Message passing. . .

Zinc® Application Framework™

Programming Techniques

Version 3.5

Zinc Software Incorporated
Pleasant Grove, Utah

TABLE OF CONTENTS

INTRODUCTION

The purpose of this manual is to help you get started using Zinc Application Framework and to teach you the theories used in the design and implementation of the library. Although most of the concepts and programming styles presented in this book can be understood by beginning C++ programmers, if you have problems we recommend you use an accompanying book on C++ as a cross-reference. Some books that introduce the C++ programming language are:

- *Borland C++, Programmer's Guide*. Scotts Valley, CA: Borland International, 1992, 444 pages.
- Ellis, Margaret A. and Bjarne Stroustrup. *Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990, 447 pages.
- Dewhurst, Stephen C. and Kathy T. Stark. *Programming in C++*. Englewood Cliffs, New Jersey: Prentice Hall, 1989, 233 pages.
- Eckel, Bruce. *Using C++*. Berkeley, CA: Osborne/McGraw-Hill, 1990, 617 pages.
- Gorlen, Keith; Stanford Orlow and Perry Plexico. *Data Abstraction and Object-Oriented Programming in C++*. New York, NY: John Wiley & Sons, 1990, 403 pages.
- Hansen, Tony L., *The C++ Answer Book*. Reading, MA: Addison-Westley, 1990, 578 pages.
- Laurel, Brenda, ed. *The Art of Human-Computer Interface Design*. Reading, MA: Addison-Wesley, 1990. (50 essays related to effective user-interface design)
- Lippman, Stanley B. *C++ Primer*. Reading, MA: Addison-Westley, 1989, 464 pages.
- *Microsoft C/C++, C++ Language Reference*. Redmond, WA: Microsoft Corporation, 1992, 452 pages.
- Petzold, Charles. *Programming Windows*. Redmond, WA: Microsoft Press, 1990, 944 pages.
- Pohl, Ira. *C++ for C Programmers*. Redwood City, CA: Benjamin/Cummings Publishing, 1989, 244 pages.

- Stevens, Al. *Teach Yourself C++*. Portland, OR: MIS Press, 1990, 272 pages.
- Stroustrup, Bjarne. *The C++ Programming Language*. Reading, MA: Addison-Westley, 1986, 328 pages.
- Voss, Greg and Paul Chui. *Turbo C++ DiskTutor*. Berkeley, CA: Osborne/McGraw-Hill, 1990.
- Wiener, Richard S. and Lewis J. Pinson. *An Introduction to Object Oriented Programming and C++*. Reading, MA: Addison-Westley, 1989, 273 pages.
- *Zortech C++, Compiler Reference*. Arlington, MA: Symantec Incorporated, 1991, 483 pages.

In addition, you should have the *Programmer's Reference* available, as many of the tutorials refer to constructors, member variables and member functions that are described in detail in the reference manual.

Every section is designed to stand on its own and to teach a particular set of design and implementation issues. In addition, the tutorials in each section range from beginning to advanced. Here is a brief introduction of the topics covered in this manual:

Section I—Hello World! tells you how to initialize the main components of Zinc Application Framework. Concepts covered in this section include:

- initializing the screen display (first tutorial).
- creating input devices, such as the keyboard and a mouse, along with their controlling object, the Event Manager (first tutorial).
- constructing windows with sub-objects and then attaching them to their controlling object, the Window Manager (first tutorial).
- using the help and error systems (second tutorial).
- using the UI_APPLICATION class to initialize your program.
- using persistent window objects created with the interactive design tool (third tutorial).

NOTE: We recommend that you read this section first so that you understand the Zinc initialization process used by all subsequent tutorials in this manual.

Section II—Dictionary describes the transition from C to C++, building Zinc Application Framework applications, using Zinc Designer, and using the Zinc data file for load/store operations.

Section III—Zinc Application Program describes the overall design and implementation issues you should be concerned about when creating applications using C++ and Zinc Application Framework. This set of tutorials examines an application program to show how Zinc Application Framework's event driven, object-oriented architecture can be used to create effective and easy-to-use applications in a fraction of the time needed to create applications without Zinc.

Section IV—Derived Classes contains a set of tutorials that show how Zinc Application Framework classes can be modified to perform customized operations. The following tutorials are contained in this section:

Macro device—This tutorial shows how to derive a macro device from the UI_-DEVICE base class. The macro input device looks for certain keyboard characters (F5 through F8) and then converts the special keys to macro operations.

Help bar—This tutorial shows you how to create a help bar class from the UI_-WINDOW_OBJECT base class and how to integrate it into an application.

Virtual list—This tutorial shows you how to create a low-level virtual list class, then how to derive a presentation virtual list class from the UIW_WINDOW base class. This class is useful when you want to present a lot of list information that is contained on disk.

Customized display—This tutorial explains how screen display classes are implemented from the UI_DISPLAY base class.

Section V—Portability Issues contains a set of tutorials that illustrate how Zinc Application Framework can be used to handle internationalization of languages and currencies. The following tutorials are contained in this section:

Multi-Language Programs—This tutorial shows how Zinc can detect the system's default country settings and then access a **.DAT** file to retrieve a particular window based on the current country settings.

International Currency—This tutorial shows how multiple currency formats can be displayed simultaneously and how exchange rate information can be used when changing country codes.

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). These tutorials begin by comparing the basic storage techniques employed by both C and C++. The tutorial concludes with a discussion of the implementation techniques used to store Zinc window objects.

Section VII—Zinc Designer contains a set of reference chapters describing the functions associated with Zinc Designer and each of its components.

File Options—This chapter outlines the general operations of Zinc Designer. File operations and presentation preferences are illustrated.

Edit Options—This chapter outlines how to change the appearance and performance of objects within the current file.

Resource Options—This chapter outlines how to create, modify, retrieve and test resources in the current file.

Object Options—This chapter displays the options that allow you to actually create objects.

Input Options—This chapter discusses how to create input objects such as: string, formatted string, text, date, time, bignum, integer and real.

Control Options—This chapter discusses how to create control objects such as: button, radio button, check box, vertical list, horizontal list, combo box, vertical scroll bar, horizontal scroll bar and child window.

Menu Options—This chapter discusses how to create menu objects such as pull-down menu and tool bar.

Static Options—This chapter discusses how to create static objects such as: prompt, group box and icon.

Utilities Options—This chapter outlines the interaction of the image and help editors.

Help Options—This chapter outlines the on-line help available within Zinc Designer.

Section VIII—Appendices addresses other topics that may be useful when developing applications. The following information is contained in appendix chapters:

Compiler Considerations—This appendix discusses the compiler dependencies that you need to consider when using Zinc Application Framework.

Example Programs—This appendix briefly describes the Zinc support programs installed in **\ZINC\EXAMPLE**. These example programs are designed to help you with specific implementation issues of using Zinc Application Framework.

Zinc Coding Standards—This appendix gives you the coding standards Zinc Software employees use when coding the library, example and tutorial source code modules. This appendix is included to help you get "up-to-speed" with the coding style you see throughout the tutorial programs, example programs and sample code contained in the Zinc Application Framework manuals.

Common Questions and Answers—This appendix contains a set of commonly asked technical support questions about Zinc Application Framework.

