NULL);
    virtual ~UIW_BUTTON(void);
    const char *DataGet(int stripString = FALSE);
    void DataSet(const char *string);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    .
    .
    .
```

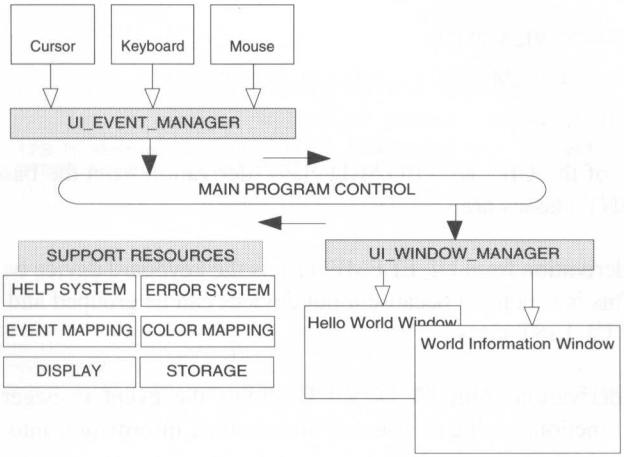
In addition to the class definition rules described above, Zinc Software employees adhere to a full set of internal coding standards, designed to improve the readability and maintenance of code. For a full explanation of these rules see "**Appendix D—Zinc Coding Standards**" of the *Progremmer's Tutorial.*

Base classes

Base classes are used to define the core operation of objects within an application. The core of Zinc Interface Library is contained in two base classes: UI_ELEMENT and UI_-LIST. The definition of these two classes (i.e., their public members) is given below:

```
class EXPORT UI_ELEMENT
{
    friend class EXPORT UI_LIST;

public:
    // Members described in UI_ELEMENT reference chapter.
    virtual ~UI_ELEMENT(void);
    UI_ELEMENT *Next(void);
    UI_ELEMENT *Previous(void);

protected:
    // Members described in UI_ELEMENT reference chapter.
    UI_ELEMENT *previous, *next;

    UI_ELEMENT(void);
};

class EXPORT UI_LIST
{
    friend class EXPORT UI_LIST_BLOCK;

public:
    // Members described in UI_LIST reference chapter.
    UI_LIST(int (*_compareFunction)(void *element1, void *element2) =
        NULL) :     first(NULL), last(NULL), current(NULL),
        compareFunction(_compareFunction);
    virtual ~UI_LIST(void);
    UI_ELEMENT *Add(UI_ELEMENT *newElement);
    UI_ELEMENT *Add(UI_ELEMENT *element, UI_ELEMENT *newElement);
    int Count(void);
    UI_ELEMENT *Current(void);
    virtual void Destroy(void);
    UI_ELEMENT *First(void);
    UI_ELEMENT *Get(int index);
    UI_ELEMENT *Get(int (*findFunction)(void *element1, void *matchData),
        void *matchData);
    int Index(UI_ELEMENT const *element);
    UI_ELEMENT *Last(void);
    void Sort(void);
    UI_ELEMENT *Subtract(UI_ELEMENT *element);
    UI_LIST &operator+(UI_ELEMENT *element);
    UI_LIST &operator-(UI_ELEMENT *element);

protected:
    // Members described in UI_LIST reference chapter.
    UI_ELEMENT *first, *last, *current;
    int (*compareFunction)(void *element1, void *element2);
};
```

If you understand the relationship and use of these two classes, you should be able to understand the underlying design and implementation features of almost all Zinc Interface Library components. For example, you may recall the general Zinc Interface Library model:



The event manager has two main classes: UI_DEVICE and UI_EVENT_MANAGER. The UI_DEVICE class is derived from UI_ELEMENT and is used to define the operation of input devices. Its derivation from UI_ELEMENT allows other classes to be grouped together, in the form of a list. The UI_EVENT_MANAGER class is derived from UI_-LIST. This derivation allows the event manager to control the operation and flow of event information from the input devices.

The window manager has three major classes: UI_WINDOW_OBJECT, UI_WINDOW_-MANAGER, and UIW_WINDOW. The UI_WINDOW_OBJECT class is derived from the UI_ELEMENT base class and also serves as the base class for all window objects (e.g., buttons, icons, menu items). Its derivation from UI_ELEMENT allows derived class objects to be combined in related groups, such as fields inside a parent window. The UI_WINDOW_MANAGER class' derivation from UIW_WINDOW allows it to control the presentation and operation of all window objects attached to the screen. The UIW_-WINDOW class is unique because it exhibits properties of both a list element (when it is attached to the window manager) and a list (as it controls the operation of sub-objects such as the border, title-bar, etc.). Appropriately, this class is derived from both the UI_ELEMENT base class (through the UI_WINDOW_OBJECT class) and the UI_LIST base class.

Derived classes

Derived classes have access to all public and protected members of their base class. Derived classes are useful because they inherit all of the features of their base class and override the features that need to be unique. One example of class inheritance is demonstrated by the UID_KEYBOARD class. Its class hierarchy is shown below:

```
class EXPORT UI_ELEMENT

class EXPORT UI_DEVICE : public UI_ELEMENT

class UID_KEYBOARD : public UI_DEVICE
```

The benefits of the UID_KEYBOARD class' derivation from the base UI_DEVICE and UI_ELEMENT classes are:

1—Its derivation from UI_ELEMENT lets the keyboard device be attached to generic lists. This is beneficial because input devices can be grouped and manipulated by the generic UI_LIST class.

2—Its derivation from UI_DEVICE allows the event manager to call its virtual **Poll( )** function, giving the device time to feed information into the input queue.

The keyboard is unique because it initializes the keyboard BIOS (through its constructor) and feeds keyboard information into the event manager's input queue (through the **Poll( )** member function). These special operations cannot be inherited from any of the two base classes.

Another good example of class inheritance is shown by the UIW_MAXIMIZE_BUTTON and UIW_MINIMIZE_BUTTON classes. These classes exhibit very similar behavior because they appear on the screen in the form of a 3-dimensional button and because they perform their operation (maximizing or minimizing a window) when the mouse button is clicked on them.

Their only differences are that they contain different screen characters (i.e., '▲' for the maximize button, '▼' for the minimize button) and that selecting the maximize button causes a window to be maximized, whereas selecting the minimize button causes the window to be minimized. The actual code difference of these classes is shown below:

```
// Class definition of maximize and minimize buttons.
class EXPORT UIW_MAXIMIZE_BUTTON : public UIW_BUTTON
{
protected:
    EVENT_TYPE MaximizeUserFunction(UI_WINDOW_OBJECT *object,
        UI_EVENT &event, EVENT_TYPE ccode)
        { event.type = S_MAXIMIZE;
            ((UIW_BUTTON *)button)->eventManager->Put(event, Q_BEGIN);
    .
    .
    .
}

class EXPORT UIW_MINIMIZE_BUTTON : public UIW_BUTTON
{
protected:
    EVENT_TYPE MinimizeUserFunction(UI_WINDOW_OBJECT *object,
        UI_EVENT &event, EVENT_TYPE ccode)
        { event.type = S_MINIMIZE;
            ((UIW_BUTTON *)button)->eventManager->Put(event, Q_BEGIN);
    .
    .
    .
}

// Class constructors of maximize and minimize buttons.
UIW_MAXIMIZE_BUTTON::UIW_MAXIMIZE_BUTTON(void) :
    UIW_BUTTON(0, 0, 0, 0, BTF_NO_FLAGS, WOF_JUSTIFY_CENTER |
        WOF_NON_FIELD_REGION, UIW_MAXIMIZE_BUTTON::MaximizeUserFunction)
{
    .
    .
    .
}

UIW_MINIMIZE_BUTTON::UIW_MINIMIZE_BUTTON(void) :
    UIW_BUTTON(0, 0, 0, 0, BTF_NO_FLAGS, WOF_JUSTIFY_CENTER |
        WOF_NON_FIELD_REGION, UIW_MINIMIZE_BUTTON::MinimizeUserFunction)
{
    .
    .
    .
}

