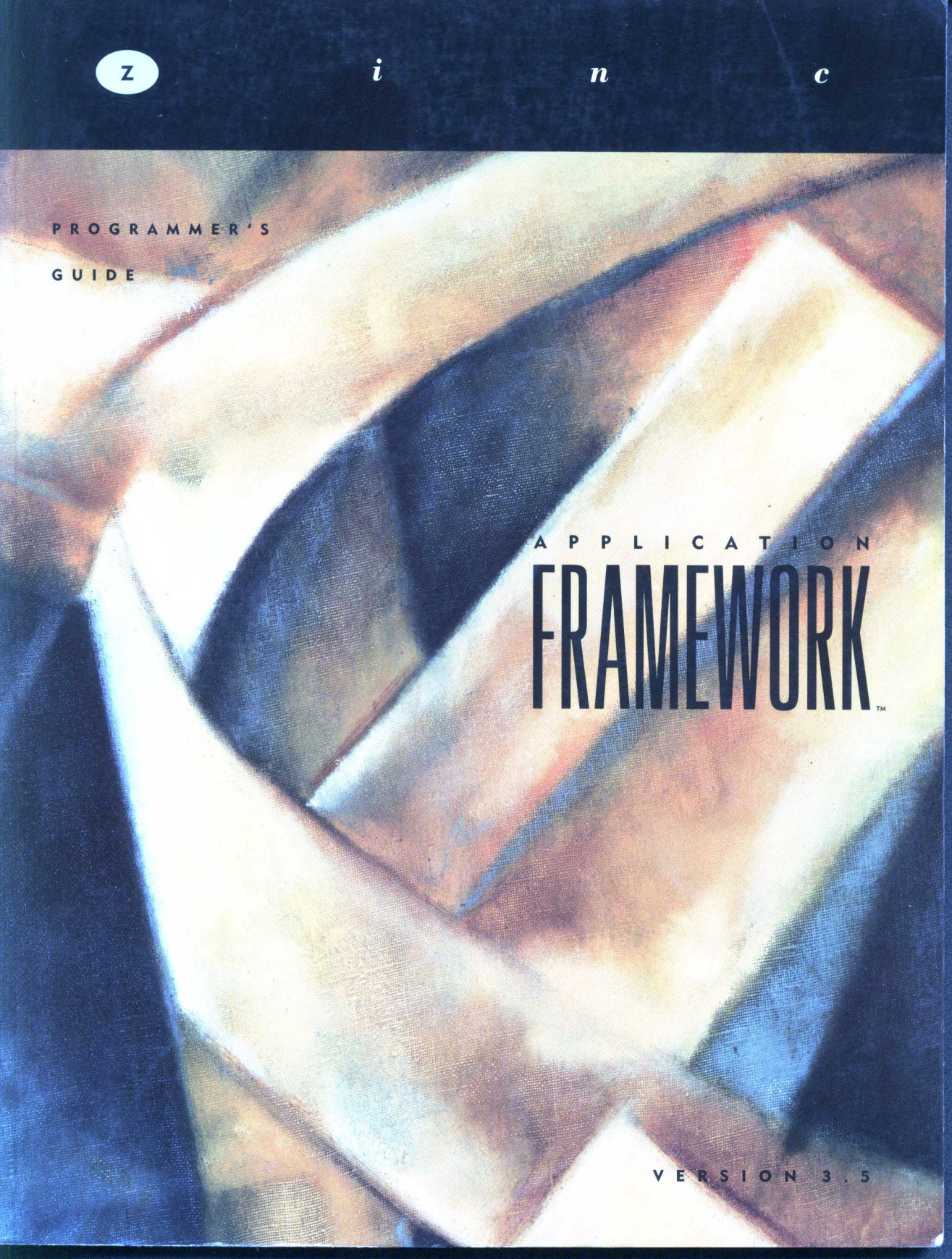

z i n c
PROGRAMMER'S
GUIDE
APPLICATION
FRAMEWORK™
VERSION 3.5

A new paradigm. . .

Zinc® Application Framework™

Programmer's Guide

Version 3.5

Zinc Software Incorporated
Pleasant Grove, Utah

TABLE OF CONTENTS

SECTION I
GETTING STARTED

CHAPTER 1 – INTRODUCTION

Overview

Zinc Application Framework 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 Text and DOS Graphics in real mode you need Zinc Application Framework 3.5 (Zinc Engine and DOS Key), 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), and a Microsoft compatible mouse driver. To develop applications for DOS Text and DOS Graphics in protected mode you also need a DOS extender SDK (see the **READ.ME** file for a list of currently supported DOS extenders).

To develop applications for Microsoft Windows 3.X you need Zinc Application Framework 3.5 (Zinc Engine and Microsoft Windows Key), a C++ compiler for Microsoft Windows (i.e., Borland C++, Microsoft C++ or Zortech C++) and Microsoft Windows 3.0 or later. To develop applications for Microsoft Windows NT you need Zinc Application Framework 3.5 (Zinc Engine and Windows NT Key) a C++ compiler for Windows NT (i.e., Borland C++, Microsoft C++ or Zortech C++) and the Microsoft Windows NT SDK.

To develop applications for IBM OS/2 2.X you need Zinc Application Framework 3.5 (Zinc Engine and OS/2 Key), a C++ compiler for OS/2 (i.e., Borland C++ or Zortech C++) and the IBM OS/2 2.0 or later SDK.

To develop applications for OSF/Motif you need Zinc Application Framework 3.5 (Zinc Engine and Motif Key), a C++ compiler compatible with version 2.1 of the AT&T C++ translator, and OSF/MOTIF 1.1 running on X11 R4. (**NOTE**: Some source code changes may be required to use the Motif Key on hardware platforms that are not directly supported by Zinc. See the **README** file for a list of currently supported hardware platforms.)

To develop applications for PenDOS 1.1 you need Zinc Application Framework 3.5 (Zinc Engine, DOS Key and PenDOS Key), a C++ compiler for DOS (i.e., Borland C++, Microsoft C++ or Zortech C++), DOS 5.0, CIC PenDOS 1.1 SDK and a PenDOS compatible digitizer or pen computer.

Using The Manuals

Programmer's Guide

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

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

Section II—Basic Concepts gives a high-level description of Zinc Application Framework, including the conceptual operation of the library and its major pieces. Introductions on how to use Zinc to build applications for DOS (real and protected modes), Microsoft Windows 3.X, Windows NT, IBM OS/2 2.X and OSF/Motif as well as an overview of Zinc Designer, are also given.

Section III—Advanced Concepts gives an in-depth description of Zinc Application Framework 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 Application Framework 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 Application Framework.

Class object references—This section (Chapters 1 through 71) contains short descriptions about the class objects (or structures), the public and protected member variables and functions (private members are not documented) and the calling conventions used with the class object.

Miscellaneous information—This section (Appendices A through F) contains support definitions, system event definitions, logical event definitions, class identifications, storage information and environment specific keyboard information.

Programming Techniques

The *Programming Techniques* contains a series of tutorials designed to help the programmer in learning the features and practical uses of Zinc Application Framework.

Section I—Hello World! describes how to initialize the main components of Zinc Application Framework.

Section II—Dictionary describes the transition from C programming to C++ programming, adding Zinc Application Framework to an existing application and using the Zinc storage file.

Section III—Zinc Application Program describes the overall design and implementation issues that should be considered when creating applications using Zinc Application Framework.

Section IV—Derived Classes contains a set of tutorials that show how Zinc Application Framework 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 V—Portability Issues contains a set of tutorials that present methods for obtaining program portability. International currency and multi-language programs are discussed.

Section VI—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 VII—Zinc Designer contains an in-depth description of Zinc Designer and all of its components. The available objects, their edit windows and general interaction are described.

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

Typefaces

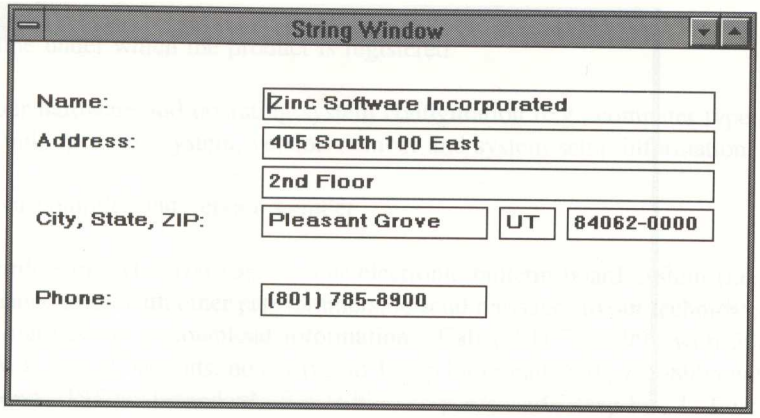
Special typefaces are used throughout the documentation in order to clarify the usage and meaning of specific terms. Familiarity with the following typeface conventions will be helpful in working with these manuals:

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.)

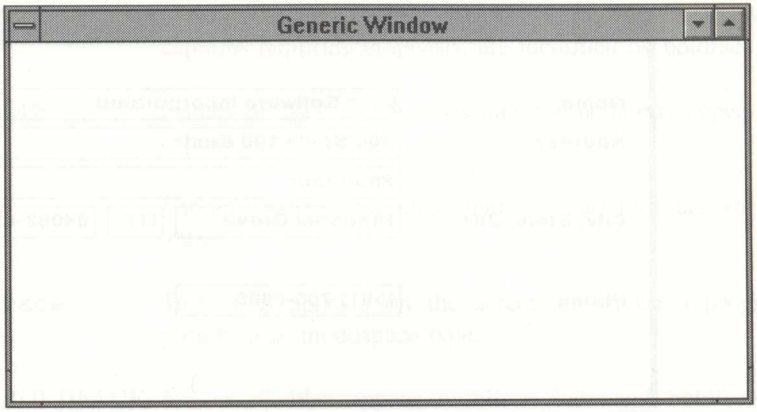


UI_—The prefix identification for all class objects used in Zinc Application Framework. The "UI" stands for "User Interface." This prefix allows programmers to distinguish the user interface part of their application.

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 Application Framework End User Software License Agreement.

Telephone support—If you need assistance beyond what the Zinc Application Framework 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 Saturdays, Sundays and holidays. Please have the following information ready before you call:

- Your Zinc Application Framework version and serial number, as well as the name under which the product is registered

- Your hardware and operating system configuration (e.g., computer type, mouse brand, operating system, version number and system setup information)

- Your compiler and version number

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 300-9600 baud (HST dual standard), 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.

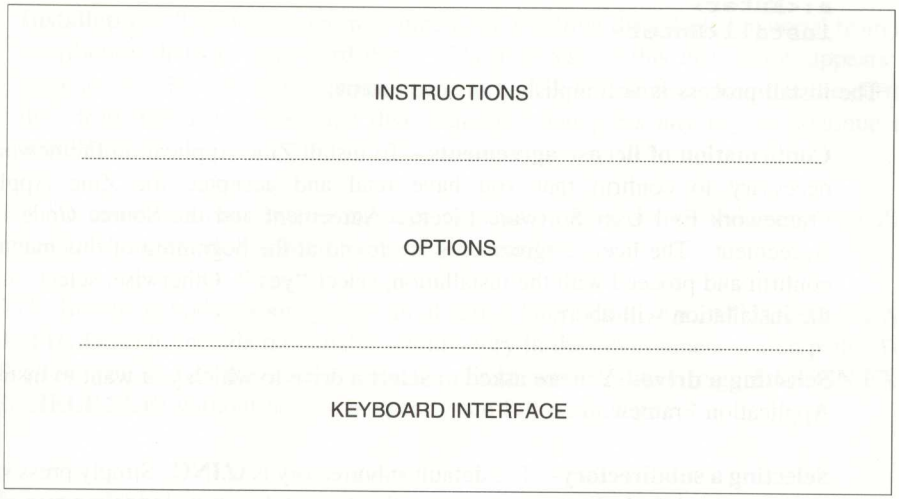
NOTE: IF YOU NEED TO SEND MORE THAN ONE PAGE OF CODE, DO NOT USE THE FAX—PLEASE USE THE ZINC BBS.

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

CHAPTER 2 – INSTALLATION

The generic Zinc Application Framework install program is a DOS application and therefore must be run from DOS or a DOS session. Please see the **READ.ME** file for environment-specific installation information (e.g., Motif).

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 Application Framework, 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 Application Framework requires DOS 2.1 or later, 640K RAM and a hard disk drive. The installation program copies files to a hard disk or network drive.

Insert "Zinc Engine Disk 1" into the desired drive, make that drive current and invoke the installation program. For example, to install Zinc Application Framework from drive A, insert the first disk and type:

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

The install process is accomplished in seven steps:

Confirmation of license agreements—To install Zinc Application Framework, it is necessary to confirm that you have read and accepted the Zinc Application Framework End User Software License Agreement and the Source Code License Agreement. 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 Zinc Application Framework.

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

Selecting the package option—You are asked to identify which Zinc Application Framework package(s) you with to install. The options are:

- Zinc Engine (required)
- DOS Key
- Windows Key
- Windows NT Key
- OS/2 Key
- PenDOS Key (requires DOS Key)

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

- Borland
- Microsoft
- Zortech

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 Application Framework has been successfully installed.

NOTE: Be sure to update your system initialization information (e.g., **AUTOEXEC.BAT**, **CONFIG.OS2**) to include the **...\BIN** subdirectory in the environment search path. This will allow you to run the Zinc Application Framework utilities (e.g., **DESIGN.EXE**, **GENHELP.EXE**) without having to specify the full path name.

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).
- **WIN_ZIL.DLL** (the DLL version of Zinc Application Framework for use with Microsoft Windows 3.X).
- **WNT_ZIL.DLL** (the DLL version of Zinc Application Framework for use with Microsoft Windows NT).
- **OS2_ZIL.DLL** (the DLL version of Zinc Application Framework for use with IBM OS/2).

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). (**NOTE: .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 Application Framework is a new generation object-oriented development tool. Zinc helps you easily solve problems related to the software dilemma. With Zinc's single-source support for DOS Text and DOS Graphics (in real and protected modes), Microsoft Windows 3.X, Windows NT, IBM OS/2, OSF/Motif and CIC PenDOS 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 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. For example, you also benefit from Zinc Designer's 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 Zinc Designer on any of the other supported platforms.

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, you will see many compelling benefits of object-orientation.

The transition to object-oriented programming is not a trivial endeavor. But Zinc Application Framework 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 programming techniques 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 Application Framework, 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, Microsoft Windows 3.X, Windows NT, IBM OS/2, OSF/Motif or PenDOS.

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 Application Framework is also conceptually intuitive and easy to understand.

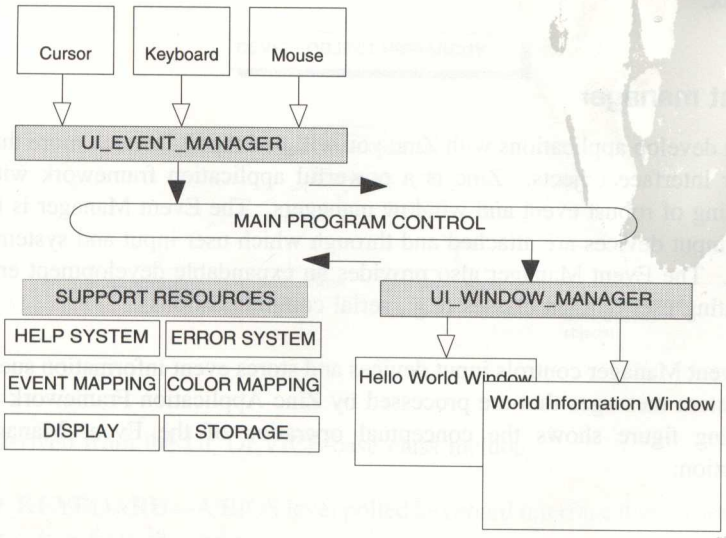
Reusability—Not only are Zinc's base classes reusable, but any object or class that you create can become a part of your tool kit. You save time by using classes that have previously been tested and debugged.

Extensibility—Because Zinc Application Framework 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 Application Framework

Zinc Application Framework'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, Microsoft Windows 3.X, Windows NT, IBM OS/2, OSF/MOTIF and CIC PenDOS 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.