If you need assistance after studying the tutorial programs and example programs, or have questions in general, please contact our technical support group at (801) 785-8998 between the hours of 8:00 a.m. and 5:00 p.m. Mountain Standard Time. 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.

In addition, our bulletin board system is updated regularly with the latest maintenance release of the software, example programs and ideas concerning Zinc Application Framework. This service is available 24 hours a day by calling (801) 785-8997 with 300-9600 baud (V.32 *bis*), 8 data bits, no parity and 1 stop bit or by calling (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.

No set of tutorials can address all of the questions that you could have concerning the design and implementation of applications. We are confident, however, that you will find the tutorial programs, example programs, technical support and bulletin board service invaluable in your effort to learn C++ and Zinc Application Framework.

NOTE: All of the figures in these tutorials were taken from the Microsoft Windows environment. The actual presentation of a particular window may vary depending on the environment and the type of display used.

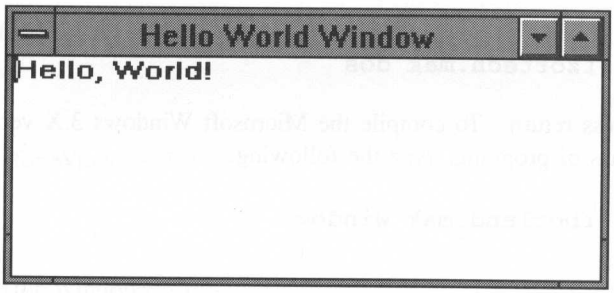
SECTION I
HELLO WORLD!

CHAPTER 1 – INITIALIZING THE LIBRARY

The first tutorial program in this section shows you how to set up the basic Zinc Application Framework elements. The basis for this tutorial comes from the classic "Hello, world" example given in several programming language books. For example, page 12 of *The C++ Programming Language* (Stroustrup, Bjarne. Reading, MA: Addison-Westley, 1986) implements the following C++ code for the "Hello, world" program:

```
#include <iostream.h>
main()
{
    cout << "Hello, world\n";
}
```

The above program prints the text "Hello, world" to the screen. The program to be presented in this chapter plays on this theme by displaying the text "Hello, World!" in a window. The figure below shows how the window will look once the program is complete.



The code for this tutorial is located in **\ZINC\TUTOR\HELLO\HELLO1.CPP**. The major steps involved in the creation of this program are:

- Declaration of include files
- Definition of the screen display (text, graphics or host environment)
- Definition of the Event Manager with input devices (mouse, keyboard)
- Definition of the Window Manager
- Creation of the simple "Hello World!" window
- The main program loop that coordinates input information

- Cleanup

If you are not familiar with the process involved in compiling source code modules, you should take a few minutes to read "Appendix A—Compiler Considerations." Most of the programs in these tutorials have been created to run in all environments (i.e., DOS Text, DOS Graphics, Microsoft Windows 3.X, Windows NT, IBM OS/2 and OSF/Motif).

Running the program

Before the Hello1 program can be run, it must be compiled. To compile the DOS version of the "Hello World!" class of programs, type the following:

```
make -fborland.mak dos
```

or

```
nmake -fmicrosft.mak dos
```

or

```
make -fzortech.mak dos
```

and then press return. To compile the Microsoft Windows 3.X versions of the "Hello World!" class of programs, type the following:

```
make -fborland.mak windows
```

or

```
nmake -fmicrosft.mak windows
```

or

```
make -fzortech.mak windows
```

and then press return. To compile the Windows NT version of the "Hello World!" class of programs, type the following:

```
nmake -f mscwnt.mak winnt
```

and then press return. To compile the IBM OS/2 version of the "Hello World!" class of programs, type the following:

```
make -fborland.mak os2
```

or

```
make -fzortech.mak os2
```

and then press return. To compile the Motif version of the "Hello World!" class of programs, type the following:

```
make -f motif.mak motif
```

or

```
make
```

and then press return.

The naming convention for the Hello1 executable program (as well as all other Zinc-provided utilities, tutorials and examples) is as follows:

hello1 (for DOS)

or

whello1 (for Windows 3.X)

or

whello1 (for Windows NT)

or

ohello1 (for OS/2)

or

hello1 (for Motif).

To exit any version of the Hello1 program, double-click on the system button with the mouse or press <Alt+F4> with the keyboard.

Include files

The first step in writing the "Hello World!" program is declaring the proper include file. Zinc Application Framework allows access to the following include files:

UI_ENV.HPP—Contains compiler and environment-specific defined values and information.

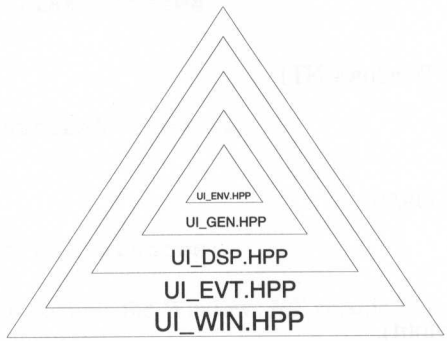
UI_GEN.HPP—Contains the definitions of low-level classes used throughout the library, including UI_ELEMENT and UI_LIST.

UI_DSP.HPP—Contains the definition of all display related class information.

UI_EVT.HPP—Contains information used by the Event Manager and window objects when they communicate to, or receive information from, the Event Manager.

UI_WIN.HPP—Contains the class definitions for the Window Manager, as well as for all windows and window objects.

These files do not require nor contain any compiler specific include files. However, environment specific include files are included in **UI_ENV.HPP** (e.g., **WINDOWS.H** is included for Microsoft Windows, **OS2.H** is included for OS/2). This makes it possible to create applications without having to determine whether or not any compiler include files have already been incorporated. The hierarchy observed by Zinc Application Framework include files is represented in the figure below:



Since the "Hello World!" program creates a window to display its text, the **UI_WIN.-HPP** include file is needed. Accordingly, it is declared at the top of **HELLO1.CPP**:

```
#include <ui_win.hpp>
```

NOTE: Because of the include file hierarchy, including the **UI_WIN.HPP** file also causes the **UI_EVT.HPP**, **UI_DSP.HPP**, **UI_GEN.HPP** and **UI_ENV.HPP** files to be included.