EVENT_TYPE UIW_MAXIMIZE_BUTTON::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    EVENT_TYPE ccode = LogicalEvent(event);
    switch (ccode)
    {
    case S_CREATE:
        value = FlagSet(parent->woStatus, WOS_MAXIMIZED) ? S_RESTORE :
            S_MAXIMIZE;
        UI_WINDOW_OBJECT::Event(event);
        if (display->isText)
        {
            text = FlagSet(parent->woStatus, WOS_MAXIMIZED) ? _tMaximized :
                _tNormal;
            true.left = true.right - 2;
            true.bottom = true.top;
        }
```

```
        else
        {
            UCHAR *bitmap = FlagSet(parent->woStatus, WOS_MAXIMIZED) ?
                _gMaximized : _gNormal;
            bitmapWidth = bitmap[0];
            bitmapHeight = bitmap[1];
            bitmapArray = &bitmap[2];
            true.bottom = --true.top + (display->cellHeight -
                display->preSpace - display->postSpace);
            true.left = ++true.right - (display->cellHeight -
                display->preSpace - display->postSpace);
        }
        break;
    .
    .
    .
}

EVENT_TYPE UIW_MINIMIZE_BUTTON::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    EVENT_TYPE ccode = LogicalEvent(event);
    switch (ccode)
    {
    case S_CREATE:
        value = FlagSet(parent->woStatus, WOS_MINIMIZED) ? S_RESTORE :
            S_MINIMIZE;
        UI_WINDOW_OBJECT::Event(event);
        if (display->isText)
        {
            text = FlagSet(parent->woStatus, WOS_MINIMIZED) ? _tMinimized :
                _tNormal;
            true.left = true.right - 2;
            true.bottom = true.top;
        }
        else
        {
            UCHAR *bitmap = _gNormal;
            bitmapWidth = bitmap[0];
            bitmapHeight = bitmap[1];
            bitmapArray = &bitmap[2];
            true.bottom = --true.top + (display->cellHeight -
                display->preSpace - display->postSpace);
            true.left = ++true.right - (display->cellHeight -
                display->preSpace - display->postSpace);
        }
        break;

        .
        .
        .
    }
```

The following chart shows the complete Zinc Interface Library class hierarchy. (Classes that are noted with † have multiple inheritance.)

```
class UI_DISPLAY
      |
      |---- †class UI_BGI_DISPLAY
      |
      |---- †class UI_FG_DISPLAY
      |
      |---- †class UI_TEXT_DISPLAY
      |
      |---- †class UI_MSC_DISPLAY
      |
```

```
        └── class UI_MSWINDOWS_DISPLAY
class UI_ELEMENT
        ├── class UI_DEVICE
                ├── class UID_CURSOR
                ├── class UID_KEYBOARD
                └── class UID_MOUSE
        ├── class UI_PATH_ELEMENT
        ├── class UI_QUEUE_ELEMENT
        ├── class UI_REGION_ELEMENT
        └── class UI_WINDOW_OBJECT
                ├── class UIW_BORDER
                ├── class UIW_BUTTON
                        ├── class UIW_MAXIMIZE_BUTTON
                        ├── class UIW_MINIMIZE_BUTTON
                        ├── class UIW_POP_UP_ITEM
                        ├── class UIW_PULL_DOWN_ITEM
                        ├── class UIW_SYSTEM_BUTTON
                        └── class UIW_TITLE
                ├── class UIW_ICON
                ├── class UIW_PROMPT
                ├── class UIW_STRING
                        ├── class UIW_BIGNUM
                        ├── class UIW_DATE
                        ├── class UIW_FORMATTED_STRING
                        ├── class UIW_INTEGER
                        ├── class UIW_REAL
                        └── class UIW_TIME
                └── †class UIW_WINDOW
                        ├── class UI_WINDOW_MANAGER
                        ├── class UIW_COMBO_BOX
                        ├── class UIW_GROUP
                        ├── class UIW_HZ_LIST
                        ├── class UIW_POP_UP_MENU
                        └── class UIW_PULL_DOWN_MENU
```

```
                                    ├── class UIW_SCROLL_BAR
                                    ├── class UIW_TEXT
                                    ├── class UIW_TOOL_BAR
                                    └── class UIW_VT_LIST

class UI_ERROR_SYSTEM

class UI_HELP_SYSTEM

class UI_INTERNATIONAL
        ├── class UI_BIGNUM
        ├── class UI_DATE
        └── class UI_TIME

class UI_LIST
        ├── class UI_EVENT_MANAGER
        ├── class UI_LIST_BLOCK
        │       └── class UI_QUEUE_BLOCK
        ├── class UI_PATH
        ├── class UI_REGION_LIST
        │       ├── †class UI_BGI_DISPLAY
        │       ├── †class UI_FG_DISPLAY
        │       ├── †class UI_TEXT_DISPLAY
        │       └── †class UI_MSC_DISPLAY
        └── †class UIW_WINDOW

class UI_STORAGE

class UI_STORAGE_OBJECT

struct UI_EVENT

struct UI_EVENT_MAP

struct UI_ITEM

struct UI_KEY

struct UI_PALETTE

struct UI_PALETTE_MAP

struct UI_POSITION

struct UI_REGION

struct UI_SCROLL_INFORMATION
```

† - indicates multiple inheritance

Multiple inheritance classes

Multiple inheritance allows a class to gain access to the protected members of more than one base class. This proves beneficial in the following library classes: UI_BGI_-DISPLAY, UI_FG_DISPLAY, UI_MSC_DISPLAY, UI_TEXT_DISPLAY, and UIW_-WINDOW. The inheritance paths of the preceding classes are shown below:

```
class EXPORT UI_DISPLAY

class EXPORT UI_LIST

class EXPORT UI_REGION_LIST

class EXPORT UI_WINDOW_OBJECT

class EXPORT UI_WINDOW_OBJECT : public UI_ELEMENT

class EXPORT UI_BGI_DISPLAY : public UI_DISPLY, UI_REGION_LIST

class EXPORT UI_FG_DISPLAY : public UI_DISPLY, UI_REGION_LIST

class EXPORT UI_MSC_DISPLAY : public UI_DISPLY, UI_REGION_LIST

class EXPORT UI_TEXT_DISPLAY : public UI_DISPLY, UI_REGION_LIST

class EXPORT UIW_WINDOW : public UI_WINDOW_OBJECT, public UI_LIST
```

The UIW_WINDOW class is derived from the UI_WINDOW_OBJECT base class so that it can be displayed to the screen, like a normal window object, and is derived from the UI_LIST base class so that it can control the presentation and operation of its sub-objects (e.g., buttons, strings, menus).

Each of the display classes is derived from UI_DISPLAY (an abstract class) and UI_REGION_LIST so that the display can manage any attached window objects.

Abstract classes

There are two Zinc Interface Library classes that are considered "abstract": UI_DEVICE and UI_DISPLAY. For a class to be considered abstract, it must have one or more pure virtual functions (i.e., functions that have an **= 0** at the end of their declaration). The device class has two pure virtual functions: **Event( )** and **Poll( )**.

```
class EXPORT UI_DEVICE : public UI_ELEMENT
{
    friend class EXPORT UI_EVENT_MANAGER;
public:
    .
    .
    .
    virtual EVENT_TYPE Event(const UI_EVENT &event) = 0;
```

```
protected:
    .
    .
    .
    virtual void Poll(void) = 0;
};
```

The display class has many pure virtual functions:

```
class EXPORT UI_DISPLAY
{
public:
    .
    .
    .
    virtual void Bitmap(SCREENID screenID, int column, int line,
        int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette = NULL,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual void Ellipse(SCREENID screenID, int column, int line,
        int startAngle, int endAngle, int xRadius, int yRadius,
        const UI_PALETTE *palette, int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL) = 0;
    virtual void Polygon(SCREENID screenID, int numPoints,
        const int *polygonPoints,   const UI_PALETTE *palette,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual void Rectangle(SCREENID screenID, const UI_REGION &region,
        const UI_PALETTE *palette, int width = 1, int fill = FALSE,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
    .
    .
    .
};
```

Abstract classes are beneficial because they let you define the conceptual operation of a class without associating any specific code with the class.

> **NOTE**: The UI_WINDOW_OBJECT looks and in many ways operates like an abstract class, but it is not an abstract class because it has no pure virtual functions.

Friend classes

Friend classes allow a specified class to gain access to the protected and private members of another class. This is useful when you want to hide the implementation details of one class but let a similar or corresponding class have special access rights.

There are many situations where a Zinc Interface Library class grants access rights to another class. The main use of friend classes is with the creation of the designer and will

not be discussed at this time. Of the remaining instances, the first occurs when a class derived from the UI_ELEMENT base class grants "friend" access to its list counterpart. This allows the list to optimize the operation and access of its list elements. The following class relationships show how this is useful:

UI_ELEMENT makes **UI_LIST** a friend class since the lists are continually manipulating and searching through list elements.

UI_LIST makes **UI_LIST_BLOCK** a friend class since it carries the same functionality as the **UI_LIST** except that is used in array form.

UI_DEVICE makes **UI_EVENT_MANAGER** a friend class so that the event manager can set up internal device information (e.g., the *display* and *eventManager* member variables) when the device is attached to the event manager.

UI_STORAGE_OBJECT makes **UI_STORAGE** a friend class since the storage lists are continually manipulating and searching through storage elements.

UI_WINDOW_OBJECT makes both **UI_WINDOW_MANAGER** and **UIW_-WINDOW** friend classes. Friendship rights are granted to the window manager so that it can set up internal window object information (e.g., the *display*, *eventManager* and *windowManager* variables) when the window object is attached to the window manager. The window gets friend access because it works similar to the window manager, in that it controls the operation of sub-objects within its scope (e.g., border, buttons, title bar).

Class Creation

Using the "new" operator

Class objects are created by you, the programmer, using the **new** operator or by specifying the new scope of a class object. Here is some sample code that initializes Zinc Interface Library's screen, event manager and window manager using the **new** operator:

```
#include <ui_win.hpp>

main()
{
    // Initialize the screen.
    UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    .
    .
    .

    // Initialize the event manager.
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
```

```
        .
        .
        .
        // Initialize the window manager.
        UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANGER(display,
            eventManager);
        .
        .
        .
    }
```

The use of the **new** operator lets you initialize a class and maintain its information, even when the scope of a function ends. For example, the second "Hello, World!" tutorial uses functions to initialize two windows. If the **new** operator were not used, the windows would be destroyed when the scope of the function ended.

Scope class construction

The example above showed how the reserved word **new** was explicitly used to construct new classes. The code above could also be written to implicitly call the class constructors:

```
    #include <ui_win.hpp>

    main()
    {
        // Initialize the screen.
        UI_TEXT_DISPLAY display;
        .
        .
        .

        // Initialize the event manager.
        UI_EVENT_MANAGER eventManager(&display);
        .
        .
        .

        // Initialize the window manager.
        UI_WINDOW_MANAGER windowManager(&display, &eventManager);
        .
        .
        .
    }
```

In this case, each object is constructed when the scope of its class is reached.

Base class construction

In addition to the class constructors you use, Zinc Interface Library classes implicitly use inherited class constructors to initialize their information. For example, the UI_TEXT_-DISPLAY class calls the UI_DISPLAY base class constructor and the UI_REGION_LIST class before it initializes any of its own information:

```
UI_TEXT_DISPLAY::UI_TEXT_DISPLAY(TDM_MODE mode) :
    UI_DISPLAY(TRUE, 1, 1),
    UI_REGION_LIST()
{
```

NOTE: In C++, a base class, with no arguments, is automatically initialized whether or not it is specified in the constructor. Although this is legal, these types of base classes are specified throughout Zinc Interface Library in order to make the code more readable. Notice that UI_REGION_LIST was included in the constructor of UI_TEXT_DISPLAY.

The UID_KEYBOARD class uses UI_DEVICE to initialize its base class information:

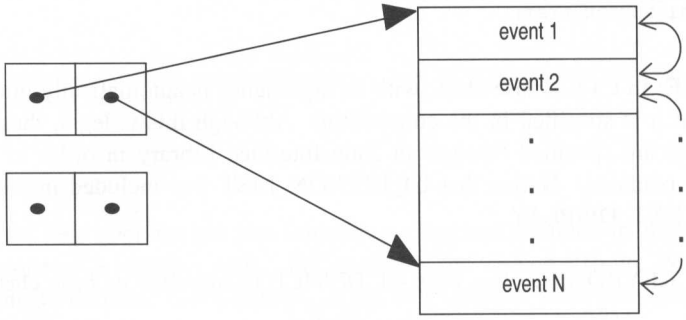
```
UID_KEYBOARD::UID_KEYBOARD(DS_STATE initialState) :
    UI_DEVICE(E_KEY, initialState)
{
```

The UIW_POP_UP_ITEM class not only calls the UIW_BUTTON class for initialization, but the UIW_BUTTON class calls UI_WINDOW_OBJECT for base class initialization. This saves a tremendous amount of code that would be required to initialize each object separately:

```
UIW_BUTTON::UIW_BUTTON(int left, int top, int width, char *_text,
    BTF_FLAGS _btFlags, WOF_FLAGS _woFlags, USER_FUNCTION _userFunction,
    EVENT_TYPE _value, char *_bitmapName) :
    UI_WINDOW_OBJECT(left, top, width, 1, _woFlags, WOAF_NO_FLAGS),
    text(NULL), btFlags(_btFlags), value(_value), depth(2),
    btStatus(BTS_NO_STATUS), bitmapWidth(0), bitmapHeight(0),
    bitmapArray(NULL)
{

UIW_POP_UP_ITEM::UIW_POP_UP_ITEM(void) :
    UIW_BUTTON(0, 0, 1, NULL, BTF_NO_3D, WOF_NO_FLAGS),
    menu(0, 0, WNF_NO_FLAGS, WOF_BORDER, WOAF_TEMPORARY | WOAF_NO_DESTROY),
    mniFlags(MNIF_SEPARATOR)
{
```

Array constructors

The only Zinc class that uses an array constructor is UI_QUEUE_BLOCK. The picture and code below show the design and implementation of event information by the queue-block class.




```
UI_QUEUE_BLOCK::UI_QUEUE_BLOCK(int _noOfElements) :
    UI_LIST_BLOCK(_noOfElements)
{
    // Initialize the queue block.
    UI_QUEUE_ELEMENT *queueBlock = new UI_QUEUE_ELEMENT[_noOfElements];
    elementArray = queueBlock;
    for (int i = 0; i < _noOfElements; i++)
        freeList.Add(NULL, &queueBlock[i]);
}
```

This implementation is useful for the event manager, because it permits us to only perform one big allocation for event information. This prevents us from allocating event information each time it comes into the system, then deallocating the information after it has been used.

Overloaded constructors

Overloaded constructors are used to simplify the creation of class objects. For example, the UI_DATE class overloads its constructor in the following manner:

```
class EXPORT UI_DATE
{
    UI_DATE(void):
    UI_DATE(const UI_DATE &date):
    UI_DATE(int year, int month, int day):
    UI_DATE(const char *string, DTF_FLAGS dtFlags = DTF_NO_FLAGS);
```

The various overloaded date constructors allow you to create a date object according to:

- the computer's system date (this method requires no arguments)
- a packed integer value (the year, month and day are represented by bit sections of the integer). This representation is the DOS packed date format.

- a previously created date class object
- three integer values (i.e., the year, month and day)
- an alphanumeric date. Zinc's powerful constructor capability lets you specify an alphanumeric date that can be interpreted in an country-independent fashion.

Each of these constructors is very useful at different points of an application.

All classes derived from UI_WINDOW_OBJECT have at least two overloaded constructors: one, or more, for basic run-time setup, and another for persistent object access (i.e., the constructor that has the *file* argument). For example, the UIW_POP_UP_-ITEM class has the following definitions:

```
class EXPORT UIW_POP_UP_ITEM : public UIW_BUTTON
{
    UIW_POP_UP_ITEM(void);
    UIW_POP_UP_ITEM(char *text, MNIF_FLAGS mniFlags = MNIF_NO_FLAGS,
        BTF_FLAGS btFlags = BTF_NO_3D, WOF_FLAGS woFlags = WOF_NO_FLAGS,
        EVENT_TYPE (*userFunction)(UI_WINDOW_OBJECT *object,
        UI_EVENT &event, EVENT_TYPE ccode) = NULL,  unsigned value = 0);
    UIW_POP_UP_ITEM(int left, int top, int width, char *text,
        MNIF_FLAGS mniFlags = MNIF_NO_FLAGS,
        BTF_FLAGS btFlags = BTF_NO_FLAGS,
        WOF_FLAGS woFlags = WOF_NO_FLAGS, EVENT_TYPE            (*userFunc-
tion)(UI_WINDOW_OBJECT *object, UI_EVENT &event,
        EVENT_TYPE ccode) = NULL, unsigned value = 0);
```

The first constructor is used to provide menu item separators. The second is used when the pop-up item is going to be attached to a parent pop-up menu. The third provides positional information useful when the item is going to be attached directly to a parent window. The last is used to construct the pop-up item from disk information.

Copy constructors

Only two library classes use copy constructors: UI_DATE and UI_TIME. A copy constructor lets you pass a previously created class into the constructor of another class object. An example of the date constructor is shown below:

```
class EXPORT UI_DATE
{
    UI_DATE(void) { DataSet(); }
    UI_DATE(const UI_DATE &date) { DataSet(date); }
    UI_DATE(int year, int month, int day) { DataSet(year, month, day); }
    UI_DATE(const char *string, DTF_FLAGS dtFlags = DTF_NO_FLAGS)
        { DataSet(string, dtFlags); }
```

Default arguments

Default arguments are used in Zinc Interface Library when there is a default mode of

operation that is only occasionally overridden. The use of default arguments allows you to skip argument definitions that you either do not need to worry about, or that you can defer until they are needed for advanced operations. For example, the text display class provides the following default argument:

```
class EXPORT UI_TEXT_DISPLAY : public UI_DISPLAY
{
public:
    UI_TEXT_DISPLAY(int mode = TDM_AUTO);
```

The default argument for *mode* is TDM_AUTO, which constructs the display using the screen's current mode of operation. While this is the standard mode of operation, four additional types of text displays may be created:

TDM_BW_25x40—Forces the screen to be initialized in a 25 line by 40 column black and white mode.

TDM_25x40—Forces the screen to be initialized in a 25 line by 40 column mode.

TDM_BW_25x80—Forces the screen to be initialized in a 25 line by 80 column black and white mode.

TDM_25x80—Forces the screen to be initialized in a 25 line by 80 column mode.

TDM_MONO_25x80—Forces the screen to be initialized in a 25 line by 80 column monochrome mode.

TDM_43x80—Forces the screen to be initialized in a 43 line by 80 column mode. (This is 50 line mode if the a VGA card is being used.)

If you do not want to override the default operation of the text display, you can call the constructor with no arguments:

```
UI_DISPLAY *display = new UI_TEXT_DISPLAY;
```

Otherwise, you can override the default by providing one of the allowed argument values:

```
// Force 43 line mode.
UI_DISPLAY *display = new UI_TEXT_DISPLAY(TDM_43x80);
```

There are many other class functions that contain default information. The *Programmer's Reference* contains information about such default arguments, their use, and the steps required to override their definition.

Class Deletion

Using the "delete" operator

The destruction of classes is either performed by you, if the reserved word **new** was used to create the object, or automatically performed when the scope of a class ends. For example, the initialization code shown earlier would require the following use of **delete**:

```
#include <ui_win.hpp>

main()
{
    // Initialize Zinc Interface Library using the new operator.
    UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANGER(display, eventManager);
    .
    .
    .
    // Restore the system.
    delete windowManager;
    delete eventManager;
    delete display;
}
```

Scope deletion

The use of the **new** operator requires us to use **delete** to destroy the class objects in the first example program. In the example below, however, class destructors are automatically called for the window manager, event manager and display when the scope of **main** ends.

```
main()
{
    // Initialize Zinc Interface Library using implicit constructors.
    UI_DOS_TEXT_DISPLAY display;
    UI_EVENT_MANAGER eventManager(&display);
    UI_WINDOW_MANAGER windowManager(&display, &eventManager);
    .
    .
    .
    // The windowManager, eventManager and display are automatically
    // destroyed when the scope of main ends.
}
```

NOTE: The order of class creation and destruction is important. In general, those objects that you create first, must be destroyed last.

Virtual destructors

Zinc Interface Library uses virtual member functions for specific class objects. The most general use of a virtual function is associated with the UI_ELEMENT and UI_LIST base classes:

```
class EXPORT UI_ELEMENT
{
public:
    .
    .
    .
    virtual ~UI_ELEMENT(void);
};

class EXPORT UI_LIST
{
public:
    .
    .
    .
    virtual ~UI_LIST(void);
};
```

The declaration of virtual destructors is useful because it requires us to call the destructor of the base class, rather than the specific derived object. For example, Zinc Interface Library uses UI_ELEMENT as the base class to all input devices. When we create the keyboard, cursor, and mouse, we are defining three different input devices. In C, we would have to explicitly call a function to free the memory for each type of input, but in C++ with the use of virtual functions, the class destructor is called automatically by the controlling list class:

```
class EXPORT UI_LIST
{
public:
    virtual ~UI_LIST(void) { Destroy(); }
    .
    .
    .
}

void UI_LIST::Destroy(void)
{
    UI_ELEMENT *tElement;

    // Delete all the elements in the list.
    for (UI_ELEMENT *element = first; element; )
    {
        tElement = element;
        element = element->next;
        delete tElement;
    }
    .
    .
    .
}
```

In addition to the delete operation called by programmers, Zinc Interface Library classes explicitly call destructors for derived objects and for objects contained within their scope. For example, if the event manager were constructed with a keyboard, cursor and mouse object, the code would be:

```
// Initialize the event manager and add three devices to it.
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
*eventManager
    + new UID_KEYBOARD
    + new UID_MOUSE
    + new UID_CURSOR;

// Calling the event manager destructor automatically calls the
// destructor for the keyboard, mouse and cursor devices.
delete eventManager;
```

When the destructor is called for the event manager (i.e., *delete eventManager;*) the event manager automatically calls the destructors for the UID_KEYBOARD, UID_MOUSE and UID_CURSOR device classes.

Any class derived from UI_LIST handles the destruction of list elements in a similar manner. Thus, the programmer does not need to worry about the details of destroying internal class information, since the information is automatically destroyed when a higher level object is destroyed.

Base class destruction

Base class destructors are automatically called when a derived class' destructor is called. For example, the UIW_BUTTON class has the following destructor:

```
UIW_BUTTON::~UIW_BUTTON(void)
{
    if (string)
        delete string;
}
```

After the button class destructor is called and executed, C++ automatically calls the destructor of UI_WINDOW_OBJECT, then the destructor for UI_ELEMENT. Thus, destruction of class objects works in an opposite order as class construction (discussed earlier in this chapter).

Array destruction

UI_QUEUE_BLOCK is the only library class that uses an array destructor to delete its queue elements. The code associated with this destruction is shown below.

```
UI_QUEUE_BLOCK::~UI_QUEUE_BLOCK(void)
{
    // Free the queue block.
    UI_QUEUE_ELEMENT *queueBlock = (UI_QUEUE_ELEMENT *)elementArray;
    delete [noOfElements]queueBlock;
}
```

Array destructors should only be used in conjunction with array constructors.

Member Variables

Variable definitions

Zinc member variables always begin with a lower case alphabetic character and are organized according to a logical order, such as byte boundary alignment, first use, most common usage, or a number of other factors. An example of the UI_LIST class, with several member variables is shown below:

```
class EXPORT UI_LIST
{
protected:
    UI_ELEMENT *first, *last, *current;
    int (*compareFunction)(void *element1, void *element2);
```

Most member variables use compiler-defined types such as **int**, **short**, and **long**, but some use a typecast declaration of unsigned values. The following low-level type declarations are used to define these unsigned values:

```
typedef unsigned char UCHAR;
typedef unsigned short USHORT;
typedef unsigned long ULONG;
```

In addition to the types described above, Zinc objects define and use member variables as bitwise flags. An example of this use is seen with the base UI_WINDOW_OBJECT::-*woFlags* member variable:

```
// --- woFlags ---
typedef unsigned WOF_FLAGS;
const WOF_FLAGS WOF_NO_FLAGS              = 0x0000;
const WOF_FLAGS WOF_JUSTIFY_CENTER        = 0x0001;
const WOF_FLAGS WOF_JUSTIFY_RIGHT         = 0x0002;
const WOF_FLAGS WOF_BORDER                = 0x0004;
const WOF_FLAGS WOF_VIEW_ONLY             = 0x0010;
const WOF_FLAGS WOF_NO_UNANSWERED         = 0x0020;
const WOF_FLAGS WOF_NO_INVALID            = 0x0040;
const WOF_FLAGS WOF_UNANSWERED            = 0x0080;
const WOF_FLAGS WOF_INVALID               = 0x0100;
const WOF_FLAGS WOF_NON_FIELD_REGION      = 0x0200;
const WOF_FLAGS WOF_NON_SELECTABLE        = 0x0400;
const WOF_FLAGS WOF_AUTO_CLEAR            = 0x0800;
```

```
class EXPORT UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    WOF_FLAGS woFlags;
```

The base UI_WINDOW_OBJECT class uses the combined bits of *woFlags* (i.e., bits that are OR'd together to form composite values) to determine its mode of operation. See the *Programmer's Reference* for the individual characteristics that each particular flag sets.

Static member variables

Occasionally, classes define static member variables. For example, the UIW_WINDOW class uses a static variable in order to create a generic window:

```
class EXPORT UIW_WINDOW : public UI_WINDOW_OBJECT, public UI_LIST
{
public:
    static UIW_WINDOW *Generic(int left, int top, int width, int height,
        char *title, UI_WINDOW_OBJECT *minObject = NULL,
        WOF_FLAGS woFlags = WOF_NO_FLAGS, WOAF_FLAGS woAdvancedFlags =
        WOAF_NO_FLAGS, UI_HELP_CONTEXT helpContext = NO_HELP_CONTEXT);
```

This definition allows the programmer to make a call to **UIW_WINDOW::Generic()** in order to create a window with a border, title, maximize button, minimize button and system button.

In addition to using static functions, static variables are used to store internal information. For example, the UI_TIME class uses this type of static variable to store its string equivalent of ante- and post-meridian date values:

```
class EXPORT UI_TIME : public UI_INTERNATIONAL
{
public:
    // Members described in UI_TIME reference chapter.
    static char *amPtr;
    static char *pmPtr;
    .
    .
    .
}

char *UI_TIME::amPtr = "a.m.";
char *UI_TIME::pmPtr = "p.m.";
```

NOTE: C++ requires that when static variables are used as part of a class, space must be reserved for them outside of the class definition.

Member Functions

Function definitions

Zinc Interface Library functions always begin with an upper case letter and usually form complete words that are used to describe the function. For example, the UI_ELEMENT class has two member functions—**Next( )** and **Previous( )**:

```
class EXPORT UI_ELEMENT
{
public:
    UI_ELEMENT *Next(void);
    UI_ELEMENT *Previous(void);
```

Default arguments

Member functions use default arguments to automatically set consistent or advanced features of a function. For example, the UI_DISPLAY uses many default arguments to specify advanced display features, such as filling zones and XOR'ing the screen output:

```
class EXPORT UI_DISPLAY
{
public:
    .
    .
    .
    virtual void Bitmap(SCREENID screenID, int column, int line,
        int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette = NULL,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual void Ellipse(SCREENID screenID, int column, int line,
        int startAngle, int endAngle, int xRadius, int yRadius,
        const UI_PALETTE *palette, int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual void ImageGet(SCREENID screenID, const UI_REGION &region,
        void far *buffer) = 0;
    virtual void ImagePut(SCREENID screenID, const UI_REGION &region,
        void far *buffer) = 0;
    virtual unsigned ImageSize(SCREENID screenID,
        const UI_REGION &region) = 0;
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL) = 0;
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground) = 0;
    virtual void Polygon(SCREENID screenID, int numPoints,
        const int *polygonPoints, const UI_PALETTE *palette,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion) = 0;
    virtual void RegionConvert(UI_REGION &region, unsigned *oldFlags,
        unsigned newFlags, int absolute) = 0;
    virtual void RegionDefine(SCREENID screenID, int left, int top,
        int right, int bottom) = 0;
```

```
    virtual void RegionMove(const UI_REGION &oldRegion, int newColumn,
        int newLine, SCREENID oldScreenID = ID_SCREEN,
        SCREENID newScreenID = ID_SCREEN) = 0;
    virtual void Text(SCREENID screenID, int left, int top,
        const char *text, const UI_PALETTE *palette, int length = -1,
        int fill = TRUE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN) = 0;
    virtual int TextWidth(const char *string,
        SCREENID screenID = ID_SCREEN) = 0;
    virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
        int bottom) = 0;
    virtual int VirtualPut(SCREENID screenID) = 0;
    .
    .
    .
};
```

Virtual member functions

Virtual member functions are used to ensure that the bottom-most object's member function is called before any base class member functions are called. For example, the UI_DEVICE class defines virtual **Event( )** and **Poll( )** routines:

```
class EXPORT UI_DEVICE : public UI_ELEMENT
{
public:
    virtual DS_STATE Event(const UI_EVENT &event) = 0;
protected:
    virtual void Poll(void) = 0;
```

Whenever the event manager calls these functions, it wants to communicate with the actual device, not with the abstract UI_DEVICE class. The use of virtual functions allows the event manager to talk to the objects directly. For example, the following event manager code causes the keyboard, mouse, and cursor virtual **Poll** routines to be called:

```
// Initialize the event manager and add three devices to it.
UI_EVENT_MANAGER eventManager(display);
eventManager
    + new UID_KEYBOARD
    + new UID_MOUSE
    + new UID_CURSOR;
.
.
.
```

```
int UI_EVENT_MANAGER::Get(UI_EVENT &event, Q_FLAGS flags)
{
    UI_DEVICE *device;
    UI_QUEUE_ELEMENT *element;
    int error = -1;

    // Stay in loop while no event conditions are met.
    do
    {
        // Call all the polled devices.
        if (!FlagSet(flags, Q_NO_POLL))
        {
#ifdef _WINDOWS
            MSG message;
            if ((!FlagSet(flags, Q_NO_BLOCK) && !queueBlock.First()) ||
                PeekMessage(&message, 0, 0, 0, PM_NOREMOVE))
            {
                GetMessage(&message, 0, 0, 0);
                UI_EVENT event(E_MSWINDOWS, message.hwnd, message.message,
                    message.wParam, message.lParam);
                event.message = message;
                Put(event, Q_BEGIN);
            }
#endif
            for (device = First(); device; device = device->Next())
                device->Poll();
}
```

The UI_WINDOW_OBJECT class defines virtual **Event( )**, **Information( )**, **Editor( )**, and **Store( )** functions.

```
class EXPORT UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data);
};
```

When the window manager communicates with a window, it accesses the **UIW_-WINDOW::Event( )** function first so that the window can override any default actions normally handled by the UI_WINDOW_OBJECT base class. If we create a window, we can pass event information to its function using **window->Event( )**. If the window cannot handle the event, it calls the **UI_WINDOW_OBJECT::Event( )** base class function.

The use of virtual functions in these cases lets us communicate with each object, without having to know the exact implementation details associated with the object. In larger hierarchies such as that used by Zinc Interface Library, this method becomes extremely useful.

Let's look at one more implementation of virtual member functions, using the pop-up item class as the example. When a message is sent to a pop-up item, it is first handled by the **UIW_POP_UP_ITEM::Event( )** member function. If the pop-up item does not wish to handle the message, it passes the information to the **UIW_BUTTON::Event( )** member function.

This switch of control is shown below:

```
EVENT_TYPE UIW_POP_UP_ITEM::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    EVENT_TYPE ccode = event.type;
    switch (ccode)
    {
    case S_INITIALIZE:
        UI_WINDOW_OBJECT::Event(event);
        break;

    default:
        ccode = UIW_BUTTON::Event(event);
        break;
    }

    // Return the control code.
    return (ccode);
}
```

The UIW_BUTTON class in turn calls **UI_WINDOW_OBJECT::Event( )** if it does not have any logical operation to handle the event.

```
EVENT_TYPE UIW_BUTTON::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    EVENT_TYPE ccode = LogicalEvent(event, ID_BUTTON);
    switch (ccode)
    {
        .
        .
        .

    default:
        ccode = UI_WINDOW_OBJECT::Event(event);
        break;
    }

    // Return the control code.
    return (ccode);
}
```

This process allows us to go up the class hierarchy, performing only those operations that are different from the base class definition.

Overloaded member functions

Overloaded member functions are used to access data in various forms. For example, the UI_DATE class overloads two member functions: **DataGet( )** and **DataSet( )**.