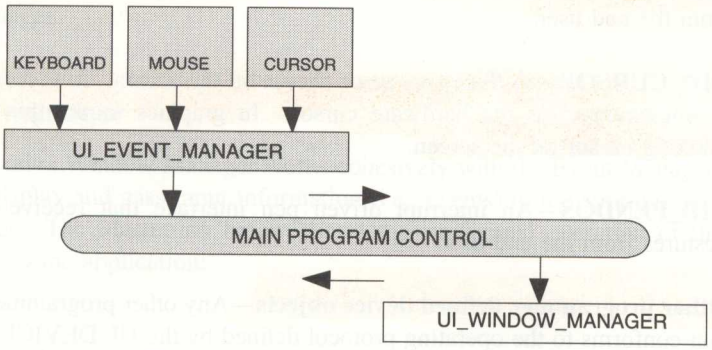
Event mapping—Controls the mapping of raw input (e.g., Microsoft Windows messages, OS/2 messages, DOS keyboard scan codes) to logical system events (e.g., sizing, moving, redrawing).

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

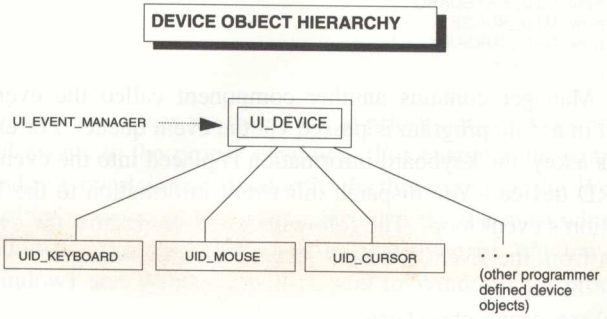
The event manager

As you develop applications with Zinc you will find that it is much more than a collection of user interface objects. Zinc is a powerful application framework with a backbone consisting of robust 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 an expandable development environment for integrating user-defined classes (e.g., serial communications).

The Event Manager controls input devices and stores event information such as user input and system messages that are processed by Zinc Application Framework 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 Application Framework, 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 abstract base class UI_DEVICE serves as a template for all Zinc devices. UI_DEVICE has the following 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.

UID_PENDOS—An interrupt driven pen interface that receives pen input and gestures from the end user.

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).

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;
```

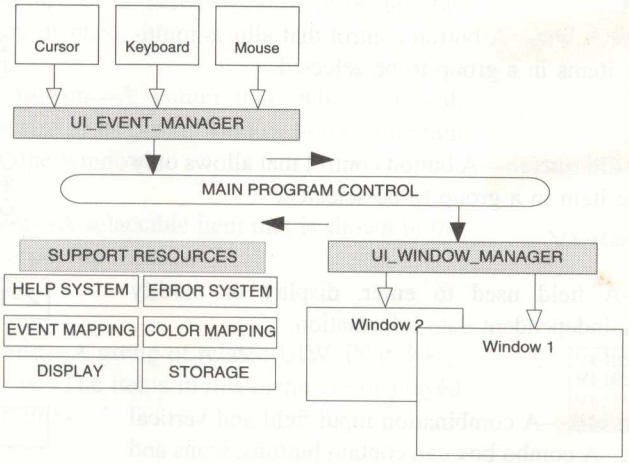
The Event Manager contains another component called the event queue. All event information in a Zinc program is passed via the event queue. For example, when the end user presses a key, the keyboard information is placed into the event queue by the UID_-KEYBOARD device. You dispatch this event information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

```
// declare event structure.
UI_EVENT event;
EVENT_TYPE ccode;
do
{
    // 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 && ccode != S_NO_OBJECT);
```

Other elements of Zinc Application Framework 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 (i.e., events) 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 and is used to channel events to the proper windows. 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 minimized windows (i.e., icons). Additional window objects may be attached to windows. All of these objects are 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

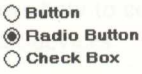
Border—An outlining border drawn around a window.

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:

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

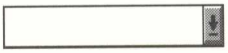
Check box—A button control that allows multiple items in a group to be selected.

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

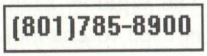


Date—A field used to enter, display or modify country-independent date information.

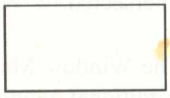
Combo box—A combination input field and vertical list box. A combo box can contain buttons, icons and string fields.



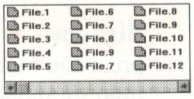
Formatted string—An input field used to enter, display or modify ASCII string buffers that contain literal characters or characters that cannot be edited (e.g., phone numbers, social security numbers).



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

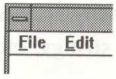
Pop-up menu—A group of related UIW_POP_UP_-ITEM objects. The items in this menu are displayed on multiple lines.

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

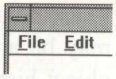
Prompt—A string that is used to describe the contents of another window field.

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.

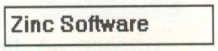


Real—A field used to enter, display or modify floating point numeric information. Real numbers are used for computation values and fractional numbers.

0.00000000

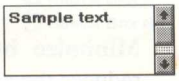
Scroll bar—A selectable region used to scroll the displayed portion of a window, list box or text input field.

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

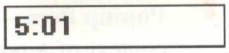


System button—A button that, when selected, shows general operations that can be performed on the parent window.

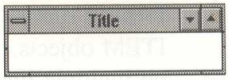
Text—A field used to enter, display or modify a multi-line text buffer.



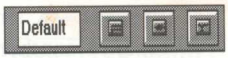
Time—A field used to enter, display or modify country-independent time information.



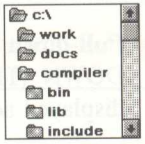
Title—An object that occupies the top region of a window and contains a window's title information.



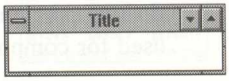
Tool bar—A group of related window objects. The tool bar is similar to a pull-down menu with the exception that it may contain different types of objects, such as: bitmapped buttons, dates, icons, strings, etc.



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.

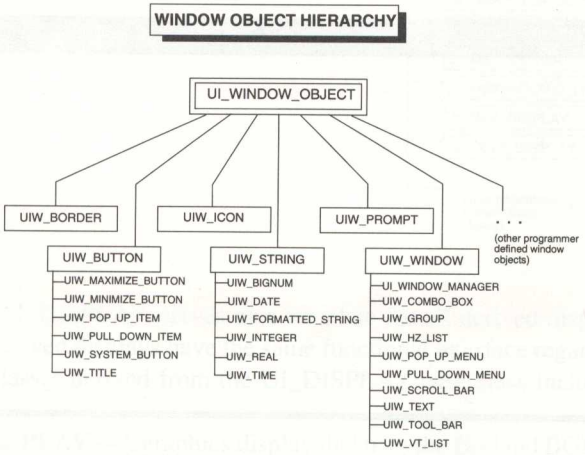


Window—A rectangular region of the screen that is composed of one or more class objects derived from the UI_WINDOW_OBJECT base class. A window may also contain sub-windows (e.g., MDI windows.)



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.

Window objects are derived from UI_WINDOW_OBJECT and have the following hierarchy:

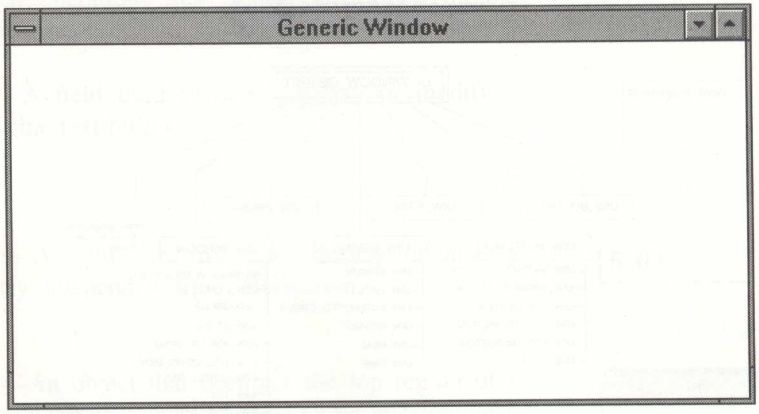


You attach windows to the Window Manager at run-time. Once a window is attached, it receives event information from the Window Manager. The following code shows how to construct a new window manager class object and how to initialize a selected window:

```
// Construct the screen display using the Zinc constructor.
UI_DISPLAY *display = new UI_GRAPHICS_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.
UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_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("Generic Window", WOF_NO_FLAGS);
*windowManager + window;
```

Windows and window objects have distinct representations in DOS Text, DOS Graphics, Microsoft Windows 3.X, Windows NT, IBM OS/2 and OSF/Motif. 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.

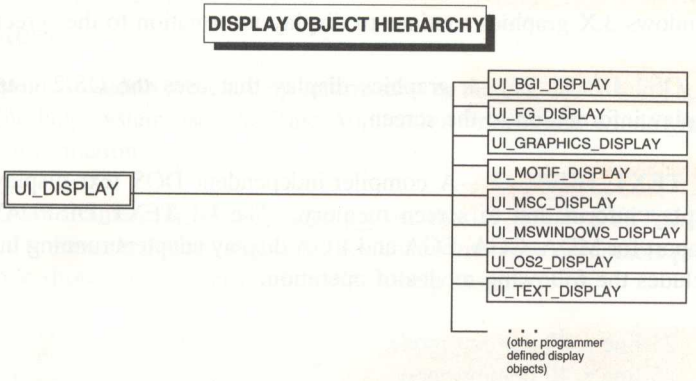
Cut—Cuts the marked area of the current field and stores the marked contents in a global paste buffer. This data can later be pasted into any other field, as long as the information is valid for that field type (e.g., the text "400" could be pasted into a numeric, string or text field).

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 Application Framework:



The base class, UI_DISPLAY, serves as a template for all derived display classes. This ensures that all derived displays have the same functional interface regardless of the actual display used. 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_GRAPHICS_DISPLAY—A compiler-independent DOS graphics display that uses the GFX graphics libraries by C-Source (included with Zinc Application Framework) to display information to the screen. The UI_GRAPHICS_DISPLAY class provides support for CGA, EGA, VGA, SVGA and Hercules monochrome display adapters running in graphics mode.

UI_MOTIF_DISPLAY—A graphics display that uses the Motif 1.1 Widget toolkit to display information on an X Windows display.

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, SVGA 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_OS2_DISPLAY—A graphics display that uses the OS/2 graphics routines to display information to the screen.

UI_TEXT_DISPLAY—A compiler-independent DOS 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. Third party display classes supporting Metawindows by Metagraphics and GX Graphics by Genus Microprogramming 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, Microsoft Windows 3.X, Windows NT, IBM OS/2, OSF/Motif and CIC PenDOS 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_GRAPHICS_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 Application Framework windowing system to present help information.

Zinc Application Framework initially does not use the UI_HELP_SYSTEM so that you are not forced to have the help system modules linked into your executable program. 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 Application Framework windowing system to present error information.

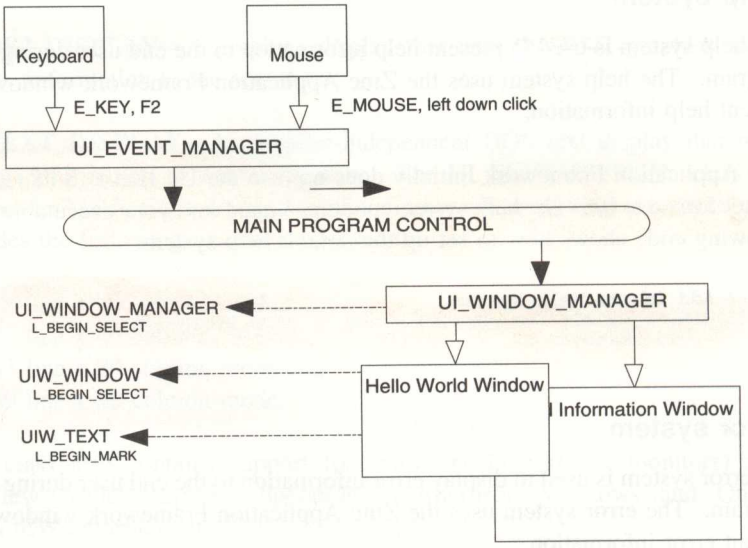
Zinc Application Framework initially does not use the UI_ERROR_SYSTEM so that you are not forced to have the error system modules linked into your executable program. The following code shows how to set-up the default error system:

```
// 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 Application Framework 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 redefine 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 Application Framework.

Storage and retrieval

Zinc Application Framework 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 Application Framework 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

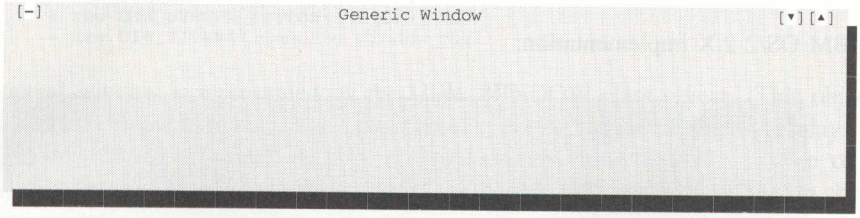
Introduction

"Chapter 3—Conceptual Design" of this manual briefly describes the types of window objects that are available with Zinc Application Framework. 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.

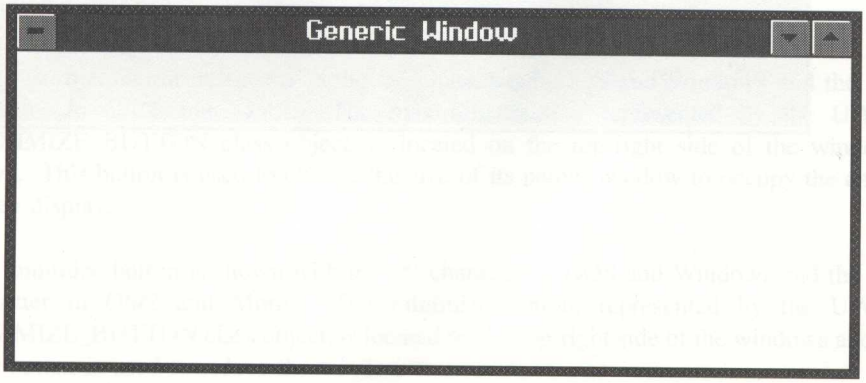
Basic window objects

Most windows created for an application will contain a border, title, maximize button, minimize button and system button. The figures below show various implementations of a window with these basic window objects and the code implementation.

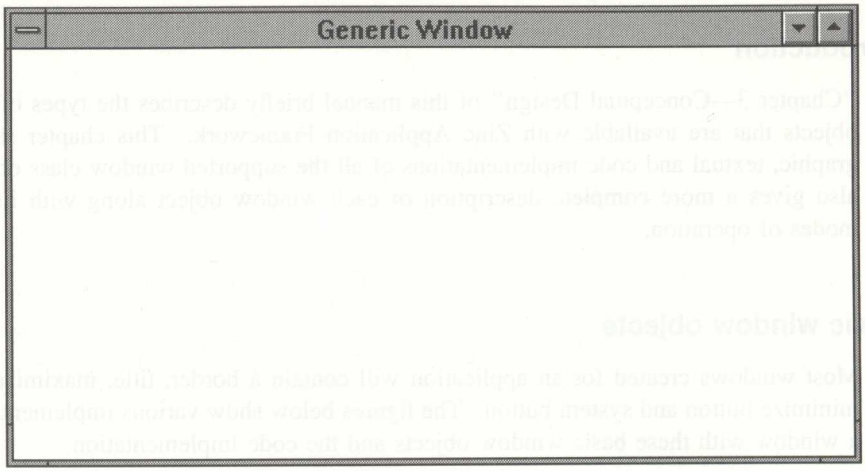
DOS Text implementation:



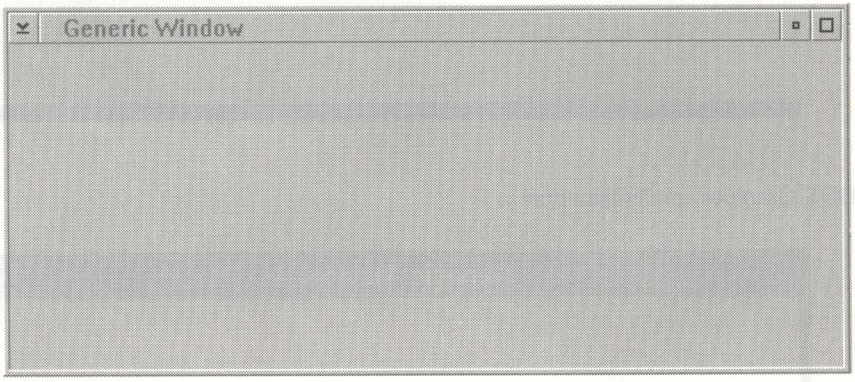
DOS Graphics implementation:



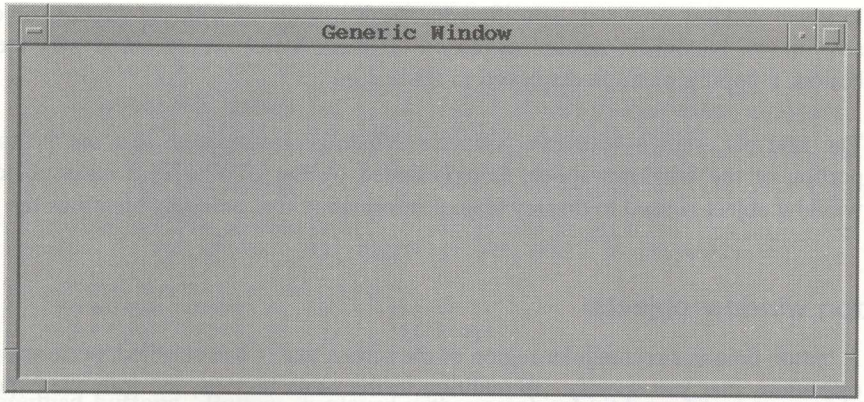
Microsoft Windows 3.X and Windows NT implementation:



IBM OS/2 2.X implementation:



OSF/Motif implementation:




```
*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON(SYF_GENERIC)
    + new UIW_TITLE(" Generic Window ");
```

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 border is displayed as a shadow.