The screen display

The next step in writing the "Hello World!" program requires that you set up the screen. This is accomplished through the following code:

```
#if defined(ZIL_MSDOS)
main()
{
    // Create the MSDOS display.
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
    if (!display->installed)
    {
        delete display;
        display = new UI_TEXT_DISPLAY;
    }
#elif defined(ZIL_MSWINDOWS)
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR,
    int nCmdShow)
{
    // Create the Windows display.
    UI_DISPLAY *display = new UI_MSWINDOWS_DISPLAY(hInstance, hPrevInstance,
        nCmdShow);
#elif defined(ZIL_OS2)
main()
{
    // Create the OS/2 display.
    UI_DISPLAY *display = new UI_OS2_DISPLAY;
#elif defined(ZIL_MOTIF)
main(int argc, char **argv)
{
    // Create the Motif display.
    UI_DISPLAY *display = new UI_MOTIF_DISPLAY(&argc, argv, "ZincApp");
#endif
```

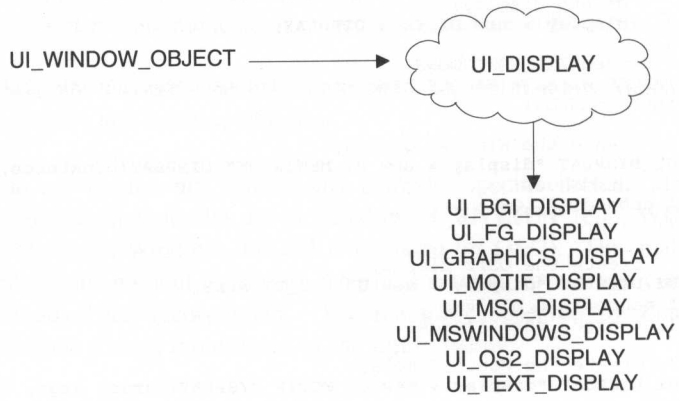
The UI_DISPLAY class is used by all Zinc Application Framework classes that present information to the screen, whether in text or graphics modes of operation. For example, the DOS version of the "Hello World!" program ensures that the highest resolution display is used by first trying to create a graphics display. If no graphics display is available, it then creates a text display. A forced 25x80 text display could have been created by replacing the DOS code above with:

```
// Initialize the display.
UI_DISPLAY *display = new UI_TEXT_DISPLAY(TDM_25x80);
```

The Windows display is initialized by three parameters: *hInstance*, *hPrevInstance* and *nCmdShow*. These parameters are passed into the program by the windows system and need only be passed directly on to the UI_MSWINDOWS_DISPLAY constructor. This will be presented again later in this tutorial.

The Motif display is initialized by three parameters: *argc*, *argv* and *"ZincApp"*. The first two are the standard command-line parameters passed to any C/C++ program. The third parameter is the name of the class of application being created. These arguments are passed to the display class so that they may be passed to the Xt Intrinsic initialization routines, thus allowing Zinc applications to take advantage of all the regular X command-line options (e.g., using other displays, colors, fonts, etc.).

You may have noticed that the *display* variable is declared as UI_DISPLAY and not as UI_MSC_DISPLAY, or any other type of display that is actually constructed. The UI_DISPLAY class is a generic base class from which all Zinc Application Framework text and graphics displays are derived. Thus, when a window is shown on the screen, it uses UI_DISPLAY member functions to draw screen information. This concept is illustrated below:



The event manager

After the display class has been created, the Event Manager and input devices must be created. This is accomplished with the following code:

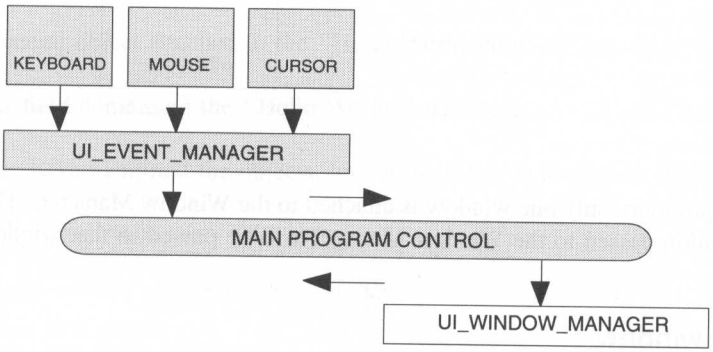
```
// Initialize the event manager and add three devices to it.
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
*eventManager
    + new UID_KEYBOARD
    + new UID_MOUSE
    + new UID_CURSOR;
```

The Event Manager is constructed in the first line. It requires one parameter:

- *display* is used by the input devices to display information on the screen. For example, the UID_CURSOR device uses the display argument to paint a blinking cursor on the screen (in graphics mode).

After the Event Manager is created, three input devices (i.e., keyboard, mouse, cursor) are attached to it using the overloaded operator **UI_EVENT_MANAGER::operator +**.

"Chapter 3—Conceptual Design" in the *Programmer's Guide* discusses the interaction between input devices and the Event Manager within Zinc Application Framework. The figure below reviews this interaction:



The window manager

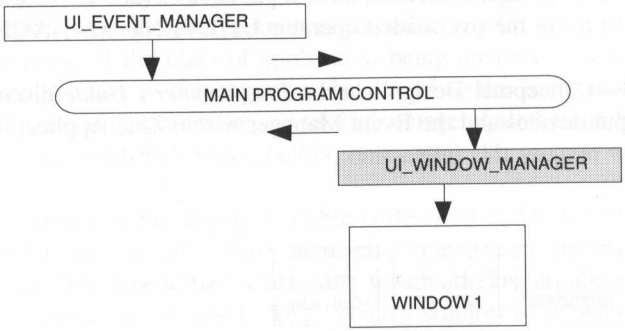
The final basic component of Zinc Application Framework is the Window Manager, which is created with the following code:

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

The Window Manager is constructed with two parameters:

- *display* is used to send window information to the screen (such as commands to draw lines, fill regions or display text on the screen).
- *eventManager* is used to send system and input information through Zinc Application Framework.

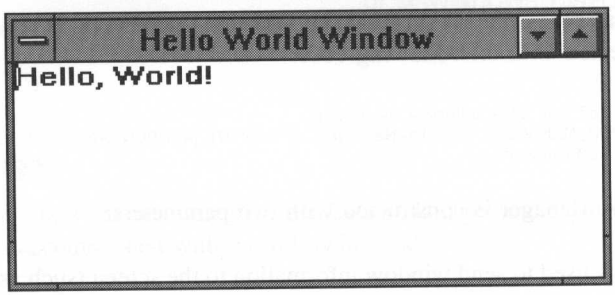
"Chapter 3—Conceptual Design" in the *Programmer's Guide* also briefly discusses the interaction of the Window Manager within Zinc Application Framework. The figure below reviews this interaction:



In this program, only one window is attached to the Window Manager. Thus, all relevant information passed to the Window Manager will be passed to that window.

A simple window

You are now ready to create the "Hello World!" window and to attach it to the screen. Let's examine the original picture of the "Hello World!" window to identify the major window objects (also known as support objects) that need to be created:



These window objects are:

- the **window** itself (This object is not visible, but it is used to store all the related window objects identified below.)
- the **border** (Shown as the exterior shaded region of the window.)
- the **maximize button** (Shown as a button at the right top of the window with a '▲' character.)

- the **minimize button** (Shown as a button at the right top of the window with a '▾' character.)
- the **system button** (Shown as a button on the left top side of the window with a '–' character.)
- the **title** (Shown with the "Hello World Window" text on the top center of the window.)

The only general object attached to the "Hello World Window" is:

- the **text field** containing the "Hello World!" message.