```
class EXPORT UI_DATE
{
public:
    void DataGet(int *year, int *month, int *day, int *dayOfWeek = NULL);
    void DataGet(char *string, int maxLength,
        USHORT dtFlags = DTF_NO_FLAGS);

    void DataSet(void);
    void DataSet(const UI_DATE &date);
    void DataSet(int year, int month, int day);
    void DataSet(const char *string, USHORT dtFlags = DTF_NO_FLAGS);
    .
    .
    .
};
```

The various overloaded export and import functions allow you to set or get a:

- system date (this method requires no arguments)
- date class object previously constructed
- date based on three integers: the year, month and day
- date based on an alphanumeric value

Each of these access functions is very useful at different points of an application.

Pointers to static member functions

Pointers to static member functions are used throughout the library. Zinc Interface Library supports the addition of user functions to objects. For example when a UIW_BUTTON object is created, it is desirable to have that button, when it is clicked, call some function that is external to the library. The following code shows how a static member function, userFunction, is called when the button is clicked:

```
EVENT_TYPE UIW_BUTTON::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    EVENT_TYPE ccode = LogicalEvent(event, ID_BUTTON);
    switch (ccode)
    {
        .
        .
        .
```

```
        case L_SELECT:
        case L_END_SELECT:
            UI_WINDOW_OBJECT::Event(event);
            if (userFunction &&
                (ccode == L_SELECT || true.Overlap(event.position)))
            {
                UI_EVENT tEvent = event;
                ccode = (*userFunction)(this, tEvent, ccode);
                break;
            }
            break;

        default:
            ccode = UI_WINDOW_OBJECT::Event(event);
            break;
        }
    }
```

Operator overloads

Operator overloads are used by Zinc Interface Library in two different fashions. The most common overload allows us to add a class object to an existing list. For example, the following code can be used to create a window and then attach sub-level window objects:

```
// Create a simple window and attach sub-level window objects.
UIW_WINDOW *window = new UIW_WINDOW(5, 5, 40, 6);
*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE("Simple Window");
```

The use of the + operator in this case allows us to add smaller elements (e.g., border, maximize button, minimize button, system button, title) to a parent control class (the window). This type of operator overload is permitted with the following major classes: UI_LIST, UI_EVENT_MANAGER, UI_WINDOW_MANAGER and UIW_WINDOW. (In addition, all objects derived from the UIW_WINDOW class inherit the overload capability).

The UI_DATE and UI_TIME classes also define operations for =, +, -, >, >=, <, <=, ++, --, +=, -=, == and !=.

```
class EXPORT UI_DATE : public UI_INTERNATIONAL
{
public:
    long operator=(long days) { value = days; return (value); }
    long operator=(const UI_DATE &date) { value = date.value;
        return (value); }
    long operator+(long days) { return (value + days); }
    long operator+(const UI_DATE &date) { return (value + date.value); }
    long operator-(long days) { return (value - days); }
    long operator-(const UI_DATE &date) { return (value - date.value); }
    int operator>(const UI_DATE &date) { return (value > date.value); }
    int operator>=(const UI_DATE &date) { return (value >= date.value); }
```

```
    int operator<(const UI_DATE &date) { return (value < date.value); }
    int operator<=(const UI_DATE &date) { return (value <= date.value); }
    long operator++(void) { value++; return (value); }
    long operator--(void) { value--; return (value); }
    void operator+=(long days) { value += days; }
    void operator-=(long days) { value -= days; }
    int operator==(const UI_DATE& date) { return (value == date.value); }
    int operator!=(const UI_DATE& date) { return (value != date.value); }
};

// ----- UI_TIME ------------------------------------------------
// ----- member functions found in TIME.CPP --------------------

class EXPORT UI_TIME : public UI_INTERNATIONAL
{
public:
    long operator=(long hundredths) { value = hundredths; return (value); }
    long operator=(const UI_TIME &time) { value = time.value;
        return (value); }
    long operator+(long hundredths) { return (value + hundredths); }
    long operator+(const UI_TIME &time) { return (value + time.value); }
    long operator-(long hundredths) { return (value - hundredths); }
    long operator-(const UI_TIME &time) { return (value - time.value); }
    int operator>(UI_TIME& time) { return (value > time.value); }
    int operator>=(UI_TIME& time) { return (value >= time.value); }
    int operator<(UI_TIME& time) { return (value < time.value); }
    int operator<=(UI_TIME& time) { return (value <= time.value); }
    long operator++(void) { value++; return (value); }
    long operator--(void) { value--; return (value); }
    void operator+=(long hundredths) { value += hundredths; }
    void operator-=(long hundredths) { value -= hundredths; }
    int operator==(UI_TIME& time) { return (value == time.value); }
    int operator!=(UI_TIME& time) { return (value != time.value); }
};
```

These operators are used to compare the chronological value of two date or time objects. The example below shows how the date operator overloads can be used to compare a date against special times throughout the year.

```
UI_DATE currentDate;     // Initialize the system date.
UI_DATE newYears1990("Jan. 1, 1990");
UI_DATE twentyFirstCentury("Jan. 1, 2001");

// Check the dates
if (currentDate == newYears1990)
    printf("Happy new year!\n");
else if (currentDate < twentyFirstCentury)
    printf("It's not the twenty-first century.\n");
else
    printf("It's the twenty-first century.\n");
```

Static member functions

Zinc Interface Library uses static member functions for the following reasons:

1—Static member functions are used to set general class information. For example, the UI_TIME class defines a static member that resets the string values associated with ante- and post-meridian times.

```
class EXPORT UI_TIME
{
public:
    static void AmPmSet(char *amPtr = NULL, char *pmPtr = NULL);
```

This allows the programmer to reset the time values without constructing a UI_TIME class.

2—Static member functions are used to check information before the associated class constructor is called. A good example of this use is with the UI_STORAGE class, where the programmer can check the validity of a file or directory path without first creating a storage unit. This is accomplished by calling the **UI_STORAGE::-ValidName( )** member function.

```
class EXPORT UI_STORAGE : public UI_LIST
{
public:
    static int ValidName(const char *name, int createStorage = FALSE);
```

3—Static members are used to perform generic operations. There are two static members that fit into this category: **UIW_WINDOW::Generic( )** and **UIW_-SYSTEM_BUTTON::Generic( )**. These member functions are used not only to construct the class object, but also to place generic sub-objects in their lists. For example, the definition for **UIW_WINDOW::Generic( )** allows you to make one call that initializes a window and adds the border, maximize, minimize and system buttons:

```
    UIW_WINDOW *UIW_WINDOW::Generic(int left, int top, int width, int
height,
        char *title, UIW_ICON *_icon, USHORT woFlags, USHORT woAdvanced-
Flags,
        int _helpContext)
    {
        UIW_WINDOW *window = new UIW_WINDOW(left, top, width, height,
            woFlags, woAdvancedFlags, _helpContext, _icon);
        // Add default window objects.
        *window
            + new UIW_BORDER
            + new UIW_MAXIMIZE_BUTTON
            + new UIW_MINIMIZE_BUTTON
            + UIW_SYSTEM_BUTTON::Generic()
            + new UIW_TITLE(title);

        // Return a pointer to the new window.
        return (window);
    }
```

4—Static member functions are used in conjunction with **UI_LIST::Get( )** to find specific elements in a list. The example below shows how the window manager defines and uses a static member function to find a window object that has the WOS_CURRENT flag set.

```
class EXPORT UI_WINDOW_OBJECT : public UI_ELEMENT
{
protected:
    static int FindStatus(void *object, void *matchStatus);
    .
    .
    .

int UI_WINDOW_OBJECT::FindStatus(void *object, void *matchStatus)
{
    return (FlagSet(((UI_WINDOW_OBJECT *)object)->woStatus,
        *(WOS_STATUS *)matchStatus) ? 0 : -1);
}

EVENT_TYPE UIW_WINDOW::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    EVENT_TYPE tcode;
    UI_WINDOW_OBJECT *object = NULL;
    EVENT_TYPE ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_WINDOW);
    switch (ccode)
    {
    case S_ERROR_RESPONSE:
        object = (UI_WINDOW_OBJECT *)
            UI_LIST::Get(UIW_WINDOW::FindStatus, &WOS_CURRENT);
        .
        .
        .
```

5—Static members are used to send system messages to the event manager. For example, when the end user clicks the mouse button on the maximize button, the maximize button sends a message through the system telling the parent window to maximize itself. The code below shows how this is accomplished.