The maximize button is shown as the '▲' character in DOS and Windows and the '□' character in OS/2 and Motif. The maximize button, represented by the UIW_-MAXIMIZE_BUTTON class object, is located on the top-right side of the windows above. This button is used to change the size of its parent window to occupy the entire screen display.

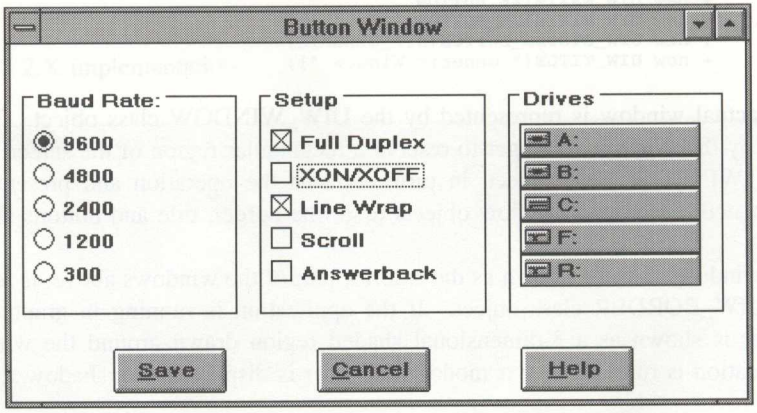
The minimize button is shown with the '▼' character in DOS and Windows and the '□' character in OS/2 and Motif. The minimize button, represented by the UIW_-MINIMIZE_BUTTON class object, is located on the top-right side of the windows above. This button is used to reduce the window to an icon.

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.

The title bar, shown with the "Generic Window" 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.

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) and the code implementation:




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

    // Add the radio buttons.
    + &(*new UIW_GROUP(1, 2, 13, 7, "Baud Rate:")
        + new UIW_BUTTON(2, 4, 10, "9600", BTF_RADIO_BUTTON)
        + new UIW_BUTTON(2, 5, 10, "4800", BTF_RADIO_BUTTON)
        + new UIW_BUTTON(2, 6, 10, "2400", BTF_RADIO_BUTTON)
        + new UIW_BUTTON(2, 7, 10, "1200", BTF_RADIO_BUTTON)
        + new UIW_BUTTON(2, 8, 10, "300", BTF_RADIO_BUTTON))

    // Add the check boxes.
    + &(*new UIW_GROUP(15, 2, 13, 7, "Setup")
        + new UIW_BUTTON(18, 4, 10, "Full Duplex", BTF_CHECK_BOX)
        + new UIW_BUTTON(18, 5, 10, "XON/XOFF", BTF_CHECK_BOX)
```

```
        + new UIW_BUTTON(18, 6, 10, "Line Wrap", BTF_CHECK_BOX)
        + new UIW_BUTTON(18, 7, 10, "Scroll", BTF_CHECK_BOX)
        + new UIW_BUTTON(18, 8, 10, "Answerback", BTF_CHECK_BOX))

    // Add the bitmapped buttons.
    + &(*new UIW_GROUP(29, 2, 13, 7, "Drives")
        + new UIW_BUTTON(38, 4, 10, "A:", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER | WOF_JUSTIFY_RIGHT, NULL, 0, softDrive)
        + new UIW_BUTTON(38, 5, 10, "B:", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER | WOF_JUSTIFY_RIGHT, NULL, 0, softDrive)
        + new UIW_BUTTON(38, 6, 10, "C:", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER | WOF_JUSTIFY_RIGHT, NULL, 0, hardDrive)
        + new UIW_BUTTON(38, 7, 10, "F:", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER | WOF_JUSTIFY_RIGHT, NULL, 0, networkDrive)
        + new UIW_BUTTON(38, 8, 10, "R:", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER | WOF_JUSTIFY_RIGHT, NULL, 0, networkDrive))

    // Add the regular buttons.
    + new UIW_BUTTON(10, 11, 9, "&Save")
    + new UIW_BUTTON(20, 11, 9, "&Cancel")
    + new UIW_BUTTON(32, 11, 9, "&Help");
```

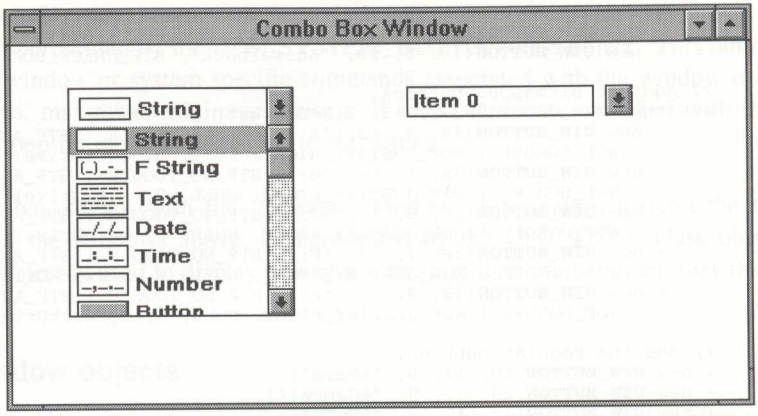
Bitmapped buttons—Buttons displayed with a graphical bitmap. The buttons function the same as regular buttons, but they display a bitmap. Bitmapped buttons may be used in text mode, but the bitmap will not be displayed.

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

Radio buttons—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

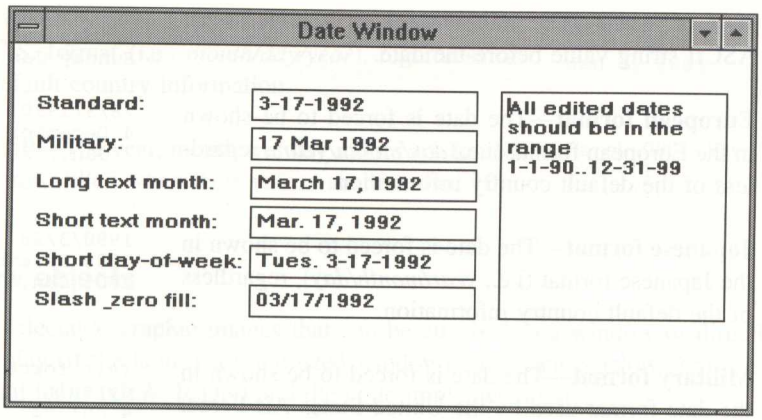
A combo box field is a one line string field with an attached button object. When the button is selected, a vertical list (described below) appears. When an item is selected, it is copied into the initial string field and the menu disappears. In Motif, however, the combo box's vertical list is positioned with the current list item being displayed directly over the combo box's string field. As a result, the top of the vertical list is dynamically positioned according the position of the current list item. The figure below shows a window with two combo boxes (UIW_COMBO_BOX) and the code implementation:




```
*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) and the code implementation:




```
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 is 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 date format used by the United States Air Force, regardless of the default country information. The air force format is ordered by *day month year* where *month* is either a 3 letter abbreviated word and *year* is a two-digit year value (if the DTF_SHORT_YEAR or DTF_SHORT_MONTH flags are set) or *month* is spelled out and *year* is a four-digit value. The air force style is used as the default. However, in order to accommodate the formats used in other branches of the military, other date formatting options (e.g., zero fill, upper case, etc.) may be used in conjunction with the standard military format.

```
(air force style-
default)
4 Jul 91
4 July 1991
```

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.

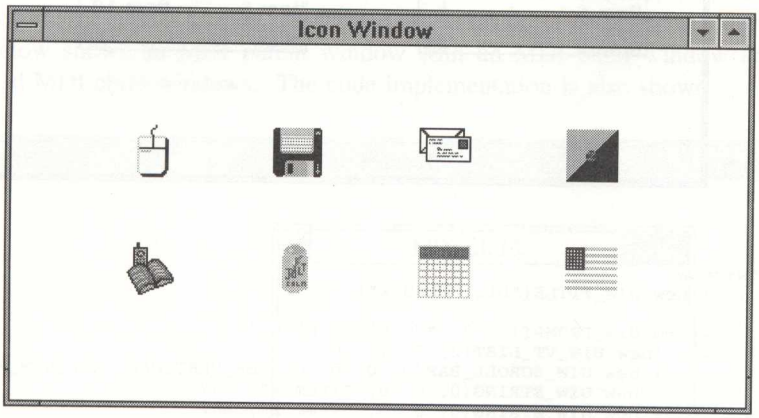
```
March 28, 1990
12/4/1980
Jan 3, 2003
```

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
```

Icon window objects

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) and the code implementation:

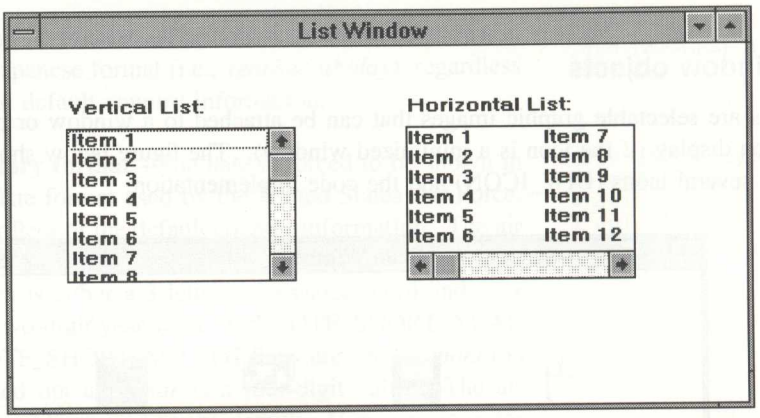



```
*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE("Icon Window")
    + new UIW_ICON(15, 4, "mouse")
    + new UIW_ICON(32, 4, "disk")
    + new UIW_ICON(49, 4, "letter")
    + new UIW_ICON(66, 4, "logo")
    + 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) and the code implementation:

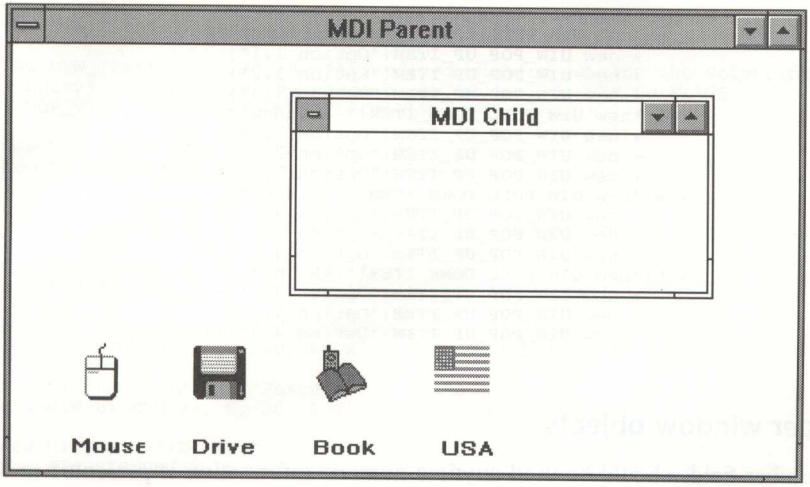



```
*window
    + new UIW_TITLE("List Window")

    + new UIW_PROMPT(2, 2, "Vertical List:")
    + &(*new UIW_VT_LIST(2, 3, 11, 6)
        + new UIW_SCROLL_BAR(0, 0, 0, 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))
    + new UIW_PROMPT(18, 2, "Vertical List:")
    + &(*new UIW_HZ_LIST(18, 3, 11, 6)
        + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_HORIZONTAL,
            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)
        + new UIW_STRING(0, 0, 0, "Item 11", 64)
        + new UIW_STRING(0, 0, 0, "Item 12", 64)
        + new UIW_STRING(0, 0, 0, "Item 13", 64)
        + new UIW_STRING(0, 0, 0, "Item 14", 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 window objects

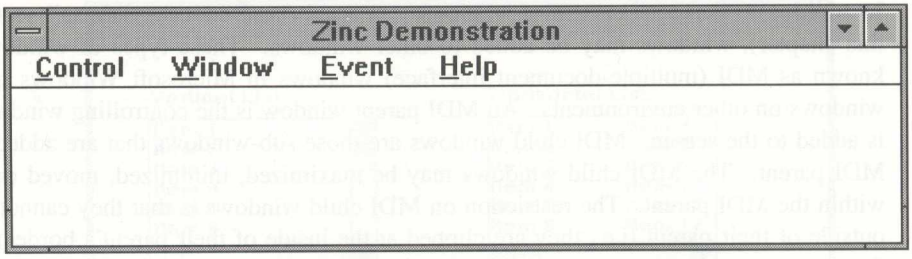
In addition to the standard use of windows (see the "Basic window objects" section of this chapter), windows may be added to other windows. These types of windows are known as MDI (multiple-document interface) windows in Microsoft Windows or sub-windows on other environments. An MDI parent window is 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 an MDI child window and several minimized MDI child windows. The code implementation is also shown:




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

Menu window objects

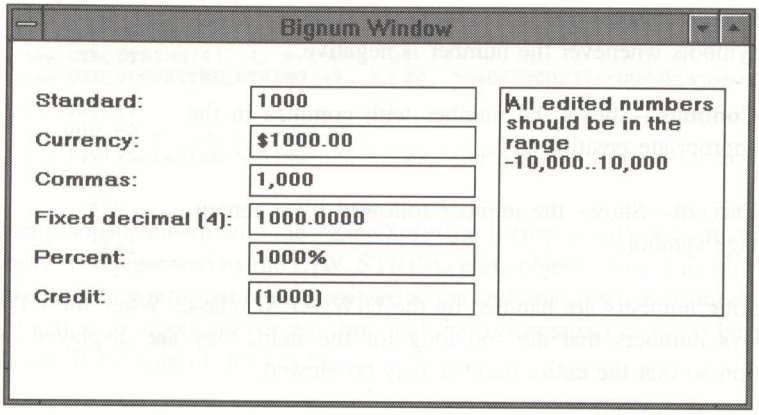
Menus should be used anytime you want to present selection information to the end user. Pull-down items should be used when a hierarchal grouping of selection items is to be used. The pull-down menu serves as the first level in the selection process. The figure below shows a window with a pull-down menu and the code implementation:




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

    + &(*new UIW_PULL_DOWN_MENU
        + &(*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")));
```

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 number fields (UIW_BIGNUM) and the code implementation:




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

    + new UIW_TEXT(43, 1, 20, 6, "All edited numbers (except the Scientific
        entry) 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 (4): ")
    + new UIW_BIGNUM(22, 4, 20, &dvalue, range, NMF_DECIMAL(4))

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

    + new UIW_PROMPT(2, 6, "Credit: ")
    + new UIW_BIGNUM(22, 6, 20, &nvalue, range, NMF_CREDIT);
```

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.