Now that we have identified the objects, let's look at the code used to create them:

```
// Create the "Hello World!" window.
UIW_WINDOW *window = new UIW_WINDOW(5, 5, 40, 6);

// Add the window objects to the window.
*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE("Hello World Window")
    + new UIW_TEXT(0, 0, 0, 0, "Hello World!", 256, WNF_NO_FLAGS,
        WOF_NON_FIELD_REGION);

// Add the window to the window manager.
*windowManager + window;
```

Notice how logical and consistent code creation is! The window is created first with the following arguments:

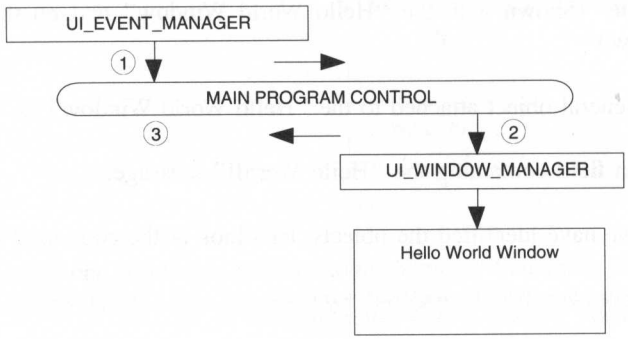
- *5* and *5* are cell coordinates that specify the left-top position of the window on the screen.
- *40* and *6* specify the width and height of the window.

The window objects are created next, using the **new** operator. Once a window object is created, it is added to the window, using the **UIW_WINDOW::operator +** operator overload. (See the *Programmer's Reference* for more information about an individual window object and the protocol used in its construction.)

Finally, the window is attached to the Window Manager, again using the + operator (overloaded by the UI_WINDOW_MANAGER class).

Program flow

In general, the conceptual flow of event driven programs is different from structured programs. The "Hello World!" program has a very simple program flow, as illustrated in the figure below:



The code implementation of this flow is shown below. (**NOTE:** The step identifiers to the right are not part of the actual code.)

```
// Wait for user response.
EVENT_TYPE ccode;
UI_EVENT event;
do
{
    // Get input from the user.
    eventManager->Get(event);                                    (1)

    // Send event information to the window manager.
    ccode = windowManager->Event(event);                         (2)
} while (ccode != L_EXIT && ccode != S_NO_OBJECT);               (3)
```

The figure and code above should help you to understand the high level operation of the program, which can be outlined as follows:

1—The user enters information by pressing a keyboard key or by pressing a mouse button.

2—The event information is passed to the Window Manager. At this point, the Window Manager sends the event information to the current window.

3—The Window Manager's return code is examined to determine whether or not to continue program execution. If execution does continue, it will return to the first step.

Cleanup

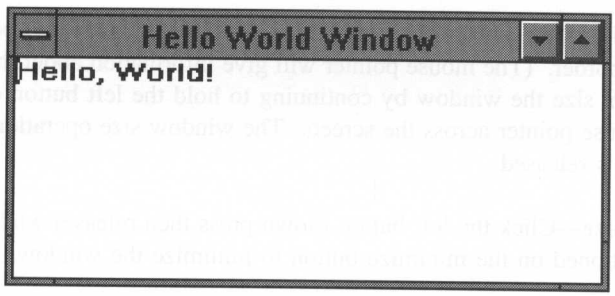
The following code is used to delete the Window Manager, Event Manager and display:

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

The order of deletion is important! The deletion of the Window Manager, Event Manager and display must be in the reverse order of their construction. Since the Window Manager maintains pointers to the Event Manager and to the display, it must be destroyed first. If the Window Manager were not deleted first, it would have valid pointers that were pointing to invalid memory (i.e., deleted objects). Also, the Event Manager must be deleted before the display since the Event Manager maintains a pointer to the display. Any objects attached to the event or window managers (e.g., UID_KEYBOARD, UID_MOUSE, the "Hello World!" window, etc.) are automatically destroyed when their respective manager is destroyed.

Run-time features

Once the application is running, you should see the following window on your display:



Some of the best features of Zinc Application Framework are inherently available to windows and the objects attached to them. For example, the following operations are available using the keyboard:

Move—Press <Alt+F7> then use the arrow keys (up, down, left, right) to change the window position. Press <Enter> to complete the move operation or <Esc> to cancel the operation.

Size—Press <Alt+F8> then use the arrow keys (up, down, left, right) to change the window size. Press <Enter> to complete the size operation or <Esc> to cancel the operation.

Minimize—Press <Alt+F9> or <Alt -> to reduce the size of the window to the minimum allowed by the UIW_WINDOW class object.

Maximize—Press <Alt+F10> or <Alt +> to increase the size of the window to occupy the whole screen. Pressing either of these keys when the window is maximized will cause the window to return to its original state.

Restore—Press <Alt+F5> to restore the window to its original size (i.e., before it was minimized or maximized). This operation only works when the window is in a maximized or minimized state.

Exit—Press <Alt+F4> to exit the program. This operation causes the window to be removed from the screen and program execution to terminate.

In addition to the keyboard operations, the same operations described above may be performed using a mouse:

Move—Press the left button down after positioning the mouse pointer over the title bar. You can move the window by continuing to hold the left button down while moving the mouse pointer across the screen. Window movement ends when the left button is released.

Size—Press the left button down after positioning the mouse pointer on some area of the border. (The mouse pointer will give information about the sizing directions.) You can size the window by continuing to hold the left button down while moving the mouse pointer across the screen. The window size operation ends when the left button is released.

Minimize—Click the left button (down press then release) while the mouse pointer is positioned on the minimize button to minimize the window.

Maximize—Click the left button while the mouse pointer is positioned on the maximize button to maximize the window. Clicking the left button on the maximize button while the window is in a maximized state will cause the window to return to its previous size.

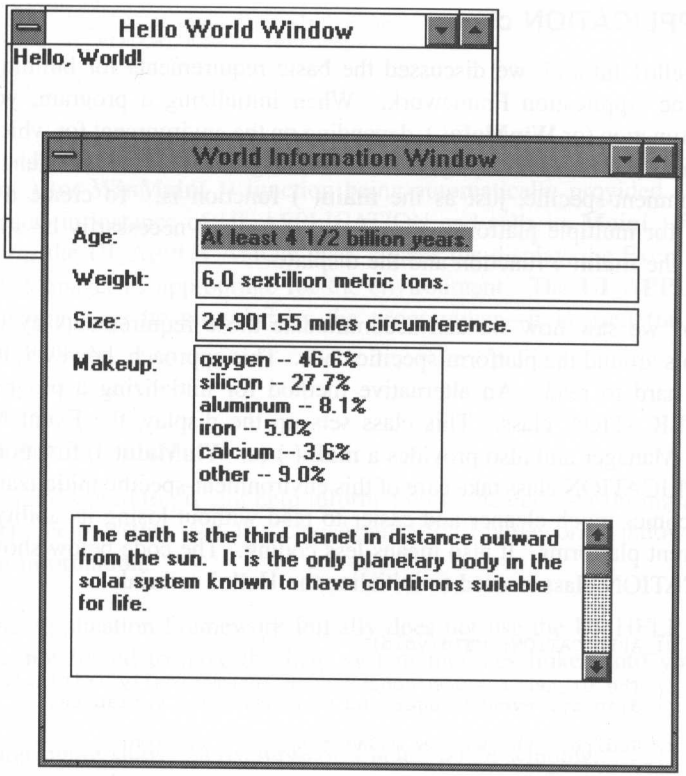
Restore—Click the left button while the mouse pointer is positioned on the maximize button (if the window is in a maximized state) or double-clicking on the minimized window (if the window is in a minimized state) to restore the original window size.