```
class EXPORT UIW_MAXIMIZE_BUTTON : public UIW_BUTTON
{
protected:
    static void MaximizeUserFunction(UI_WINDOW_OBJECT *button,
        UI_EVENT &event, EVENT_TYPE ccode)
        { event.type = S_MAXIMIZE;
            button->eventManager->Put(event, Q_BEGIN); }
```

5—Finally, static members are used to access member constructors. This advanced operation is only used when persistent objects are created within an application. For example, the **UIW_BUTTON::New( )** static member is defined as follows:

```
    UIW_BUTTON(const char *name, UI_STORAGE *file = NULL,
    USHORT loadFlags = L_NO_FLAGS);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        USHORT loadFlags)
        { return ((name || file) ?
            new UIW_BUTTON(name, file, loadFlags) :
            new UIW_BUTTON(0, 0, 10, "button")); }
```

This static member should not be directly accessed by programmers.

The use of static members is beneficial, if used properly. Many times however, there is a tendency to over-use static members to allow structured programming. You should carefully evaluate your use of static members in your application.

Conclusion

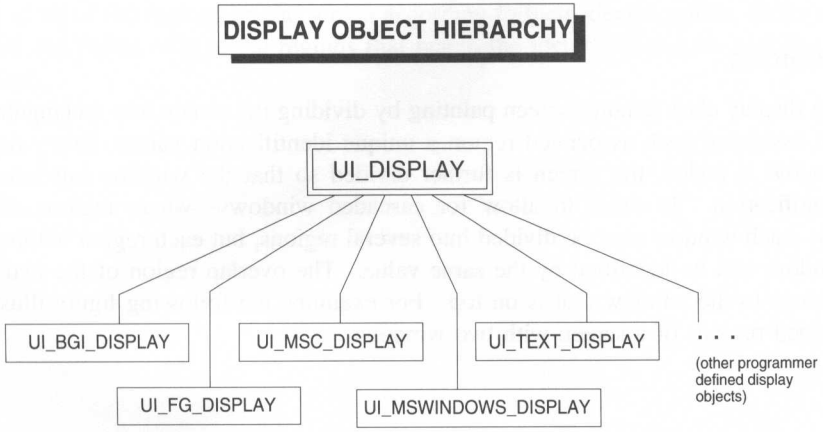
This concludes the discussion of Zinc Interface Library and its implementation of C++ features. See the *Programmer's Tutorial* for tutorials that are designed to help you with specific implementation or design skills that will help you write more effective applications using Zinc Interface Library.

CHAPTER 20 - SCREEN DISPLAY

Introduction

The screen display is a support resource to Zinc Interface Library. It provides low-level screen support in both graphics and text modes.

The base display class is UI_DISPLAY, which is an abstract class that defines the general features needed for screen output. It is designed to be abstract so that Windows, DOS Graphics, and DOS Text modes can be supported without limiting the unique functionality of any one mode. From UI_DISPLAY five more specific classes are derived: UI_BGI_-DISPLAY, which controls Borland graphics mode; UI_FG_DISPLAY, which controls Zortech graphics mode; UI_MSC_DISPLAY, which controls the Microsoft graphics mode; UI_TEXT_DISPLAY, which controls text mode; and UI_MSWINDOWS_DISPLAY, which operates under Microsoft Windows. This hierarchy is represented in the figure below:



Coordinate system

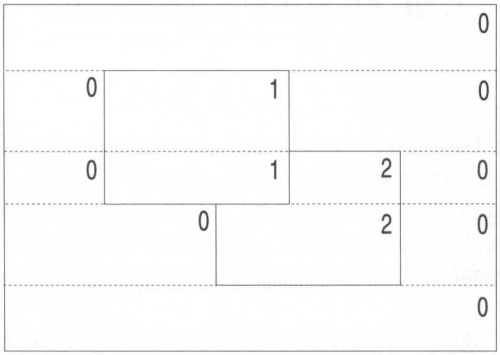
The upper left corner of the screen display is position (0,0). The x-coordinates increase from the left of the screen to the right, while the y-coordinates increase from the top of

the screen towards the bottom. The following list shows the coordinates for the bottom right corner of the modes supported by Zinc Interface Library:

- 25 line x 80 column text mode = 24,79
- 25 line x 40 column text mode = 24,39
- 43 line x 80 column text mode = 42,79
- 50 line x 80 column text mode = 49,79
- 200 line x 320 column CGA mode = 199,319
- 200 line x 640 column CGA mode = 199,639
- 200 line x 640 column EGA mode = 199,639
- 350 line x 640 column EGA mode = 349,639
- 348 line x 720 column Hercules mode = 347,719
- 200 line x 320 column Hercules mode = 199,319
- 200 line x 640 column Hercules mode = 199,639
- 400 line x 640 column Hercules mode = 399,639
- 200 line x 640 column VGA mode = 199,639
- 350 line x 640 column VGA mode = 349,639
- 480 line x 640 column VGA mode = 479,639

Clip regions

The display class handles screen painting by dividing the screen into rectangular regions and assigning each associated region a unique identification value. Every time a new window is added, the screen is further divided so that the window can have its own identification. In order to allow for cascaded windows—where regions often overlap—each window may be divided into several regions, but each region within the same window will be identified by the same value. The overlap region of the two windows belongs to the window that is on top. For example, the following figure illustrates the divided regions of a screen with two windows:



The background display always has an identification value of 0, and each window attached to it has its own identification.

When a message is received that requires updating the screen, all input devices are first automatically turned off so that they do not interfere. Next, the display, which maintains a list of all of the regions on the screen according to their identifications, walks through the list and paints only to the regions that match the identification sent with the update message.

CHAPTER 21 – DEFAULT EVENT MAPPING

Overview

"Chapter 2—Conceptual Design" of this manual briefly discussed the implementation of event mapping in Zinc Interface Library. This chapter describes the default mapping of events for the UID_KEYBOARD and UID_MOUSE devices. This default event mapping conforms to the key assignments specified by IBM's Systems Application Architecture document—the Common User Access Panel Design and User Interaction edition.

Zinc Interface Library maintains internal communication through a continuous flow of event messages. These messages are sent through the event queue portion of the event manager, which determines the proper destination for the event.

Event map table

Interpretation of event messages is determined by event map tables. These tables contain a listing of all events that can be sent by the various devices and the logical interpretations of those events. For example, the following portions of **_eventMapTable** define the interpretations associated with the selection process on a window object for the keyboard and mouse devices:

```
static UI_EVENT_MAP __eventMapTable[] =
{
    { ID_WINDOW_OBJECT,     L_NEXT,          WM_CHAR,        TAB },
    { ID_WINDOW_OBJECT,     L_PREVIOUS,      WM_CHAR,        BACKTAB },
    { ID_WINDOW_OBJECT,     L_SELECT,        WM_CHAR,        ENTER },
    .
    .
    .
    { ID_WINDOW_OBJECT,     L_CONTINUE_SELECT,  E_MOUSE,        M_LEFT },
    .
    .
    .
    // End of array.
    { ID_END, 0, 0, 0 }
};
```

An event map table entry is composed of the identification for the type of object, the logical event, the device type that produced the message, and the raw scan code of the event. The first entry above, for example, indicates that a window object will process an L_NEXT message when a user presses the <Tab> key.

Not only does Zinc Interface Library's event mapping allow for different devices to generate the same logical message, but it also allows the same event to be interpreted differently by various objects. For example, the following table defines event inter-

pretations for a string object interprets a click on the left mouse button as part of a mark operation instead of as a select operation:

```
{ ID_STRING,        L_BEGIN_MARK,    E_MOUSE,    M_LEFT | M_LEFT_CHANGE},
{ ID_STRING,        L_CONTINUE_MARK, E_MOUSE,    M_LEFT},
{ ID_STRING,        L_END_MARK,      E_MOUSE,    M_LEFT_CHANGE},
```

Algorithm

When an event message is received by the event manager, the default mapping algorithm walks through the event map table and searches for the best match according to the object's identification, the device's identification, and the raw scan code associated with the event. For example, if the left mouse button has been pressed while the user is positioned in a string object, the map table will be scanned until the best possible match is found, which is shown below:

```
{ ID_STRING,        L_BEGIN_MARK,    E_MOUSE,    M_LEFT | M_LEFT_CHANGE}
```

As a result, the mark operation will begin within the string object. When the L_END_-MARK logical message is interpreted, the mark operation will be completed.

Default keyboard mapping

Action	Key	Description
Begin field	**<Ctrl Home>** **<Ctrl Gray Home>**	Moves to the beginning of the field.
Copy	**<Ctrl F7>**	Copies the entire contents of the current window field. The copied section is stored in a global paste buffer. This key only has effect in fields that can be edited.
Cut	**<Ctrl F6>**	Cuts the entire contents of the current window field. The cut section is removed and stored in a global paste buffer. This key only has effect in fields that can be edited.
Delete next character	**<Del>** **<Gray Delete>**	Deletes the character underneath the cursor, leaving the position of the cursor unchanged. This key only has effect in

		fields that can be edited and only where the cursor is not in the field's *last* position.
Delete previous character	**<Backspace>**	Moves the cursor *left* one position, deleting the character underneath the cursor (i.e. the character immediately to the left of the cursor before it is moved). This key only has effect in fields that can be edited and only where the cursor is not in the field's first character position.
Delete temporary window	**<Esc>**	If the current window is identified as a temporary window (WOAF_TEMPORARY), pressing <Esc> removes the current window from the screen display. For example, when an end user selects the system button, a pop-up menu appears. If the user presses <Esc> at this time, the pop-up menu is erased from the screen display.
Delete window	**<Shift F4>**	Closes a window that is not temporary.
Delete word	**<Ctrl Del>** **<Ctrl Gray Delete>**	Positions the cursor at the beginning of the word to be deleted, then deletes the word and any trailing spaces. The cursor remains in its original position after the deletion.
Down	**<↓>** **<Gray ↓>**	If the field occupies a single line on the screen or the cursor is positioned on the bottom line of a multi-line field, pressing <Down-arrow> moves from the current (or selected) window field to the window field immediately *below* the current field. The left or right edge of the field above must be on the same boundary as the current field (i.e., their left edges or right edges must have the

		same pixel or cell coordinate). If the field is a multi-line field and the cursor is not positioned on the bottom line, pressing <Down-arrow> moves the cursor *down* one line on the display.
Down page	**<PgDn>** **<Gray PgDn>**	If the field occupies a single line on the screen or the cursor is positioned on the bottom line of a multi-line field, pressing <PgDn> moves from the current (or selected) window field to the *last* window field. If the field is a multi-line field and the cursor is not positioned on the bottom line, pressing <PgDn> moves the cursor *down* one page in the current field.
End field	**<Ctrl End>** **<Ctrl Gray End>**	Moves to the end of the field.
End line	**<End>** **<Gray End>**	Moves the cursor to the end of the current line.
Exit	**<Alt F4>** **<Shift F3>** **<Ctrl Break>** **<Ctrl C>**	Exits the application program.
Help—context sensitive	**<F1>**	Displays context sensitive help information regarding the current window.
Help—general	**<Alt F1>**	Displays general help information for the application program.
Home	**<Home>** **<Gray Home>**	Moves the cursor to the beginning of the current line.
Left	**<←>** **<Gray ←>**	If the cursor is positioned in the *first* character position of a right-hand field, pressing <Left-Arrow> moves the cursor to the *last* character position of a left-

		hand field. Otherwise, pressing <Left-Arrow> moves the cursor one character to the left.
Left word	**<Ctrl ←>** **<Ctrl Gray ←>** **<Alt Gray ←>**	Moves the cursor to the beginning of the previous word or to the beginning of the same word if the cursor was originally positioned in the middle of that word. (word left)
Mark	**<Ctrl F5>**	Begins a marked region on the position of the cursor (only in fields that can be edited). When followed by any movement keys and then <Enter>, the marked text is copied. When followed by any movement keys and then <Del>, the marked text is cut. The cut section is removed and stored in a global paste buffer.
Maximize	**<Alt +>** **<Alt F10>**	Maximizes the size of the current window (i.e., increases the size of the window to occupy the whole screen). This key only has effect when the current window can be sized and if it is not already in a minimized state. If the window is in a maximized state, selecting this key causes the window to be restored to its original size.
Menu control	**<Alt>** **<F10>**	Selects the pull-down menu (if any) associated with the current window. This changes the highlight field, or cursor position, from the current field to the pull-down menu. This key only has effect when the current window has a pull-down menu.
Minimize	**<Alt –>** **<Alt F9>**	Minimizes the size of the current window (i.e., reduces the size of the window to the minimum allowed by the object type). This key only has effect when the current window can be sized

		and if it is not already in a maximized state. If the window is in a minimized state, selecting this key causes the window to be restored to its original size.
Move window	**<Alt F7>**	Moves the current window when followed by any movement key and then <Enter>. When followed by any movement key and then <Esc>, the selected window is returned to its original position.
Next field	**<Enter>** **<Gray Enter>** **<Tab>** **<F6>**	Moves from the current (or selected) window field to the *next* selectable window field. If the last window field is currently selected, pressing <Tab> cycles to the *first* selectable window field.
Next window	**<Alt F6>**	Moves from the current (or selected) window to the *next* selectable window in the window manager's list of windows.
Paste	**<Ctrl F8>**	Retrieves the cut section from the global paste buffer and pastes it in the current field. This key only has effect in fields that can be edited.
Previous field	**<Shift F6>** **<Shift Tab>**	Moves from the current (or selected) window field to the *previous* selectable window field. If the first window field is currently selected, pressing <Back-Tab> cycles to the *last* selectable window field.
Refresh	**<F5>**	Refreshes the screen. (Re-displays all of the window objects on the screen.)
Restore	**<Alt F5>**	Restores the original size of the window. Used with <Alt +> and <Alt ->.
Right	**<→>**	If the cursor is positioned in the *last*

	<Gray →>	character position of a left-hand field, pressing <Right-Arrow> moves the cursor to the *first* character position of a right-hand field. Otherwise, pressing <Right-Arrow> moves the cursor one character to the right.
Right word	**<Ctrl →>** **<Ctrl Gray →>** **<Alt →>** **<Alt Gray →>**	Moves the cursor to the beginning of the next word. (word right)
Size window	**<Alt F8>**	Sizes, from the bottom right corner, the current window when followed by any movement key. Pressing <Enter> accepts the alteration in size, while pressing <Esc> returns the window to its original size.
System	**<Alt Spacebar>** **<Alt .>**	Selects the system button (if any) associated with the current window. This causes the pop-up menu associated with the current window's system button to be displayed on the screen.
Toggle	**<Ins>** **<Gray Insert>**	Toggles the edit mode from *insert* to *overstrike* mode or vise-versa. This key only has effect in fields that can be edited.
Up	**<↑>** **<Gray ↑>**	If the field occupies a single line on the screen or the cursor is positioned on the top line of a multi-line field, pressing <Up-arrow> moves from the current (or selected) window field to the window field immediately *above* the current field. The left or right edge of the field above must be on the same boundary as the current field (i.e., their left edges or right edges must be on the same pixel or cell coordinate). If the field is a multi-

		line field and the cursor is not positioned on the top line, pressing <Up-arrow> moves the cursor *up* one line on the display.
Up page	**<PgUp>** **<Gray PageUp>**	If the field occupies a single line on the screen or the cursor is positioned on the top line of a multi-line field, pressing <PgUp> moves from the current (or selected) window field to the *first* window field. If the field is a multi-line field and the cursor is not positioned on the top line, pressing <PgUp> moves the cursor *up* one page in the current field.

Default mouse mapping

Action	Mouse	Description
Choose	**<Left-down-click>**	If the end user is on the window's title bar, pressing this button moves the window. If the end user is on the window's border, pressing this button sizes the window. Otherwise, pressing the left mouse button selects the field positioned under the mouse cursor (if the field is selectable).
Mark	**<Left-drag>**	If the current field is an field that can be edited, holding the left button down and dragging the mouse specifies the mark location.
Select	**<Left-release>**	If the current field is a field that can be edited, releasing this button completes the mark specification. Otherwise, releasing this button completes the select operation.

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z