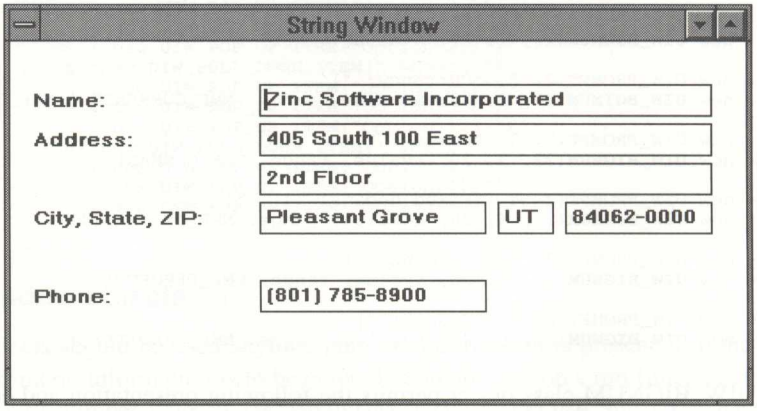
```
$10,000.00
DM100
£195
```

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 Application Framework. 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 and UIW_FORMATTED_STRING) and the code implementation:




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

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

    + new UIW_PROMPT(2, 2, "Address:")
    + new UIW_STRING(22, 2, 41, "405 South 100 East", 256)
    + new UIW_STRING(22, 3, 41, "2nd Floor", 256)
```

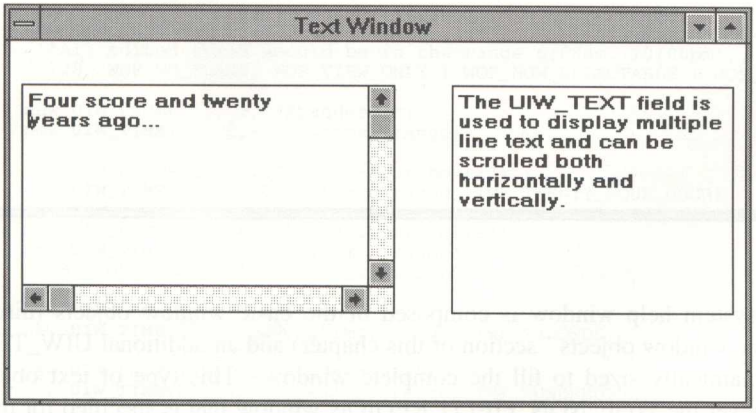
```
+ new UIW_PROMPT(2, 4, "City, State, ZIP:")
+ new UIW_STRING(22, 4, 20, "Pleasant Grove", 256)
+ new UIW_STRING(43, 4, 4, "UT", 3)
+ new UIW_FORMATTED_STRING(49, 4, 20, "840620000", "NNNNNLNNNN",
    ".....-....")

+ new UIW_PROMPT(2, 6, "Phone:")
+ new UIW_FORMATTED_STRING(22, 6, 20, "8017858900", "LNNNLLNNNLXXXX",
    "(...) ...-....")
```

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

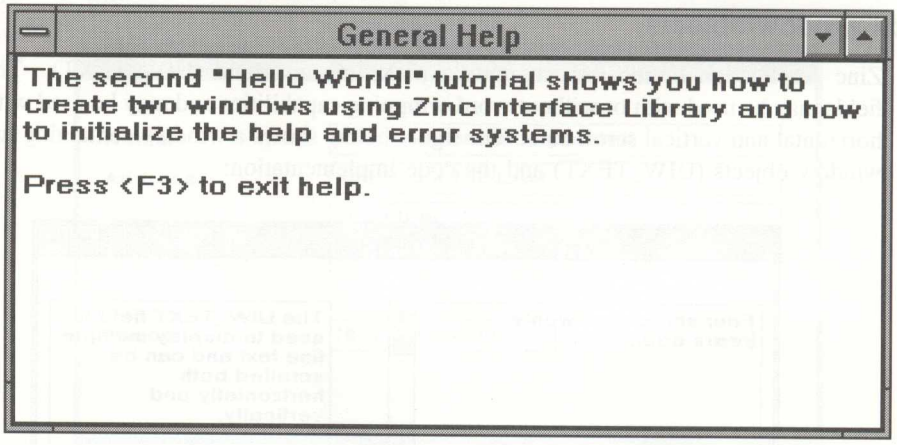
Zinc Application Framework supports a multi-line text field (UIW_TEXT). The text fields may be used with or without word-wrapping capabilities and may be used with both horizontal and vertical scroll bars. The figure below shows a window containing two text window objects (UIW_TEXT) and the code implementation:



```
*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, "Four 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.

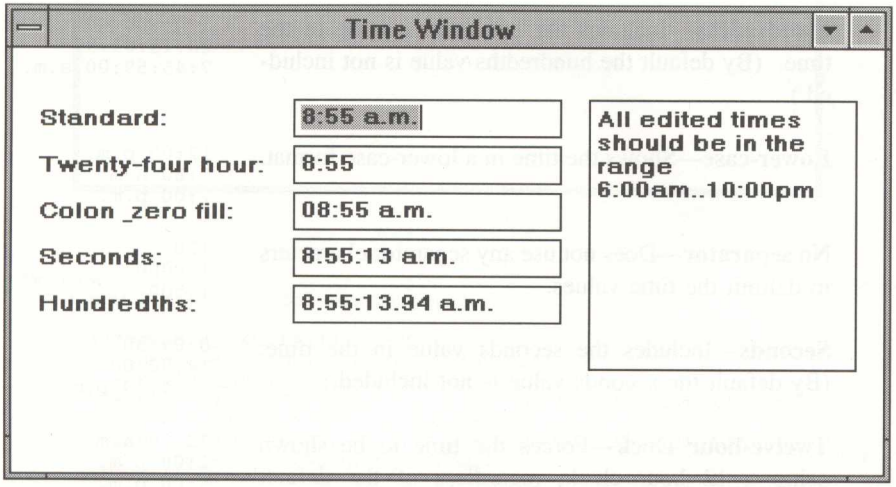
In addition to the standard text field, the UIW_TEXT class permits the creation of a text object that takes the complete window region (inside the border). For example, the graphic image below shows the help window system where the help text is shown in a text object:



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 object.

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) and the code implementation:




```
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.`

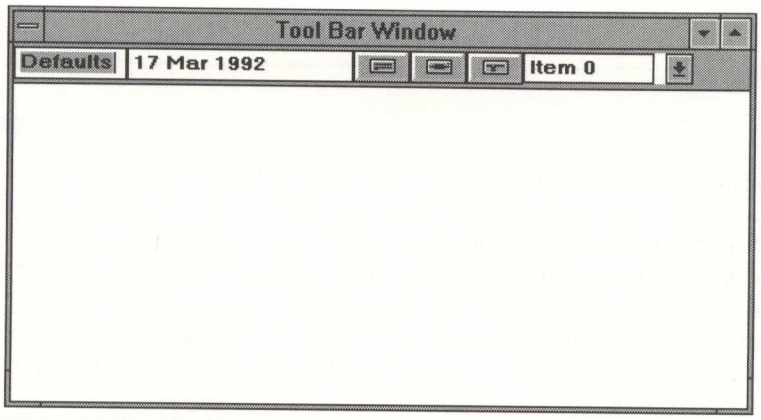
Twenty-four hour clock—Forces the time to be shown using a 24 hour clock, regardless of the default country information.	`12:00`
	`13:00`
	`17:00`

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) and the code implementation:

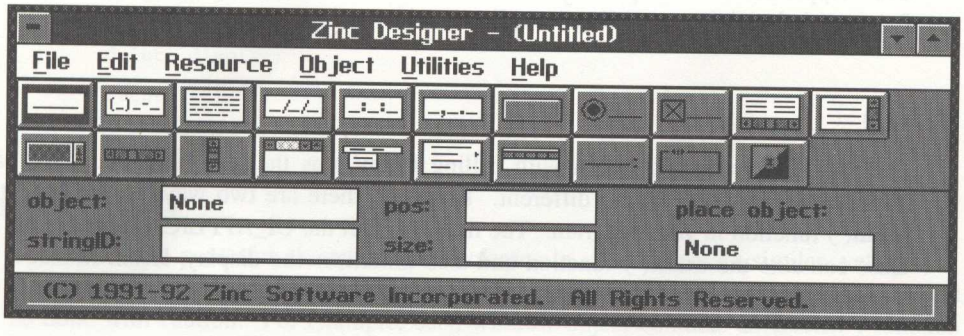



```
*window
    + new UIW_TITLE("Tool Bar Window")

    + &(*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(0, 0, 5, "", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER, NULL, 0, softDrive)
        + new UIW_BUTTON(0, 0, 5, "", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER, NULL, 0, hardDrive)
        + new UIW_BUTTON(0, 0, 5, "", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER, NULL, 0, networkDrive)
        + new UIW_COMBO_BOX(0, 0, 15, 5));
```

CHAPTER 5 – DOS APPLICATIONS

Introduction

If you have purchased Zinc Application Framework for DOS (i.e., Zinc Engine and DOS Key), you will be able to create DOS Text and DOS Graphics applications in real and protected modes. With Zinc Application Framework, DOS Text, DOS Graphics, Microsoft Windows, Windows NT, IBM OS/2 and OSF/Motif applications may be created from the same set of source code. This chapter will discuss those aspects of building a Zinc application that are specific to DOS. The following figure shows the presentation of a Zinc application running under DOS:



Look and feel

In DOS, a Zinc application follows IBM's SAA/CUA specification (where applicable) for both the screen display and input devices. In order to achieve its multi-platform capabilities, Zinc Application Framework abstracts hardware and operating system dependencies. Thus, you don't have to know the intricacies of the environment but can still access them directly if desired.

DOS library

The DOS version of Zinc Application Framework has been compiled into a single library file called **DOS_ZIL.LIB**. When creating a DOS application, **DOS_ZIL.LIB** must be linked into the .EXE file. (**NOTE:** See "Appendix A—Compiler Considerations" in the *Programming Techniques* manual for compiler-specific graphics library and DOS extender library filenames.)

Compiler options

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

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

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

See "Appendix A—Compiler Considerations" in the *Programming Techniques* manual for more information regarding compiler-specific options.

main( )

Ordinary C++ programs begin with calling **main( )** as the first function. Zinc-based applications for DOS are no different. However, there are two ways to implement the **main( )** function in your program. The first is to call the UI_APPLICATION class. This class contains the **main( )** function and also initializes the display, Event Manager and Window Manager. The following code sample demonstrates this technique:

```
#include <ui_win.hpp>

// Creating an instance of UI_APPLICATION causes a main() to be
// linked in automatically.  This main() calls UI_APPLICATION::Main(),
// defined by the programmer.
UI_APPLICATION app(argc, argv);

int UI_APPLICATION::Main(void)
{
    // The UI_APPLICATION constructor automatically initializes the
    // display, eventManager and windowManager variables.
    .
    .
    .
    // Return the exit code.
    return (0);
}
```

The second technique is to create the **main( )** function in your program and initializing the display, Event Manager and Window Manager explicitly. The following code sample demonstrates this technique:

```
int main()
{
    // Initialize the environment dependent display.
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
```

```
    if (!display->installed)
    {
        delete display;
        display = new UI_TEXT_DISPLAY;
    }

    // Create the event manager and add devices.
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;

    // Create the window manager.
    UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANAGER(display,
        eventManager);

    .
    .
    .
    // Clean up.
    delete windowManager;
    delete eventManager;
    delete display;

    return (0);
}
```

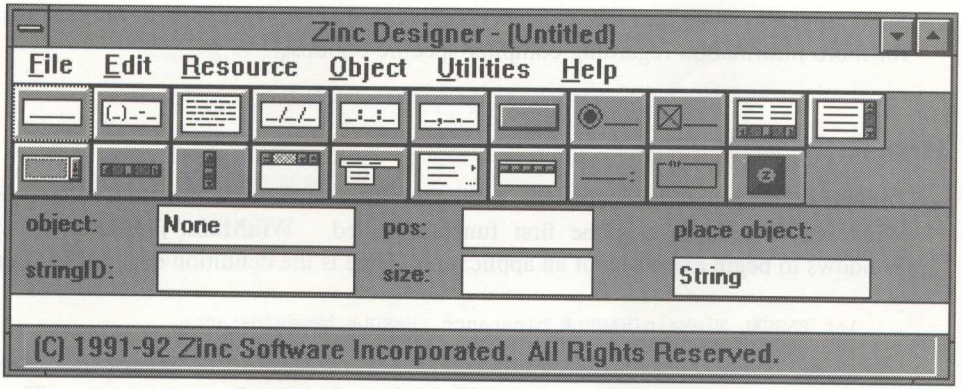
Derived objects

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 window object contains an **Event( )** function that processes messages, and these messages differ between environments. (See the "Help Bar" tutorial in the *Programming Techniques* manual for detailed information on how to create a new object that meets the specifications for all environments.)

CHAPTER 6 – WINDOWS APPLICATIONS

Introduction

If you have purchased Zinc Application Framework for Windows or Windows NT (i.e., Zinc Engine and Windows Key or Windows NT Key), you will be able to create Microsoft Windows 3.X or Windows NT applications. With Zinc Application Framework, DOS Text, DOS Graphics, Microsoft Windows, Windows NT, IBM OS/2 and OSF/Motif applications may be created from the same set of source code. This chapter will discuss those aspects of building a Zinc application that are specific to Windows. The following figure shows the presentation of a Zinc application running under Windows:



Look and feel

In Windows, a Zinc application is an actual Windows application built with actual Windows objects. Because you're creating a Windows application, you have full access to the Windows API and Windows resources. In order to achieve its multi-platform capabilities, however, Zinc Application Framework abstracts the Windows environment. Thus, you don't have to know the Windows API or its messages, but can still access them directly if desired.

Windows library

The Windows version of Zinc Application Framework has been compiled into a single library file called **WIN_ZIL.LIB**. The Windows NT library file is called **WNT_ZIL.-LIB**. When creating a Windows application, **WIN_ZIL.LIB** (or **WNT_ZIL.LIB** for Windows NT) must be linked into the .EXE file.

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 other environments in addition to Windows (or Windows NT), you should select the compiler option to create the application as a Windows (or Windows NT) executable program.

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

See "Appendix A—Compiler Considerations" in the *Programming Techniques* manual for more information regarding compiler-specific options.

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);
```

There are two ways to implement the **WinMain( )** function in your program. The first is to call the UI_APPLICATION class. This class contains the **WinMain( )** function and also initializes the display, Event Manager and Window Manager. The following code sample demonstrates this technique:

```
#include <ui_win.hpp>

// Creating an instance of UI_APPLICATION causes a WinMain() to be
// linked in automatically.  This WinMain() calls UI_APPLICATION::Main(),
// defined by the programmer.

UI_APPLICATION app(argc, argv);

int UI_APPLICATION::Main(void)
{
    // The UI_APPLICATION constructor automatically initializes the
    // display, eventManager and windowManager variables.
    .
    .
    .
    // Return the exit code.
    return (0);
}
```

The second technique is to create the **WinMain( )** function in your program and initializing the display, Event Manager and Window Manager explicitly. The following code sample demonstrates this technique:

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpszCmdLine, int nCmdShow)
{
    // Initialize the environment dependent display.
    UI_DISPLAY *display = new UI_MSWINDOWS_DISPLAY(hInstance, hPrevInstance,
        nCmdShow);

    // Create the event manager and add devices.
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;

    // Create the window manager.
    UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANAGER(display,
        eventManager);
    .
    .
    .