Exit—Double-click the left button while the mouse pointer is positioned on the system button to exit the program.

This concludes the first "Hello World!" tutorial. The next tutorial tells you how to add the help and error window systems in Zinc Application Framework.

CHAPTER 2 – HELP AND ERROR SYSTEMS

Congratulations on completing the first tutorial, where you learned how to set up the basic Zinc Application Framework elements. This tutorial extends the capabilities of the first "Hello World!" tutorial to add an alternate program initialization technique using the UI_APPLICATION class, windowed help and error systems, an exit function, and a "World Information" window. The final outcome should be similar to the following:



The code for this tutorial is located in **\ZINC\TUTOR\HELLO\HELLO2.CPP**.

Since this program is an extension of the original "Hello World!" program, only the new components of it will be discussed in this tutorial. These new components are:

- Use of the UI_APPLICATION class
- Creation of the help system

- Creation of the error system
- Addition of the exit function
- Addition of the "World Information" window
- Cleanup

The UI_APPLICATION class

In the Hello1 tutorial, we discussed the basic requirements for building an application using Zinc Application Framework. When initializing a program, you must write a **main( )** function (or **WinMain( )**, depending on the environment for which the application is intended) and set up a display, an Event Manager and a Window Manager. The display is environment-specific, just as the **main( )** function is. To create a program that is intended for multiple platforms means that there will necessarily be some non-portable code for the **main( )** function and the display.

In Hello1 we saw how we can accommodate those requirements by using *#if defined* statements around the platform-specific code. This approach, however, is not elegant and is often hard to read. An alternative method for initializing a program is to use the UI_APPLICATION class. This class sets up the display, the Event Manager and the Window Manager and also provides a **main( )** (or **WinMain( )**) function. By letting the UI_APPLICATION class take care of this environment-specific initialization for you, your code becomes much cleaner and easier to read without losing its ability to be compiled on different platforms. It also means less coding. The code below shows how the UI_-APPLICATION class is used to initialize the Hello2 tutorial:

```
int UI_APPLICATION::Main(void)
{
    // The UI_APPLICATION constructor automatically initializes the
    // display, eventManager and windowManager variables.

    // Redirect the window manager's exit function.
    windowManager->exitFunction = ExitFunction;

    // Initialize the help and error systems.
    UI_WINDOW_OBJECT::errorSystem = new UI_ERROR_SYSTEM;
    UI_WINDOW_OBJECT::helpSystem = new UI_HELP_SYSTEM("hello.dat",
        windowManager, HELP_GENERAL);

    // Add two windows to the window manager.
    UIW_WINDOW *window1 = HelloWorldWindow1();
    UIW_WINDOW *window2 = HelloWorldWindow2();
    (void *)&(*windowManager
        + window1
        + window2);
```

```
    // This line assigns the exit function to be called before the main
    // window is closed.  It MUST be after the window is added to
    // windowManager.
    windowManager->screenID = window1->screenID;

    // Wait for user response.
    UI_EVENT event;
    EVENT_TYPE ccode;
    do
    {
        eventManager->Get(event, Q_NORMAL);
        ccode = windowManager->Event(event);
    } while (ccode != L_EXIT && ccode != S_NO_OBJECT);

    // Clean up.
    delete UI_WINDOW_OBJECT::helpSystem;
    delete UI_WINDOW_OBJECT::errorSystem;

    return (0);
}
```

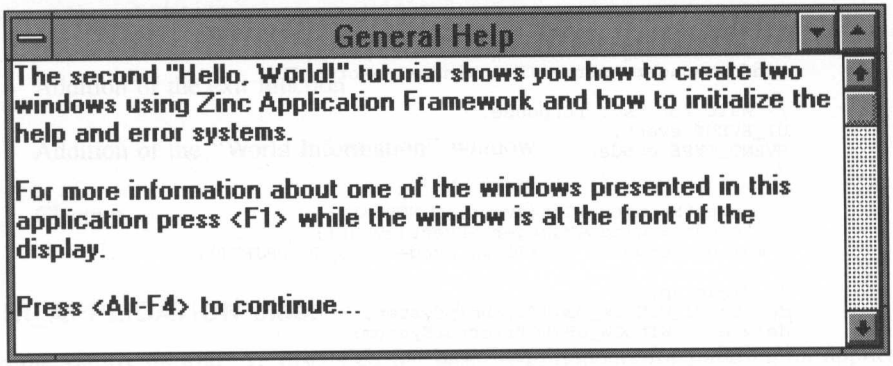
The reference to the UI_APPLICATION class causes the module to be linked in, resulting in the **main( )** (or **WinMain( )**) function being automatically provided. This **main( )** function creates in instance of UI_APPLICATION and calls its **Main( )** function. The constructor for the UI_APPLICATION class creates the display, the Event Manager and the Window Manager as appropriate for the environment. The **UI_APPLICATION::-Main( )** function must be provided by the programmer—it is used for the program-specific initialization.

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.

NOTE: 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 figure shows an example of a help system window:



The help window system is included by adding the following code to the tutorial program:

```
UI_WINDOW_OBJECT::helpSystem = new UI_HELP_SYSTEM("hello.dat",
    &windowManager, HELP_GENERAL);
```

The help window system constructor arguments are:

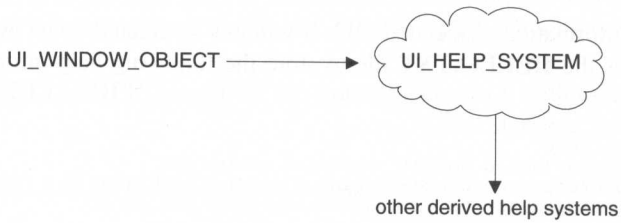
- *hello.dat* is the name of the binary help file (generated from an ASCII text file using **GENHELP.EXE** or produced from the interactive designer).

- *windowManager* is a pointer to the Window Manager. This argument is used to display information if an error is encountered while initializing the help system.

- *HELP_GENERAL* is the name of the help context that will be used if no other help context is specified when help is requested.

Notice that not only must you create a help system class object, but you must also assign it to the UI_WINDOW_OBJECT member variable *helpSystem*.

Objects make requests to the help system whenever help is requested by an end-user during an application. This interaction is represented in the figure below:



This flow of interaction can be outlined as follows:

1—The window calls the help system with a message:

```
EVENT_TYPE UI_WINDOW_OBJECT::Event(const UI_EVENT &event)
{
    .
    .
    .

    case L_HELP:
        // Display help for the current window.
        helpSystem->DisplayHelp(windowManager, helpContext);
        break;
```

The arguments used by the help system are:

- *windowManager*, which is a pointer to the Window Manager that will be used to display the help information on the screen.
- *helpContext*, which specifies the help information to be displayed.

2—The help system attaches its help information window to the screen via the Window Manager:

```
void UI_HELP_SYSTEM::DisplayHelp(UI_WINDOW_MANAGER *windowManager,
    HELP_CONTEXT helpContext)
{
    .
    .
    .

    *windowManager + helpWindow;
```

If the help window is already visible on the screen, its title and help text are updated to reflect the current help context.