    // Clean up.
    delete windowManager;
    delete eventManager;
    delete display;

    return (0);
}
```

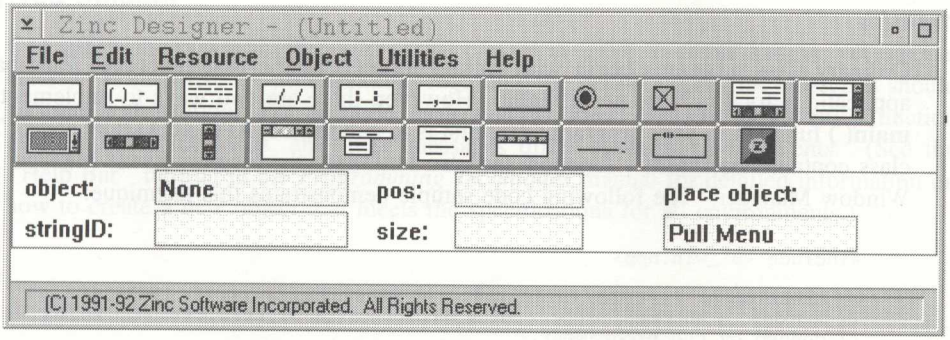
Derived objects

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 window object contains an **Event( )** function that processes messages, and these messages differ between environments. (See the "Help Bar" tutorial in the *Programming Techniques* manual for detailed information on how to create a new object that meets the specifications for all environments.)

CHAPTER 7 – OS/2 APPLICATIONS

Introduction

If you have purchased Zinc Application Framework for OS/2 (i.e., Zinc Engine and OS/2 Key), you will be able to create OS/2 applications. With Zinc Application Framework, DOS Text, DOS Graphics, Microsoft Windows, Windows NT, IBM OS/2 and OSF/Motif applications may be created from the same set of source code. This chapter will discuss those aspects of building a Zinc application that are specific to OS/2. The following figure shows the presentation of a Zinc application running under OS/2:



Look and feel

In OS/2, a Zinc application is an actual OS/2 application built with actual OS/2 objects. Because you're creating an OS/2 application, you have full access to the OS/2 API and OS/2 resources. In order to achieve its multi-platform capabilities, however, Zinc Application Framework abstracts the OS/2 environment. Thus, you don't have to know the OS/2 API or its messages, but can still access them directly if desired.

OS/2 library

The OS/2 version of Zinc Application Framework has been compiled into a single library file called **OS2_ZIL.LIB**. When creating an OS/2 application, **OS2_ZIL.LIB** must be linked into the .EXE file.

Compiler options

When creating an OS/2 application, the following compiler options should be selected:

OS/2 application—If your compiler is able to create applications for other environments in addition to OS/2, you should select the compiler option to create the application as an OS/2 executable program.

See "Appendix A—Compiler Considerations" in the *Programming Techniques* manual for more information regarding compiler-specific options.

main( )

Ordinary C++ programs begin with calling **main( )** as the first function. Zinc-based applications for OS/2 are no different. However, there are two ways to implement the **main( )** function in your program. The first is to call the UI_APPLICATION class. This class contains the **main( )** function and also initializes the display, Event Manager and Window Manager. The following code sample demonstrates this technique:

```
#include <ui_win.hpp>

// Creating an instance of UI_APPLICATION causes a main() to be
// linked in automatically.  This main() calls UI_APPLICATION::Main(),
// defined by the programmer.

UI_APPLICATION app(argc, argv);

int UI_APPLICATION::Main(void)
{
    // The UI_APPLICATION constructor automatically initializes the
    // display, eventManager and windowManager variables.
    .
    .
    .
    // Return the exit code.
    return (0);
}
```

The second technique is to create the **main( )** function in your program and initializing the display, Event Manager and Window Manager explicitly. The following code sample demonstrates this technique:

```
main()
{
    // Initialize the environment dependent display.
    UI_DISPLAY *display = new UI_OS2_DISPLAY;

    // Create the event manager and add devices.
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
```

```
    // Create the window manager.
    UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANAGER(display,
        eventManager);
    .
    .
    .
    // Clean up.
    delete windowManager;
    delete eventManager;
    delete display;

    return (0);
}
```

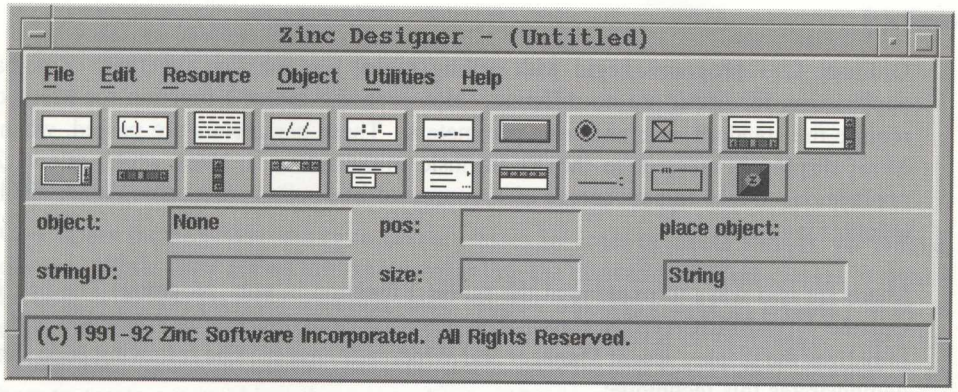
Derived objects

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 window object contains an **Event( )** function that processes messages, and these messages differ between environments. (See the "Help Bar" tutorial in the *Programming Techniques* manual for detailed information on how to create a new object that meets the specifications for all environments.)

CHAPTER 8 – MOTIF APPLICATIONS

Introduction

If you have purchased Zinc Application Framework for Motif (i.e., Zinc Engine and Motif Key), you will be able to create Motif applications. With Zinc Application Framework, DOS Text, DOS Graphics, Microsoft Windows, Windows NT, IBM OS/2 and OSF/Motif applications may be created from the same set of source code. This chapter will discuss those aspects of building a Zinc application that are specific to Motif. The following figure shows the presentation of a Zinc application running under Motif:



Look and feel

In Motif, a Zinc application is an actual Motif application built with actual Motif widgets. Because you're creating a Motif application, you have full access to the Motif toolkit, Xt Intrinsics, X Library and all X resources. In order to achieve its multi-platform capabilities, however, Zinc Application Framework abstracts the host environment. Thus, you don't have to know the native API or its messages, but can still access them directly if desired.

Motif library

The Motif version of Zinc Application Framework has been compiled into a single library file called **lib_mtf_zil.a**. When creating a Motif application, **lib_mtf_zil.a** must be linked into the executable file. (**NOTE**: Some source code changes may be required to use the Motif Key on hardware platforms that are not directly supported by Zinc. See the **README** file for a list of currently supported hardware platforms.)

Compiler options

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

Motif application—If your compiler is able to create applications for other environments in addition to Motif, you should select the compiler option to create the application as a Motif executable program.

See "Appendix A—Compiler Considerations" in the *Programming Techniques* manual for more information regarding compiler-specific options.

main( )

Ordinary C++ programs begin with calling **main( )** as the first function. Zinc-based applications for Motif are no different. However, the **main( )** function for Motif does require the standard *argc* and *argv* parameters. These parameters are passed, when the Motif display is created, to the Xt Intrinsic initialization routines which allows Zinc applications to take full advantage of X command-line options (e.g., using other displays, colors, fonts, etc.).

There are two ways to implement the **main( )** function in your program. The first is to call the UI_APPLICATION class. This class contains the **main( )** function and also initializes the display, Event Manager and Window Manager. The following code sample demonstrates this technique:

```
#include <ui_win.hpp>

// Creating an instance of UI_APPLICATION causes a main() to be
// linked in automatically.  This main() calls UI_APPLICATION::Main(),
// defined by the programmer.

UI_APPLICATION app(argc, argv);

int UI_APPLICATION::Main(void)
{
    // The UI_APPLICATION constructor automatically initializes the
    // display, eventManager and windowManager variables.
    .
    .
    .
    // Return the exit code.
    return (0);
}
```

The second technique is to create the **main( )** function in your program and initializing the display, Event Manager and Window Manager explicitly. The following code sample demonstrates this technique:

```
int main(int argc, char *argv[])
{
    // Initialize the display.
    UI_DISPLAY *display = new UI_MOTIF_DISPLAY(&argc, argv);

    // Initialize the event manager.
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;

    // Initialize the window manager.
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    // Clean up.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

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 window object contains an **Event( )** function that processes messages, and these messages differ between environments. (See the "Help Bar" tutorial in the *Programming Techniques* manual for detailed information on how to create a new object that meets the specifications for all environments.)

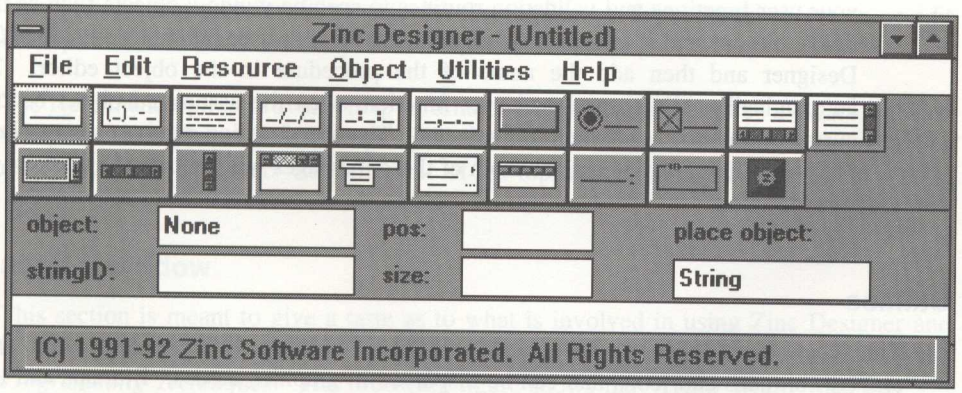
CHAPTER 9 – ZINC DESIGNER

This chapter provides an introduction of Zinc Designer and a brief overview of its usage. For a complete description of Zinc Designer, see "Chapter 21—Getting Started" of the *Programming Techniques* manual.

Zinc 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 Application Framework contrasts sharply with this a la carte approach.

Zinc Designer is an interactive design tool that was created using Zinc Application Framework 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. The output generated by Zinc Designer is portable between all compilers and operating platforms that are supported by Zinc.

The following figure shows the presentation of Zinc Designer:



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 the system button. 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 that is saved as a single resource.

Default Information—A feature of many object editors which allows 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—Can be assigned both at the object and window levels. If you do not assign a specific help context to an object the object will use the help context that is assigned 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 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 calls an assigned user function when the associated object becomes current, non-current or when it is selected. The user function receives a control code that will allow it to 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 that can be displayed in 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.

The Help Editor allows you to create and edit help contexts (i.e., help text with a user-defined help identification). Help contexts that you create with the Help Editor are assigned to windows or objects through the object editors.

Getting around

File operations are located in the File menu option. Here you will find options allowing files to be created, deleted, opened or saved. Under the File | Preferences option, you can set options to control the use of graphics and the number of backups performed by Zinc Designer.

The Edit menu option contains items used to edit existing window objects. The features provided include: cut and paste, object deletion, moving, sizing and invoking the object editors.

The Resource menu option contains items allowing resources to be created, deleted, edited, loaded, saved or tested. In Zinc Designer, a resource is a window that may contain various window objects. When a resource is tested, it is temporarily given the ability to function as it would within a user program. During the testing phase, no other designer features are available.

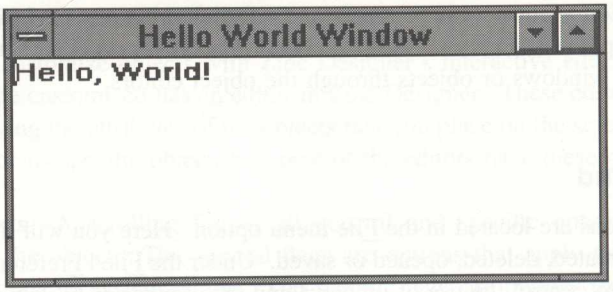
The Object menu option provides a means of selecting objects to be added to a resource. This is used in addition to the tool bar of window objects displayed on the Zinc Designer control window.

The Utilities menu option is used to invoke the Image Editor (used to create or edit bitmaps and icons) and the Help Editor (used to create help contexts).

The Help menu option provides help information regarding the features available in Zinc Designer. The Help | Index brings up a window containing a horizontal list of topics relating to Zinc Designer. Selecting one of the topics will display a related help screen.

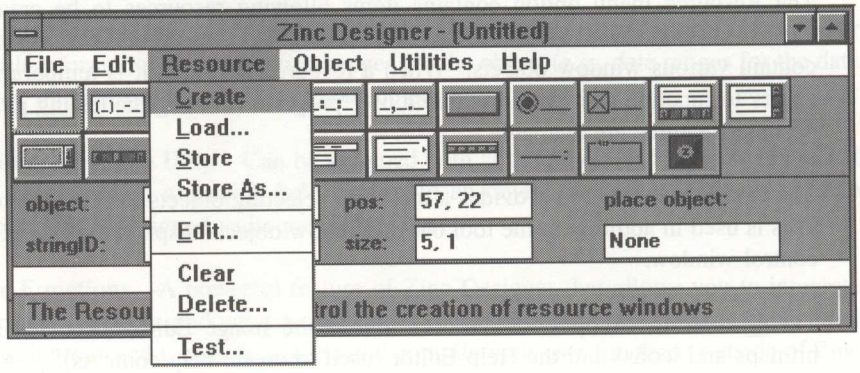
Creating a window

This section is meant to give a taste as to what is involved in using Zinc Designer and is an adaptation of a similar section in "Chapter 3—Using Zinc Designer" of the *Programming Techniques*. The following figure shows the "Hello World Window!":

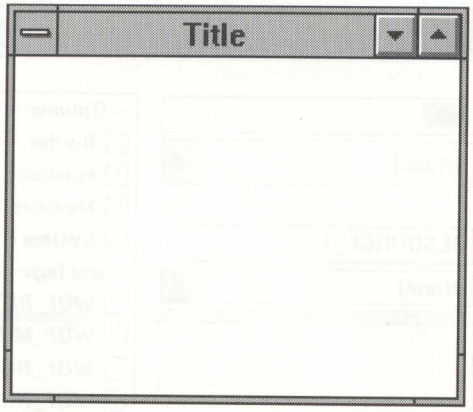


This window is created interactively with the following steps:

1—Select "Resource" from the menu on Zinc Designer's main control window. Selecting this option causes the following pop-up menu to be displayed:

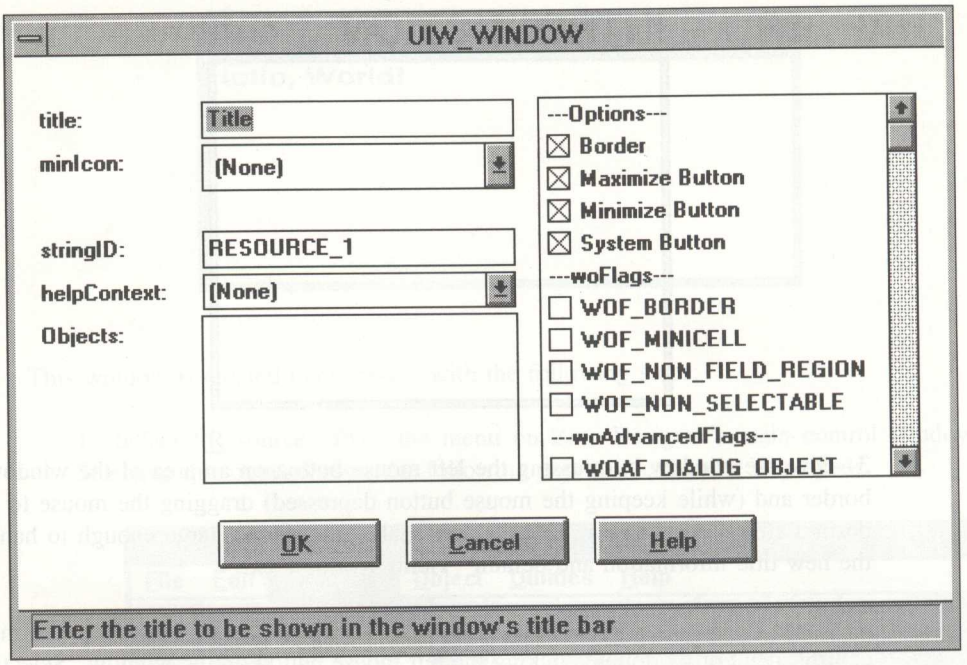


2-Select "Create" from the pop-up menu. At this point a new resource (i.e., a generic window) appears on the screen:



3—Size the window by pressing the left mouse button on an area of the window's border and (while keeping the mouse button depressed) dragging the mouse to the desired size of the window. You should make the window large enough to handle the new title information and default "Hello World!" text.

4—Enter an identification for the window by selecting Edit | Object from the main control menu or by double clicking the left mouse button on the window. Selecting this option causes the window editor to be displayed:

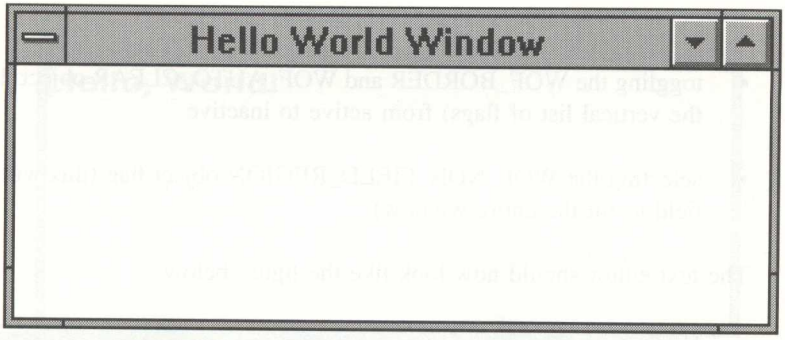


5—Enter `Hello World Window` in the "title:" field.

6—Enter the window identification by typing `HELLO_WORLD_WINDOW` in the field adjacent to the "stringID:" prompt.

7—Save the identification by selecting the "OK" button.

Your window should now look similar to the figure below:

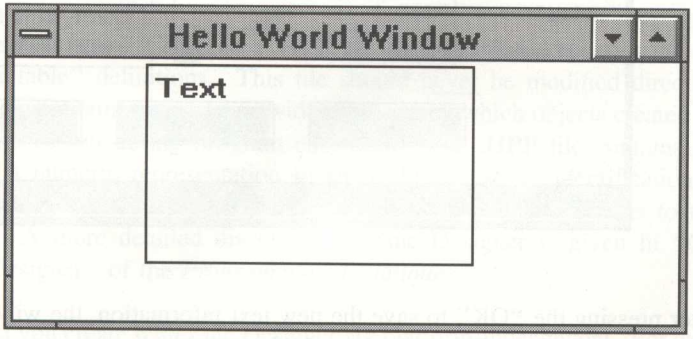


Creating a window object

Creation of the "Hello World!" text is similar to the window creation described above:

1—Select Object | Input | Text from the main control menu.

2—Place the text object in the middle of the "Hello World!" window by positioning the mouse cursor within the "Hello World Window" and clicking with the left mouse button. Your window should now have a text field within its border:

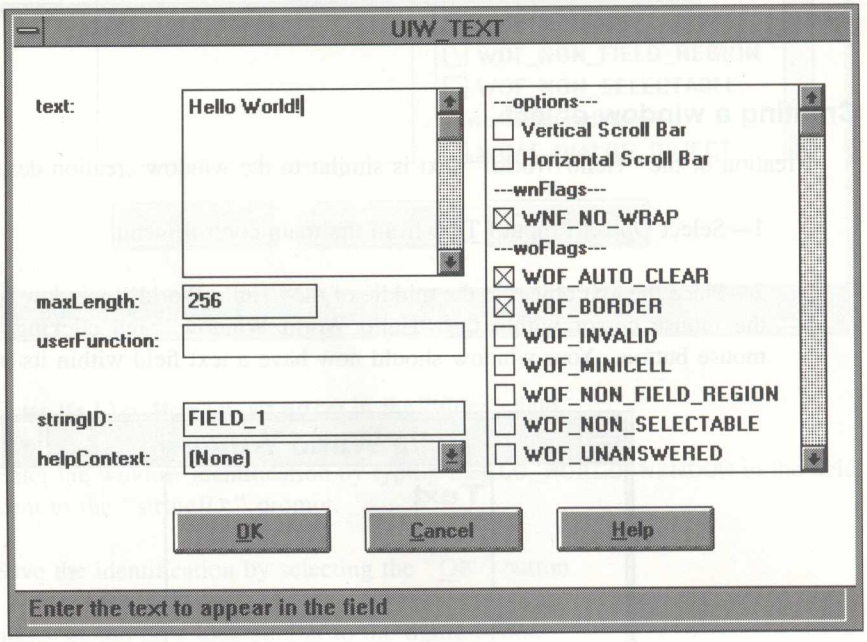


3—Change the default information associated with the text object by:

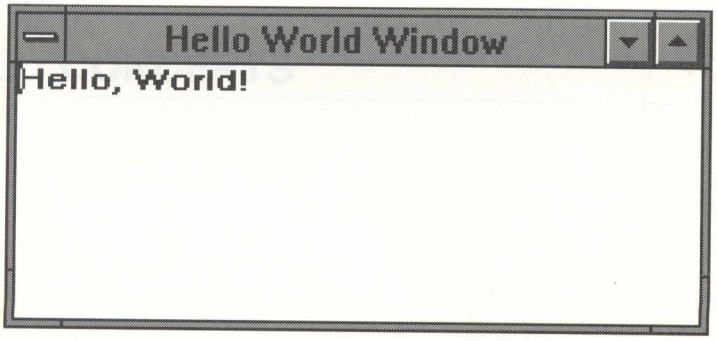
- calling the text editor (e.g., by clicking the left mouse button twice on the text object)
- typing `Hello World!` in the field under the "text:" prompt

- typing `256` in the field adjacent to the "maxLength:" prompt
- toggling the WOF_BORDER and WOF_AUTO_CLEAR object flags (located in the vertical list of flags) from active to inactive
- selecting the WOF_NON_FIELD_REGION object flag (this will cause the text field to fill the entire window)

The text editor should now look like the figure below:



After pressing the "OK" to save the new text information, the window should look like the window shown at the beginning of this section:



Zinc Designer files

After creating your screens (or resources) you save them to disk by selecting File | Save. The resource data will be stored in a file with a **.DAT** extension. You can 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");
```

Zinc Designer also outputs a **.CPP** and a **.HPP** file. The **.CPP** file contains the "object table" and "user table" definitions. This file should never be modified directly by a programmer since it contains code that provides a means by which objects created in Zinc Designer can be accessed during program execution. The **.HPP** file contains defined values that give a numeric representation to each object's string identification. This allows a programmer to use defined constants rather that character strings to refer to window objects. A more detailed discussion of Zinc Designer is given in "Chapter 3—Using Zinc Designer" of the *Programming Techniques*.

The resources that you create with Zinc Designer are platform-independent. For example, resources you create with the Windows version of Zinc Designer can be opened and edited with the DOS, OS/2 or Motif versions and vice versa.

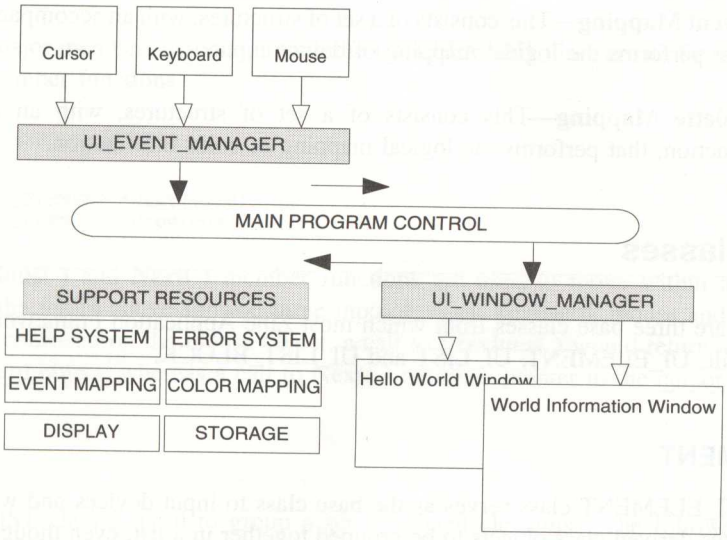
Zinc Designer's complete access to the Zinc class library, straightforward integration of your code and platform-independent storage can dramatically enhance your productivity.

SECTION III
ADVANCED CONCEPTS

CHAPTER 10 – ZINC LIBRARY CLASSES

The purpose of the chapters in this section is to familiarize you with the class objects and C++ features used throughout Zinc Application Framework. 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 Application Framework model:



These concepts, as well as others that are fundamental to the operation of Zinc Application Framework, will be discussed in this chapter. They include:

Base Classes—These are the core classes used within Zinc Application Framework. 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 (e.g., 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 (e.g., 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 inform the user of 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 device input.

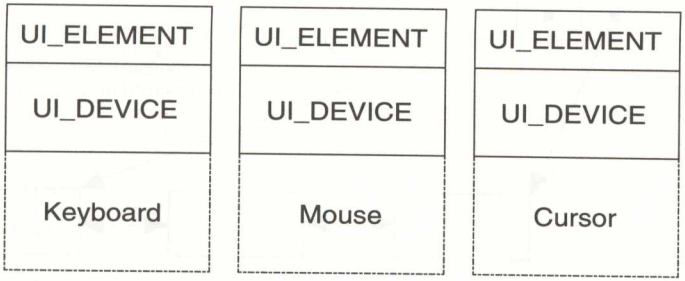
Palette Mapping—This consists of a set of structures, with an accompanying function, that performs the logical mapping of color information.

Base Classes

There are three base classes from which most Zinc Application Framework 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:



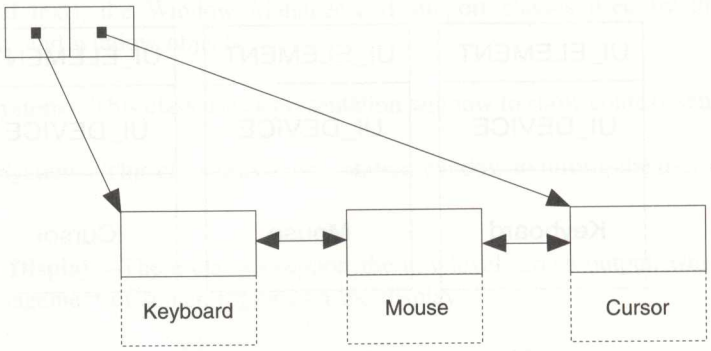
The two major members associated with the UI_ELEMENT class are its **Previous( )** and **Next( )** member functions.

```
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 members associated with the UI_LIST class: its **First( )**, **Last( )**, **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.

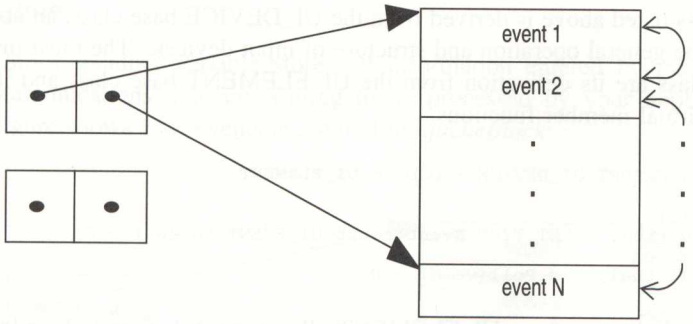
The **Add( )** and **Subtract( )** member functions, along with the **+** and **–** operator overloads, are used to add or subtract list elements to and from the current list object. For example,

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:



The UI_ELEMENT, UI_LIST and UI_LIST_BLOCK 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 aspects of 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 Application Framework:

UID_CURSOR
UID_KEYBOARD
UID_MOUSE
UID_PENDOS

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, Windows NT, OS/2 and Motif versions. 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.

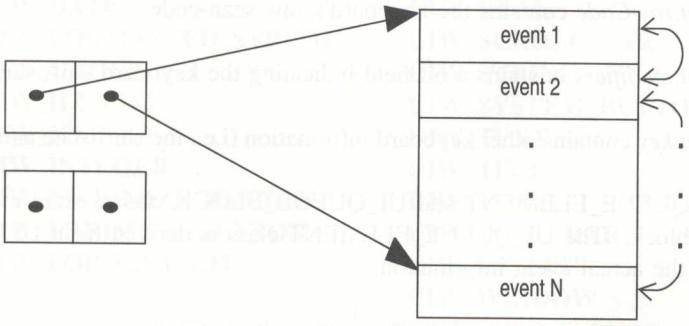
D_POSITION—Changes the position of a device. For example, if the device receiving this message were a cursor, the position of the blinking cursor would be changed to the screen position given by *event.position*.

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 see if any keys have been pressed by the user. If a key has been pressed, the **UID_KEYBOARD::Poll( )** routine places the key event 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.
    RAW_CODE rawCode;
    RAW_CODE modifiers;
#if defined (ZIL_MSWINDOWS)
    MSG message;
#elif defined (ZIL_OS2)
    QMSG message;
#elif defined (ZIL_MOTIF)
    typedef XEvent message;
#endif

    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.modifiers* contains a bit field indicating the keyboard shift-states.

- *event.key* contains other keyboard information (i.e., the shift state and 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 aspects of 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. The following window objects are defined by Zinc Application Framework:

UIW_BIGNUM
UIW_BORDER
UIW_BUTTON
UIW_COMBO_BOX
UIW_DATE
UIW_FORMATTED_STRING
UIW_GROUP
UIW_HZ_LIST
UIW_ICON
UIW_INTEGER
UIW_MAXIMIZE_BUTTON
UIW_MINIMIZE_BUTTON
UIW_POP_UP_ITEM
UIW_POP_UP_MENU
UIW_PROMPT
UIW_PULL_DOWN_ITEM
UIW_PULL_DOWN_MENU
UIW_REAL
UIW_SCROLL_BAR
UIW_STRING
UIW_SYSTEM_BUTTON
UIW_TEXT
UIW_TIME
UIW_TITLE
UIW_TOOL_BAR
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 complement 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 three 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.

Environment Specific—These events are generated by the operating system or host environment in which the Zinc application is running. For example, when running in Windows, Zinc classes understand and interpret **WM_** messages (e.g., WM_-PAINT, etc.) or any of the other Windows messages. The same holds true for other operating environments (e.g., Motif and OS/2).

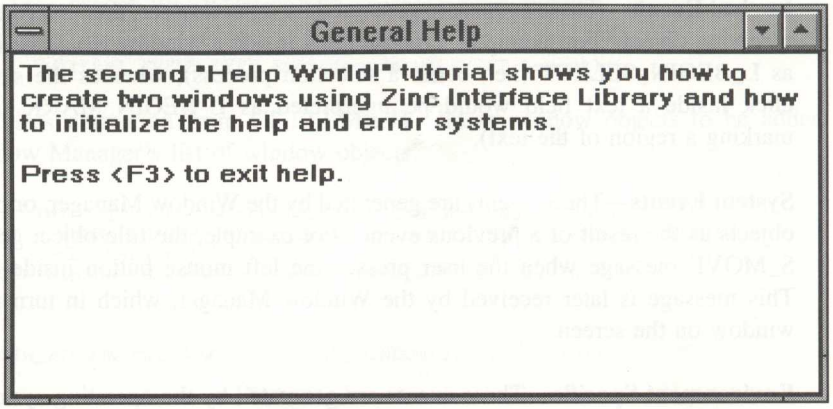
Help System

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:



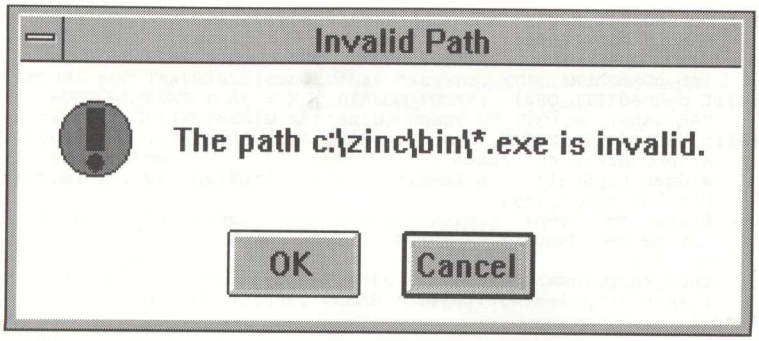
The help window system's **DisplayHelp( )** member function provides context sensitive help information during an application. Each help context contains a title (shown on the title bar), and a help message (shown in the text portion of the window). The *helpContext* argument is used as an identifier 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, **Report-Error( )**.

```
class EXPORT UI_ERROR_SYSTEM
{
public:
    virtual UIS_STATUS ReportError(UI_WINDOW_MANAGER *windowManager,
        UIS_STATUS errorStatus, char *format, ...);
```

The UI_ERROR_SYSTEM class uses a UIW_WINDOW object or an environment specific error handling mechanism to present error information to the screen. A picture of this window is shown below (where an invalid path name 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 Application Framework:

UI_BGI_DISPLAY	**UI_MSC_DISPLAY**
UI_FG_DISPLAY	**UI_MSWINDOWS_DISPLAY**
UI_GRAPHICS_DISPLAY	**UI_OS2_DISPLAY**
UI_MOTIF_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.