3—Program flow continues as normal. The help window is now present on the screen and will receive all input messages, as long as it is the current window.

The help information associated with a window is created in an ASCII text file. This tutorial uses the **HELLO.TXT** file to store the following help information:

```
--- HELP_GENERAL
General help
The second "Hello World!" tutorial shows you
how to create two windows using Zinc Application
Framework and how to initialize the help and error
systems.

For more information about one of the windows
presented in this application press <F1> while
the window is at the front of the display.

Press <Alt+F4> to continue...
--- HELP_HELLO_WORLD
Hello World
.
.
.

--- HELP_WORLD_INFORMATION
World Information
.
.
.
```

Each help context contains the following elements:

Help context name—This name is converted to a C++ constant and specifies the help context index referenced in your code. This name must be preceded by "---", which is used as a parsing token. (The first help context name above is HELP_-GENERAL.)

Help context title—The title is used at the top of the help window as its title field. It should be a descriptive string that tells what help context is being viewed. (The first help context title above is "General Help.")

Help information—The help information is text that is displayed in the body of the help window. It should contain all the help information needed to describe the particular help being requested.

The ASCII help text file is converted using the **GENHELP.EXE** utility (located in the **\ZINC\BIN** directory). To convert the "Hello World!" help file, type:

```
genhelp hello.txt hello.dat <Enter>
```

The help generation program performs the following operations:

Creates a **HELLO.DAT** file—This file contains the help information along with help contexts. This file is stored in binary form and should not be modified by the programmer. It is the only file (in addition to the executable file) used during the application. (You do not need to ship the **.HPP** or **.TXT** file with your application.)

Creates a **HELLO.HPP** file—This file contains the C++ definitions for the help contexts.

The generated **HELLO.HPP** file is shown below:

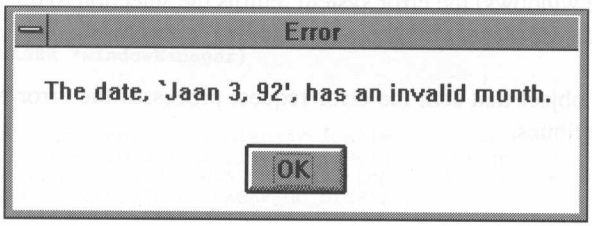
```
#ifdef USE_HELP_CONTEXTS
#define HELP_GENERAL                    1 // General help.
#define HELP_HELLO_WORLD                2 // Hello World.
#define HELP_WORLD_INFORMATION          3 // World Information.
#endif
```

You must include the application **.HPP** file in all modules that make reference to help indexes. The **HELLO2.CPP** file has the following modified include file list:

```
#include <ui_win.hpp>
#define USE_HELP_CONTEXTS
#include "hello.hpp"
```

The error system

The implementation of the error system is very similar to that of the help system in that a stub is provided as the default by Zinc Application Framework. In addition, also similar to the help system, an error window system can be defined that will override the default stub. The figure below shows an example of an error window:



The default error system is overridden by re-defining the error system variable in the following manner:

```
UI_WINDOW_OBJECT::errorSystem = new UI_ERROR_SYSTEM;
```

The flow of control with the error system is outlined as follows:

1—A window object calls the error system. In the example shown above, UIW_-DATE is the window object that calls the error system with an error message from its error message table.

```
int UIW_DATE::Validate(int processError)
{
    .
    .
    .

    for (int i = 0; errorTable[i].message; i++)
        if (errorTable[i].errorCode == errorCode)
        {
            errorSystem->ReportError(windowManager, WOS_INVALID,
                errorTable[i].message, stringDate, range);
            break;
        }
```

2—The error system attaches a modal error window to the screen display:

```
UIS_STATUS UI_ERROR_SYSTEM::ReportError(UI_WINDOW_MANAGER
    *windowManager, UIS_STATUS errorStatus, char *format, ...)
{
    .
    .
    .

    *windowManager + window;
```

Modal windows prevent the end-user from interacting with any window other than the current window—in this case the error window—until the window is closed. Since the error window is modal, it will receive all event information until it is closed.

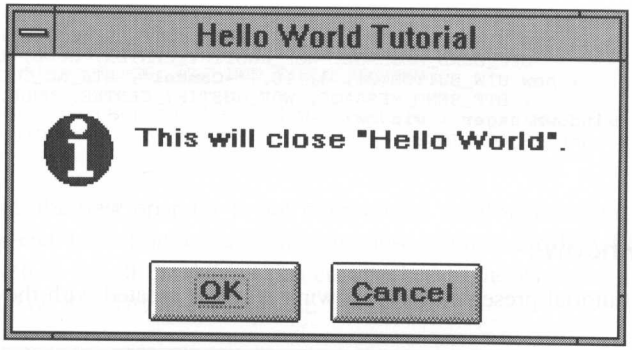
3—Once a method of correction is selected (either "OK" or "Cancel"—available on some windows) the error system returns the selection to the object where the error occurred. Consequently, the error window is removed.

4—The object that sent the error request processes the error response and program flow continues.

Exit function

When a program is about to terminate execution, it is sometimes desirable to perform special cleanup or to inform the user that the program will exit. To facilitate this, UI_-WINDOW_MANAGER has a special member variable, *exitFunction* (passed as a parameter), which is a user function that is called when the Window Manager receives an L_EXIT or L_EXIT_FUNCTION message.

The following window is displayed when <Alt+F4> is pressed:



The exit function can have any function name, but must have the following declaration:

```
static EVENT_TYPE ExitFunction(UI_DISPLAY *display,
    UI_EVENT_MANAGER *eventManager, UI_WINDOW_MANAGER *windowManager)
```

This declaration allows the exit function to have pointers to the current display, Event Manager and Window Manager. The exit function must be declared to be a static so that its address may be taken at compile time.