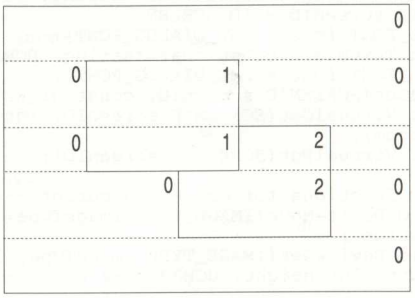
```
class EXPORT UI_DISPLAY
{
public:
    int installed;
    int isText;
    int isMono;
    int columns, lines;
    int cellWidth, cellHeight;
    int preSpace, postSpace;
    long miniNumeratorX, miniDenominatorX;
    long miniNumeratorY, miniDenominatorY;

    static UI_PALETTE *backgroundPalette;
    static UI_PALETTE *xorPalette;
    static UI_PALETTE *colorMap;
```

```
#if defined(ZIL_MSWINDOWS)
    HANDLE hInstance;
    HANDLE hPrevInstance;
    int nCmdShow;
#elif defined(ZIL_OS2)
    HAB hab;
#elif defined(ZIL_MOTIF)
    XtAppContext appContext;
    Widget topShell;
    Display *xDisplay;
    Screen *xScreen;
    int xScreenNumber;
    GC xGc;
    char *appClass;
    Pixmap interleaveStipple;
#endif

    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, HBITMAP *colorBitmap = NULL,
        HBITMAP *monoBitmap = NULL);
    virtual void BitmapArrayToHandle(SCREENID screenID, int bitmapWidth,
        int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette, HBITMAP *colorBitmap,
        HBITMAP *monoBitmap);
    virtual void BitmapHandleToArray(SCREENID screenID, HBITMAP colorBitmap,
        HBITMAP monoBitmap, int *bitmapWidth, int *bitmapHeight,
        UCHAR **bitmapArray);
    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);
    virtual void IconArrayToHandle(SCREENID screenID, int iconWidth,
        int iconHeight, const UCHAR *iconArray, const UI_PALETTE *palette,
        HICON *icon);
    virtual void IconHandleToArray(SCREENID screenID, HICON icon,
        int *iconWidth, int *iconHeight, UCHAR **iconArray);
    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);
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground);
    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);
    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);
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion);
    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);
    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,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
```

```
        virtual int TextHeight(const char *string,
            SCREENID screenID = ID_SCREEN,
            LOGICAL_FONT font = FNT_DIALOG_FONT);
        virtual int TextWidth(const char *string, SCREENID screenID = ID_SCREEN,
            LOGICAL_FONT font = FNT_DIALOG_FONT);
        int VirtualGet(SCREENID screenID, const UI_REGION &region);
        virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
            int bottom);
        virtual int VirtualPut(SCREENID screenID);

        // ADVANCED functions for mouse and cursor --- DO NOT USE! ---
        virtual void DeviceMove(IMAGE_TYPE imageType, int newColumn,
            int newLine);
        virtual void DeviceSet(IMAGE_TYPE imageType, int column, int line,
            int width, int height, UCHAR *image);

    protected:
        struct EXPORT UI_DISPLAY_IMAGE
        {
            UI_REGION region;
            UCHAR *image;
            UCHAR *screen;
            UCHAR *backup;
        };

        UI_DISPLAY_IMAGE displayImage[MAX_DISPLAY_IMAGES];

        UI_DISPLAY(int isText);
        int RegionInitialize(UI_REGION &region, const UI_REGION *clipRegion,
            int left, int top, int right, int bottom);
    };
```

Region lists (DOS version only)

The DOS display 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;

    #if defined(ZIL_MSWINDOWS)
        void Assign(const RECT &rect);
    #elif defined(ZIL_OS2)
        void Assign(const RECTL &rect);
#endif
        int Encompassed(const UI_REGION &region);
        int Height(void);
        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 Width(void);
        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.

NOTE: Some operating environments (e.g., MS Windows, Windows NT, OS/2 and Motif) handle clipping internally. Therefore, their display classes are not derived from UI_REGION_LIST.

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, Microsoft Windows, Windows NT, IBM OS/2 and OSF/Motif 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;                        // The type of event.
    RAW_CODE rawCode;
    RAW_CODE modifiers;
#if defined (ZIL_MSWINDOWS)
    MSG message;
#elif defined (ZIL_OS2)
    QMSG message;
#elif defined (ZIL_MOTIF)
    typedef XEvent message;
#endif
    .
    .
    .
};
```

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;
    RAW_CODE modifiers;

    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*, *event.rawCode* and *event.modifiers* 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 Application Framework: *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 *logicalPalette*, the entry's *palette* is returned.

NOTE: There are three pre-defined palette map tables used in Zinc Application Framework: *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 11 – C++ FEATURES

In this chapter we will look at Zinc Application Framework 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 Application Framework.

3—Methods used to destroy a class object. These methods correspond to the construction methods described, as well as some implementation details Zinc Application Framework uses in conjunction with virtual destructors.

4—Definition of the types of member variables used by Zinc Application Framework. 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 Application Framework. This includes the many features provided by C++ including the use of default arguments, virtual member functions, overloaded functions, pointers to 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 identifies the class whose definitions follow. For example, the UI_LIST class has the following lead information:

```
// ----- UI_LIST ---------------------------------------
class EXPORT UI_LIST
    .
    .
    .
```

2—Second, all class definitions are preceded by the reserved word **class**, an environment specific identifier, **EXPORT**, and one of the Zinc prefixes (i.e., "UI_", "UID_" and "UIW_"). The reserved word class tells the compiler that the definition not only has structural information, but also contains unique information that constitutes a class, such as member functions, single and multiple inheritance, pointers to member functions, etc.

The keyword EXPORT is not part of the C++ language; rather it is a typedef used by Zinc to facilitate portability. In Windows, for example, classes are defined as class HUGE UI_ELEMENT. In DOS, the class is defined as class UI_ELEMENT. In order to provide a single set of source, EXPORT is inserted in the class definition and typedef'ed according to the requirements of a specific environment.

The prefix "UI_" is used to indicate a "**U**ser **I**nterface" type class. The prefix "UID_" is used to indicate a "**U**ser **I**nterface **D**evice" type class. The prefix "UIW_" is used to indicate a "**U**ser **I**nterface **W**indow object" 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 Application Framework. 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 class 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 members are not documented in any of the Zinc Application Framework manuals.

The UI_DEVICE class shows how this member access order is followed:

```
class EXPORT UID_KEYBOARD : public UI_DEVICE
{
public:
    static EVENT_TYPE breakHandlerSet;

    UID_KEYBOARD(DEVICE_STATE state = D_ON);
    virtual ~UID_KEYBOARD(void);
    virtual EVENT_TYPE Event(const UI_EVENT &event);

protected:
    virtual void Poll(void);
};
```

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
{
    friend class EXPORT UIF_BUTTON;
public:
    BTF_FLAGS btFlags;
    EVENT_TYPE value;

    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);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    char *DataGet(int stripText = FALSE);
    void DataSet(char *text);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);
    static EVENT_TYPE Message(UI_WINDOW_OBJECT *object, UI_EVENT &event,
        EVENT_TYPE ccode);
    .
    .
    .
```

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

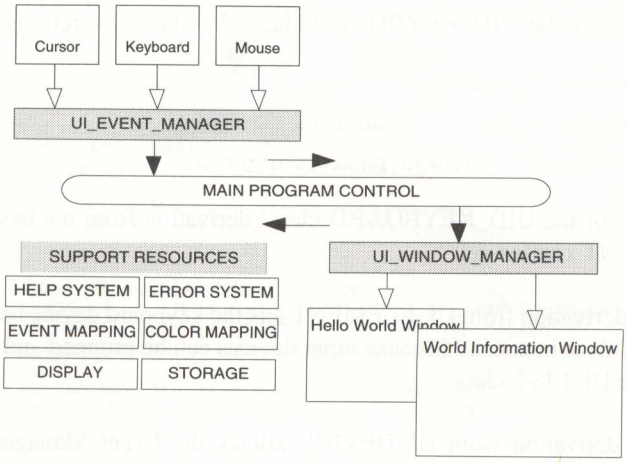
maintenance of code. For a full explanation of these rules see "Appendix D—Zinc Coding Standards" of the *Programming Techniques* manual.

Base classes

Base classes are used to define the core operation of objects within an application. The core of Zinc Application Framework is contained in two base classes: UI_ELEMENT and UI_LIST. The definition of these two classes (i.e., their public and protected members) is given below:

```
class EXPORT UI_ELEMENT
{
    friend class EXPORT UI_LIST;
public:
    virtual ~UI_ELEMENT(void);
    int ListIndex(void);
    UI_ELEMENT *Next(void);
    UI_ELEMENT *Previous(void);

protected:
    UI_ELEMENT *previous, *next;

    UI_ELEMENT(void);
};

class EXPORT UI_LIST
{
    friend class EXPORT UI_LIST_BLOCK;
public:
    int (*compareFunction)(void *element1, void *element2);

    UI_LIST(int (*_compareFunction)(void *element1, void *element2) = NULL);
    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;
};
```

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 Application Framework components. For example, you may recall the general Zinc Application Framework 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. Since the UI_EVENT_MANAGER class is derived from UI_LIST, it is able to maintain a list of all attached devices. This derivation also 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 EXPORT 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 when running in DOS) 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. (**NOTE:** This only applies to the DOS environment.)

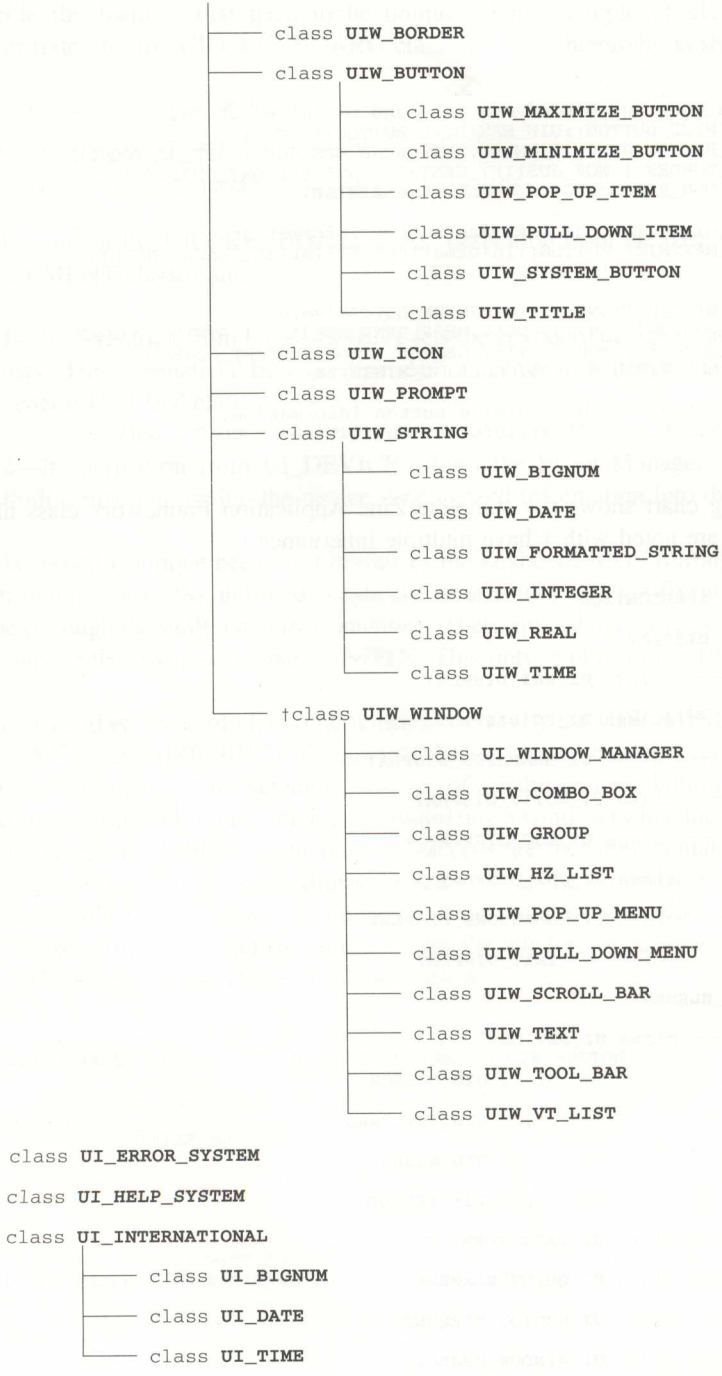
Another good example of class inheritance is shown by the UIW_MINIMIZE_BUTTON and UIW_MAXIMIZE_BUTTON classes. These classes exhibit very similar behavior because they appear on the screen in the form of a 3-dimensional button and because they perform their operation (maximizing or minimizing a window) when they are clicked upon by the left mouse button. Their only differences are that they contain different screen characters (e.g., '▲' for the maximize button, '▼' for the minimize button in DOS and Windows) 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:

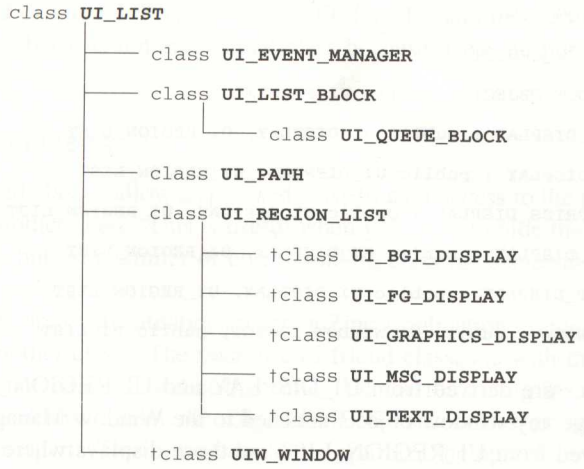
```
// ----- UIW_MAXIMIZE_BUTTON
class EXPORT UIW_MAXIMIZE_BUTTON : public UIW_BUTTON
{
    friend class EXPORT UIF_MAXIMIZE_BUTTON;
public:
    UIW_MAXIMIZE_BUTTON(void);
    .
    .
    .
};

// ----- UIW_MINIMIZE_BUTTON
class EXPORT UIW_MINIMIZE_BUTTON : public UIW_BUTTON
{
    friend class EXPORT UIF_MINIMIZE_BUTTON;
public:
    UIW_MINIMIZE_BUTTON(void);
```

```
    .
    .
    .
};

// Class constructors of maximize and minimize buttons.
UIW_MAXIMIZE_BUTTON::UIW_MAXIMIZE_BUTTON(void) :
    UIW_BUTTON(0, 0, 0, NULL, BTF_SEND_MESSAGE | BTF_NO_TOGGLE,
    WOF_BORDER | WOF_JUSTIFY_CENTER | WOF_SUPPORT_OBJECT |
    WOF_NON_FIELD_REGION, NULL, L_MAXIMIZE)
{
    // Initialize the maximize button information.
    UIW_MAXIMIZE_BUTTON::Information(INITIALIZE_CLASS, NULL);
}

UIW_MINIMIZE_BUTTON::UIW_MINIMIZE_BUTTON(void) :
    UIW_BUTTON(0, 0, 0, NULL, BTF_SEND_MESSAGE | BTF_NO_TOGGLE,
    WOF_BORDER | WOF_JUSTIFY_CENTER | WOF_SUPPORT_OBJECT |
    WOF_NON_FIELD_REGION, NULL, L_MINIMIZE)
{
    // Initialize the minimize button information.
    UIW_MINIMIZE_BUTTON::Information(INITIALIZE_CLASS, NULL);
}
```

The following chart shows the complete Zinc Application Framework class hierarchy. (Classes that are noted with † have multiple inheritance.)

```
class UI_APPLICATION

class UI_DISPLAY
    ├── †class UI_BGI_DISPLAY
    ├── †class UI_FG_DISPLAY
    ├── †class UI_GRAPHICS_DISPLAY
    ├── class UI_MOTIF_DISPLAY
    ├── †class UI_MSC_DISPLAY
    ├── class UI_OS2_DISPLAY
    ├── class UI_MSWINDOWS_DISPLAY
    └── †class UI_TEXT_DISPLAY

class UI_ELEMENT
    ├── class UI_DEVICE
    │       ├── class UID_CURSOR
    │       ├── class UID_KEYBOARD
    │       ├── class UID_MOUSE
    │       └── class UID_PENDOS
    ├── 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



† - 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_GRAPHICS_DISPLAY, UI_MSC_DISPLAY, UI_-TEXT_DISPLAY, UIW_WINDOW, etc. 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_DISPLAY, UI_REGION_LIST
class EXPORT UI_FG_DISPLAY : public UI_DISPLAY, UI_REGION_LIST
class EXPORT UI_GRAPHICS_DISPLAY : public UI_DISPLAY, UI_REGION_LIST
class EXPORT UI_MSC_DISPLAY : public UI_DISPLAY, UI_REGION_LIST
class EXPORT UI_TEXT_DISPLAY : public UI_DISPLAY, UI_REGION_LIST
class EXPORT UIW_WINDOW : public UI_WINDOW_OBJECT, public UI_LIST
```

Most of the display classes are derived from UI_DISPLAY and UI_REGION_LIST so that the display can manage any window objects attached to the Window Manager. The displays that are not derived from UI_REGION_LIST are those displays where the host environment automatically handles the region clipping.