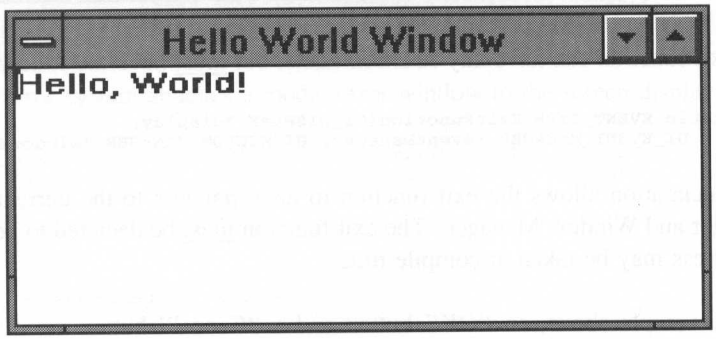
In the example above, an "OK" button and a "Cancel" button are displayed. These buttons have the BTF_SEND_MESSAGE flag set. The purpose of this flag is to create an event that has the type field set to the button's value and then to put the new event onto the event queue. When the "OK" button is pressed, an L_EXIT message is placed on the event queue and the application ends. When the "Cancel" button is pressed, the S_CLOSE message is sent and the current window (i.e., exit function window) is closed. The following code shows the implementation of this exit function:

```
static EVENT_TYPE ExitFunction(UI_DISPLAY *display, UI_EVENT_MANAGER *,
    UI_WINDOW_MANAGER *windowManager)
{
    int width = 42;
    int height = 7;
    int left = (display->columns / display->cellWidth - width) / 2;
    int top = (display->lines / display->cellHeight - height) / 2;
    UIW_WINDOW *window = new UIW_WINDOW(left, top, width, height,
        WOF_NO_FLAGS, WOAF_MODAL | WOAF_NO_SIZE);
    *window
        + new UIW_BORDER
        + &(*new UIW_SYSTEM_BUTTON
            + new UIW_POP_UP_ITEM("&Move", MNIF_MOVE)
            + new UIW_POP_UP_ITEM("&Close\tAlt+F4", MNIF_CLOSE))
        + new UIW_TITLE("Hello World Tutorial");

        if (display->isText)
            *window
                + new UIW_PROMPT(4, 1, "This will close \"Hello World\".");
        else
            *window
                + new UIW_ICON(2, 1, "ASTERISK")
                + new UIW_PROMPT(8, 1, "This will close \"Hello World\".");
```

```
    *window
        + new UIW_BUTTON(9, 4, 10, "~OK", BTF_NO_TOGGLE | BTF_AUTO_SIZE |
            BTF_SEND_MESSAGE, WOF_JUSTIFY_CENTER, NULL, L_EXIT)
        + new UIW_BUTTON(21, 4, 10, "~Cancel", BTF_NO_TOGGLE | BTF_AUTO_SIZE
            | BTF_SEND_MESSAGE, WOF_JUSTIFY_CENTER, NULL, S_CLOSE);
    *windowManager + window;
    return (S_CONTINUE);
}
```

Multiple windows

The first tutorial presented the following window created with the accompanying code:




```
// Create the hello world window.
UIW_WINDOW *window = new UIW_WINDOW(5, 5, 40, 6);

// Add the window objects to the window.
*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE("Hello World Window")
    + new UIW_TEXT(0, 0, 0, 0, "Hello World!", 256, WNF_NO_FLAGS,
        WOF_NON_FIELD_REGION);
```

To simplify the code associated with this window, we introduce the concept of "Generic" static functions. Two high level Zinc Application Framework objects have a **Generic( )** function: UIW_WINDOW and UIW_SYSTEM_BUTTON. The **UIW_WINDOW::-Generic( )** member function automatically creates a window with a border, maximize button, minimize button, system button and title. The following function shows how we can replace this code:

```
static UIW_WINDOW *HelloWorldWindow1()
{
    // Create the standard Hello World! window.
    UIW_WINDOW *window = UIW_WINDOW::Generic(2, 2, 40, 6,
        "Hello World Window", NULL, WOF_NO_FLAGS, WOAF_NO_FLAGS,
         HELP_HELLO_WORLD);
```

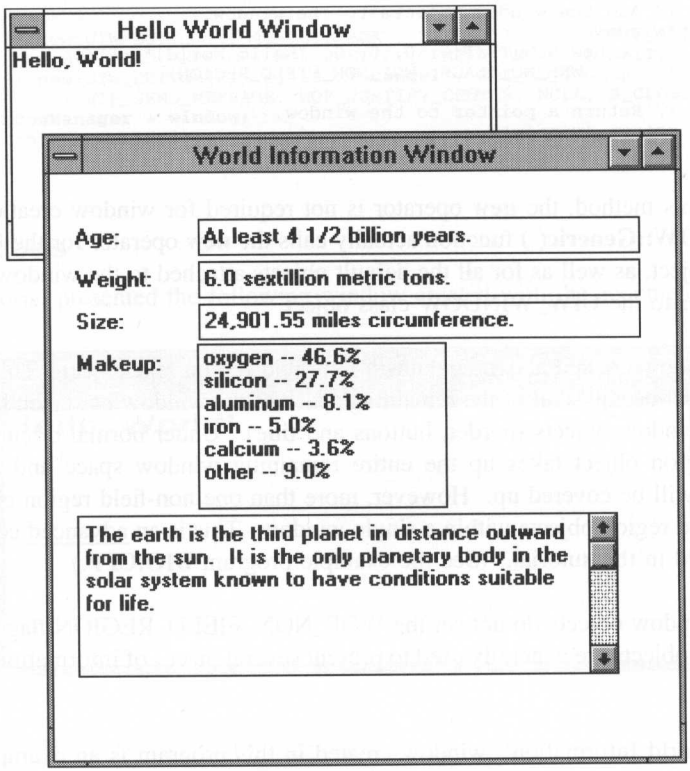
```
    // Add the window objects to the window.
    *window
        + new UIW_TEXT(0, 0, 0, 0, "Hello World!", 256,
            WNF_NO_FLAGS, WOF_NON_FIELD_REGION);

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

Using this method, the **new** operator is not required for window creation. The **UIW_WINDOW::Generic( )** function actually calls the **new** operator for the UIW_WINDOW class object, as well as for all the default objects attached to the window. It then returns a pointer to the UIW_WINDOW class object.

The window created above contains a non-field region text object. This means that the text object occupies all of the remaining space of the window not taken by the previously added window objects (border, buttons and title). Under normal circumstances, a non-field region object takes up the entire remaining window space and any field region objects will be covered up. However, more than one non-field region object may reside with field region objects within a single window. This is an advanced concept and is not addressed in this tutorial. (See the example program **BIO.CPP**.)

Field window objects do not set the WOF_NON_FIELD_REGION flag. These types of window objects are generally used to present several pieces of information in an organized manner.

The "World Information" window created in this program is an example of a window that uses field window objects to display information. This window and its code implementation is shown below:




```
static UIW_WINDOW *HelloWorldWindow2()
{
    // Create the world information window.
    UIW_WINDOW *window = UIW_WINDOW::Generic(5, 5, 52, 15,
        "World Information Window", NULL, WOF_NO_FLAGS, WOAF_NO_SIZE,
        HELP_WORLD_INFORMATION);

    // Add the window objects to the window.
    *window
        + new UIW_PROMPT(2, 1, "Age:")
        + new UIW_STRING(12, 1, 36, "At least 4 1/2 billion years.", 50)

        + new UIW_PROMPT(2, 2, "Weight:")
        + new UIW_STRING(12, 2, 36, "6.0 sextillion metric tons.", 50)

        + new UIW_PROMPT(2, 3, "Size:")
        + new UIW_STRING(12, 3, 36, "24,901.55 miles circumference.", 50)

        + new UIW_PROMPT(2, 4, "Makeup:")
        + &(*new UIW_VT_LIST(12, 4, 20, 4,
            WNF_NO_WRAP | WNF_SELECT_MULTIPLE)
            + new UIW_STRING(0, 0, 0, "oxygen -- 46.6%")
            + new UIW_STRING(0, 0, 0, "silicon -- 27.7%")
            + new UIW_STRING(0, 0, 0, "aluminum -- 8.1%")
            + new UIW_STRING(0, 0, 0, "iron -- 5.0%")
            + new UIW_STRING(0, 0, 0, "calcium -- 3.6%")
            + new UIW_STRING(0, 0, 0, "other -- 9.0%"))
```