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).

Abstract classes

Zinc Application Framework has one class that is considered "abstract": UI_DEVICE. 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). A pure virtual function cannot be called directly; rather it forces the class to become a "template" for defining other derived classes. 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;
};
```

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 Application Framework 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 it 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).

UIW_STRING makes **UIW_TEXT** a friend class since the text field uses UIW_STRING objects to display the "lines" of text.

UIW_VT_LIST makes **UIW_COMBO_BOX** a friend class since the pull-down list associated with the combo box is implemented with a UIW_VT_LIST.

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 Application Framework's display, 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, found in the *Programming Techniques* manual, 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. Functions can be written to implicitly call the class constructors:

```
#include <ui_win.hpp>

ExampleFunction()
{
    // Create a window.
    UIW_WINDOW window(0, 0, 25, 5);
    .
    .
    .
}
```

In this case, the object is constructed when the scope of its class is reached. (**NOTE:** In this example, the window created will be automatically deleted when the scope of the function ends.)

Base class construction

In addition to the class constructors you use, Zinc Application Framework 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 constructor before it initializes any of its own information:

```
UI_TEXT_DISPLAY::UI_TEXT_DISPLAY(TDM_MODE mode) :
    UI_DISPLAY(TRUE), UI_REGION_LIST()
{
    .
    .
    .
}
```

NOTE: In C++, a base class, with no arguments, is automatically initialized whether or not it is called from the derived constructor. Although this is legal, these types of base classes are nonetheless explicitly called throughout Zinc Application Framework in order to make the code more readable. Notice that UI_REGION_LIST was called from 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)
{
    .
    .
    .
}
```

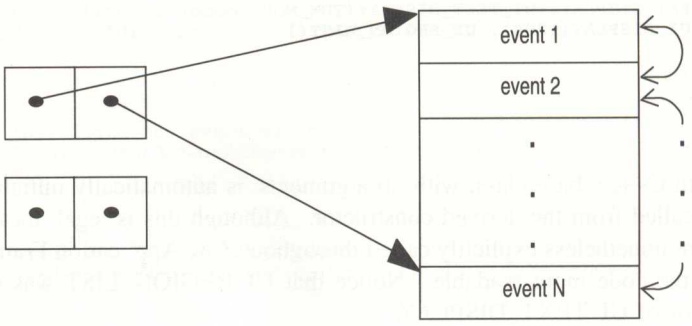
The UIW_POP_UP_ITEM class calls the UIW_BUTTON class for initialization, which in turn calls UI_WINDOW_OBJECT for base class initialization. This saves a tremendous amount of code that would be required to initialize each object separately:

```
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 the library to only perform one big allocation for event information. This prevents the library 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 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 a 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);

    // Persistent object constructor.
    UIW_POP_UP_ITEM(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

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 last is used to construct the pop-up item from disk information.

Copy constructors

Three library classes use copy constructors: UI_BIGNUM, 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 Application Framework when there is a default mode of operation that is only occasionally overridden. The use of default arguments allows you to leave off the ending argument definitions that you either do not need to worry about, or that you can use only when 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_REGION_LIST
{
public:
    UI_TEXT_DISPLAY(TDM_MODE 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, six 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 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 else it is 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 Application Framework 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, the class destructor is automatically called when the scope of **ExampleFunction** ends.

```
ExampleFunction()
{
    UIW_WINDOW window(0, 0, 25, 5);
    .
    .
    .
    // The window is automatically destroyed when the scope of
    // ExampleFunction 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 Application Framework 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 allows the library to call the destructor of the base class, rather than the specific derived object. For example, Zinc Application Framework 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. 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 Application Framework classes explicitly call destructors for derived objects and for objects that have been attached (e.g., objects attached to a UI_LIST). 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 order opposite of class construction.

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 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 typedef 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_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 function 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 a 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 declared for them outside of the class definition.

Member Functions

Function definitions

Zinc Application Framework 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 the member functions **ListIndex( )**, **Next( )** and **Previous( )**:

```
class EXPORT UI_ELEMENT
{
public:
    int ListIndex(void);
    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 ~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,
        HBITMAP *colorBitmap = NULL, HBITMAP *monoBitmap = NULL);
    virtual void BitmapArrayToHandle(SCREENID screenID, int bitmapWidth,
        int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette, HBITMAP *colorBitmap,
        HBITMAP *monoBitmap);
    virtual void BitmapHandleToArray(SCREENID screenID, HBITMAP colorBitmap,
        HBITMAP monoBitmap, int *bitmapWidth, int *bitmapHeight,
        UCHAR **bitmapArray);
    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);
    virtual void IconArrayToHandle(SCREENID screenID, int iconWidth,
        int iconHeight, const UCHAR *iconArray, const UI_PALETTE *palette,
        HICON *icon);
    virtual void IconHandleToArray(SCREENID screenID, HICON icon,
        int *iconWidth, int *iconHeight, UCHAR **iconArray);
    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);
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground);
    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);
    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);
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion);
    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);
    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,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextWidth(const char *string, SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    int VirtualGet(SCREENID screenID, const UI_REGION &region);
    virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
        int bottom);
    virtual int VirtualPut(SCREENID screenID);
    .
    .
    .
};
```

Virtual member functions

Virtual member functions are used to ensure that the most-derived 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))
        {
#if defined(ZIL_MSWINDOWS)
            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);
            }
#elif defined(ZIL_OS2)
            QMSG message;
            if ((!FlagSet(flags, Q_NO_BLOCK) && !queueBlock.First()) ||
                WinPeekMsg(display->hab, &message, 0, 0, 0, PM_NOREMOVE))
            {
                WinGetMsg(display->hab, &message, 0, 0, 0);
                UI_EVENT event(E_OS2, message.hwnd, message.msg,
                    message.mp1, message.mp2);
                event.message = message;
                Put(event, Q_BEGIN);
            }
#elif defined(ZIL_MOTIF)
            if (XtAppPending(display->appContext))
            {
                MSG message;
                XtAppNextEvent(display->appContext, &message);
                UI_EVENT event(E_MOTIF, message);
                Put(event, Q_BEGIN);
            }
#endif
            for (device = First(); device; device = device->Next())
                device->Poll();
        }
        .
        .
        .
}
```

The UI_WINDOW_OBJECT class defines virtual **Event( )**, **Information( )**, **Load( )** 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);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL);
};
```

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 Application Framework, 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)
    {
    .
    .
    .
    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( )**.

```
// ---- UI_DATE ----------------------------------------
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 **DataGet( )** functions allow you to get:

- a date based on three integers: year, month and day
- a date based on an alphanumeric value

The various overloaded **DataSet( )** functions allow you to set:

- a system date (this method requires no arguments)
- a date based on a date class object previously constructed
- a date based on three integers: the year, month and day
- a 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 functions are used throughout the library. Zinc Application Framework 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:
        {
        UI_TIME currentTime;
        if (FlagSet(btFlags, BTF_DOUBLE_CLICK) && ccode == L_END_SELECT &&
            userFunction && !parent->Inherited(ID_LIST) &&
            currentTime - lastTime < doubleClickRate)
        {
            UI_EVENT uEvent = event;
            ccode = (*userFunction)(this, uEvent, L_DOUBLE_CLICK);
        }
    }
}
```

Operator overloads

Operator overloads are used by Zinc Application Framework 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 !=.

```
// ---- UI_DATE ----------------------------------------
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-(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 ----------------------------------------------
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 Application Framework 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 object.

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 button, minimize button, system button and title:

```
UIW_WINDOW *UIW_WINDOW::Generic(int left, int top, int width,
    int height, char *title, UI_WINDOW_OBJECT *minObject,
    WOF_FLAGS woFlags, WOAF_FLAGS woAdvancedFlags,
    UI_HELP_CONTEXT helpContext)
{
    // Create the window and add default window objects.
    UIW_WINDOW *window = new UIW_WINDOW(left, top, width, height,
        woFlags, woAdvancedFlags, helpContext, minObject);
    (void)&(*window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON(SYF_GENERIC)
        + new UIW_TITLE(title));
```

```
    // Return a pointer to the new window.
    return (window);
}
```

4—Static members are used to send system messages to the Event Manager. For example, when the end user presses <ENTER> or clicks the mouse button on a UIW_BUTTON object whose BTF_SEND_MESSAGE flag is set, the button sends a message (whose type is *UIW_BUTTON::value*) to the Event Manager. The following code shows how this is accomplished:

```
EVENT_TYPE UIW_BUTTON::Message(UI_WINDOW_OBJECT *object,
    UI_EVENT &event, EVENT_TYPE ccode)
{
    // Check for valid button processing.
    if (ccode != L_SELECT || FlagSet(object->woStatus, WOS_EDIT_MODE))
        return (ccode);

    // Process the button value as a system message.
    EVENT_TYPE command;
    object->Information(GET_VALUE, &command);
    event.type = command;
    event.rawCode = 0;
    event.data = object;
    if (command == L_RESTORE || command == L_MOVE || command == L_SIZE
        || command == L_MINIMIZE || command == L_MAXIMIZE ||
        command == S_CLOSE)
    {
        for (UI_WINDOW_OBJECT *tObject = object; tObject;
            tObject = tObject->parent)
            if (tObject->Inherited(ID_WINDOW) &&
                !tObject->Inherited(ID_MENU))
            {
                event.data = tObject;
                break;
            }
    }
    object->eventManager->Put(event);
    return (ccode);
}
```

5—Static member functions are used in conjunction with all window objects when the persistent object constructor is called. Each window object loaded from a **.DAT** file (i.e., created by Zinc Designer) has a static member function called **New( )**. When a call is made to an object's constructor, all code related to the class is linked into the executable program. However, if the call is made in a static function and the static function is never called, the linker can link out the object's code. Since Zinc creates an object table when persistent objects are used, the **New( )** function proves very useful. Below is an example of an object's **New( )**:

```
static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
    UI_STORAGE_OBJECT *object)
    { return (new UIW_BUTTON(name, file, object)); }
```

NOTE: 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

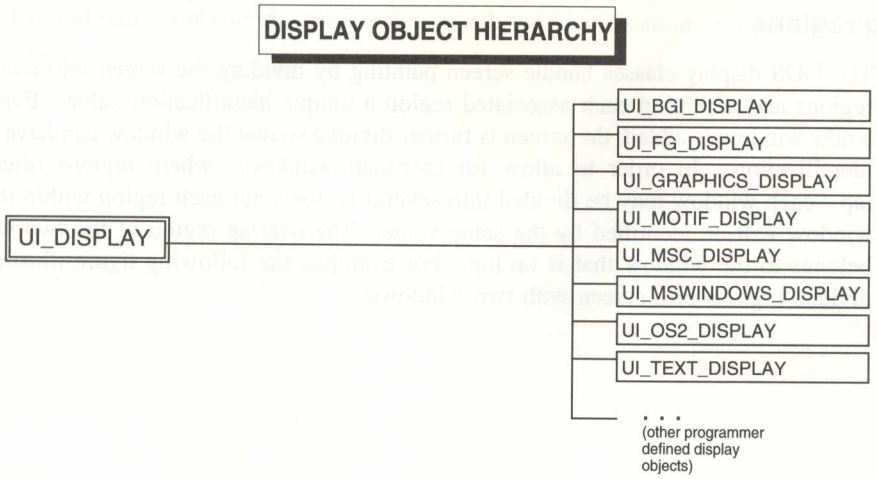
This concludes the discussion of Zinc Application Framework and its implementation of C++ features. See the *Programming Techniques* manual for tutorials that are designed to help you with specific implementation or design skills that will help you write more effective applications using Zinc Application Framework.

CHAPTER 12 – SCREEN DISPLAY

Introduction

The screen display is a support resource to Zinc Application Framework. It provides low-level screen support in both graphics and text modes.

The base display class is UI_DISPLAY, which defines the general features needed for screen output. It is designed as a generic base class so that DOS Text, DOS Graphics, Windows, OS/2 and Motif modes can be supported without limiting the unique functionality of any one mode. From UI_DISPLAY eight more specific classes are derived: UI_BGI_DISPLAY, which controls Borland graphics mode; UI_FG_DISPLAY, which controls Zortech graphics mode; UI_GRAPHICS_DISPLAY, which controls the Zinc graphics mode; UI_MOTIF_DISPLAY, which operates under Motif; UI_MSC_-DISPLAY, which controls the Microsoft graphics mode; UI_OS2_DISPLAY, which operates under OS/2; 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

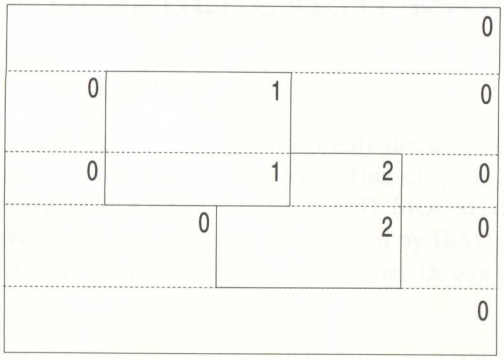
The upper left corner of the screen display is position (0,0). The x-coordinates increase from the left of the screen to the right, while the y-coordinates increase from the top of

the screen towards the bottom. The following list shows the coordinates for the bottom right corner of the modes supported by Zinc Application Framework:

- 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 DOS display classes handle 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 13 – DEFAULT EVENT MAPPING

Overview

"Chapter 3—Conceptual Design" of this manual briefly discussed the implementation of event mapping in Zinc Application Framework. 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 Application Framework 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** (a static table accessed by UI_WINDOW_OBJECT::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,             E_KEY,      TAB },
    { ID_WINDOW_OBJECT,     L_PREVIOUS,         E_KEY,      BACKTAB },
    { ID_WINDOW_OBJECT,     L_SELECT,           E_KEY,      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 Application Framework'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. A string 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 Window Manager, the mapping algorithm walks through the event map table and searches for the best match according to the object's identification, the device's identification, the raw scan code and the input modifier (e.g., keyboard shift state) 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+Ins>**	Copies the marked portion 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	**<Shift+Del>**	Cuts the marked portion 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	**<Del>**	Deletes the marked text from the current window field. The cut section is not stored in the global paste buffer. This key only has effect in fields that can be edited.
Delete next character	**<Del>** **<Gray+Delete>**	Deletes the character beneath 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 to the left of 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. (**NOTE:** All temporary windows will be deleted first.)
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 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 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. (**NOTE:** The <Ctrl+Break> and <Ctrl+C> keystrokes can be modified by changing UID_KEYBOARD::breakHandlerSet)
Help—context sensitive	**<F1>**	Displays context sensitive help information regarding the current window object.
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 not positioned in the *first* character position of a field, 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.

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 <Ctrl+Ins>, the marked text is copied. When followed by any movement keys and then <Shift+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 maximized 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 minimized 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	**<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.
Next MDI window	**<Ctrl+F6>**	Moves from the current (or selected) MDI child window to the *next* selectable MDI child window within the parent window's list of windows.
Paste	**<Ctrl+F8>** **<Shift+Ins>**	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	**<→>** **<Gray →>**	If the cursor is not positioned in the *last* character position of a left-hand field, 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.
Size window	**<Alt+F8>**	Sizes, relative to the top left 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.
System (MDI)	**<Ctrl+Spacebar>**	Selects the system button (if any) associated with the current MDI child window. This causes the pop-up menu associated with the current MDI child window's system button to be displayed on the screen.
Toggle	**<Ins>** **<Gray Insert>**	Toggles the edit mode from *insert* to *overstrike* mode or vice-versa. This key only has effect in fields that can be edited.
Up	**<↑>** **<Gray ↑>**	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 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 a 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