```
        + &(*new UIW_TEXT(2, 8, 46, 4,
            "The earth is the third planet in distance "
            "outward from the sun.  It is the only "
            "planetary body in the solar system known to "
            "have conditions suitable for life.", 2048)
            + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_VERTICAL));

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

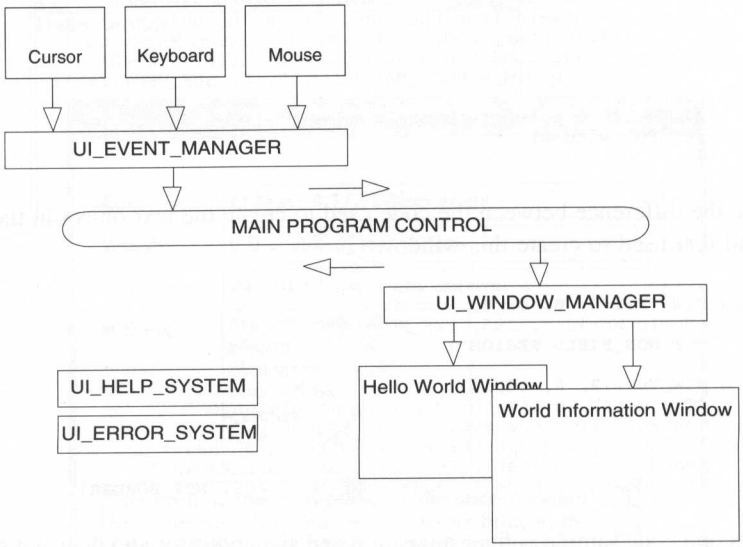
Notice the difference between the code used to create the text object in the first window (1) and that used to create this window (2):

```
new UIW_TEXT(0, 0, 0, 0,
    "Hello World!", 256, TXF_NO_FLAGS,                    (1)
    WOF_NON_FIELD_REGION);

new UIW_TEXT(2, 8, 46, 4,                                 (2)
    "The earth is the third planet in distance "
    "outward from the sun.  It is the only "
    "planetary body in the solar system that has "
    "conditions suitable for life, at least known "
    "to modern science.", 2048, WNF_NO_FLAGS, WOF_BORDER);
```

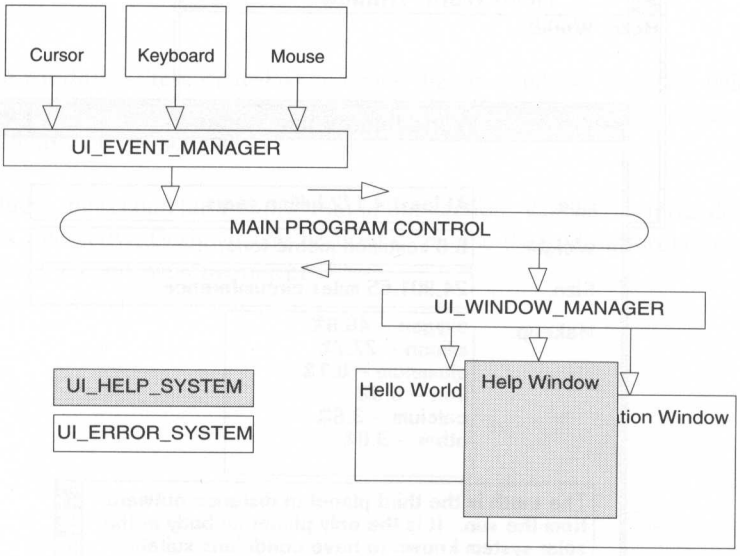
The second code sample defines a position and size indicator and does not set the WOF_NON_FIELD_REGION flag. Instead, it uses WOF_BORDER to display the boundaries of the field's region.

Program flow

Now that the help system, error system and world windows have been added, let's look at the initial program flow:



Notice that this program flow is the same as that discussed in the previous tutorial, except that there are two windows on the screen instead of one. This flow remains unchanged until an error occurs or until help is requested. When the help or error system adds its window to the screen, the Window Manager changes its control to allow interaction with the third window:



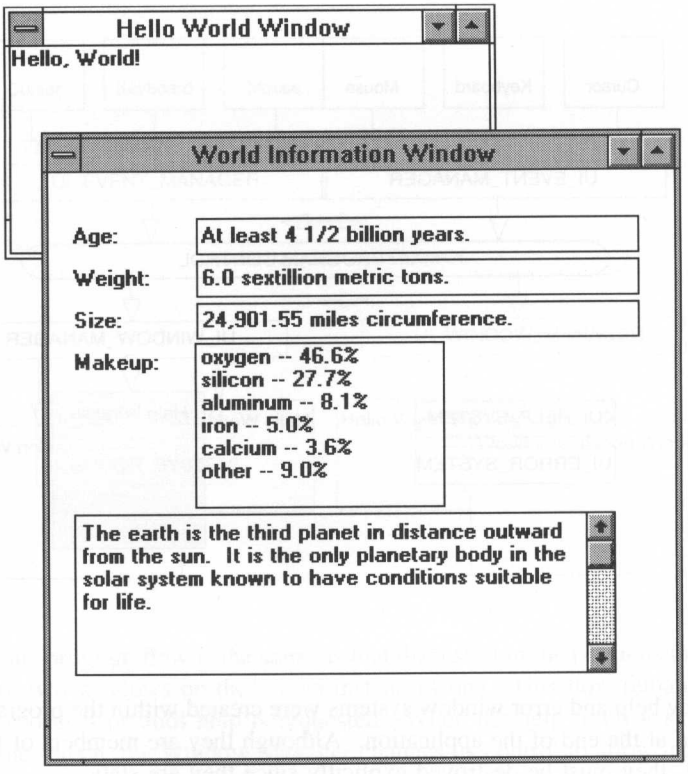
Cleanup

Since new help and error window systems were created within the program, they must be destroyed at the end of the application. Although they are members of UI_WINDOW_-OBJECT, they must be destroyed explicitly since they are static.

```
// Clean up.
delete UI_WINDOW_OBJECT::errorSystem;
delete UI_WINDOW_OBJECT::helpSystem;
delete windowManager;
delete eventManager;
delete display;
```

Run-time features

The first screen that appears when you run the application should be similar to the following:



The added run-time features of this tutorial program are:

Field movement—Either select the window object with the mouse (by clicking the left mouse button while positioned over the object) or press:

- <Tab> to move to the next field on the window.
- <Shift+Tab> to move to the previous field on the window.

Select—Position the mouse pointer on top of a window, then click the left button to select a new current window. To select a new current window from the keyboard, press <Alt+F6>.

Restore, **Maximize**, **Minimize**, **Move**, **Size** and **Close**—The system button created in the **UIW_WINDOW::Generic( )** function allows you to select these options directly from a menu. Position the mouse pointer on top of the system button and click the left button to make the menu appear. Then select the desired option from

the menu by clicking on it. To select this button from the keyboard, press <Alt . > or <Alt+space>.

Delete window—Press <Alt+F4> to delete the top window. This will delete the top window but still allows the application to continue running as long as there is at least one window on the screen.

This concludes the second tutorial program in this section. The next tutorial demonstrates how Zinc's Interactive Designer can be used to reduce the code information associated with windows and sub-window objects.

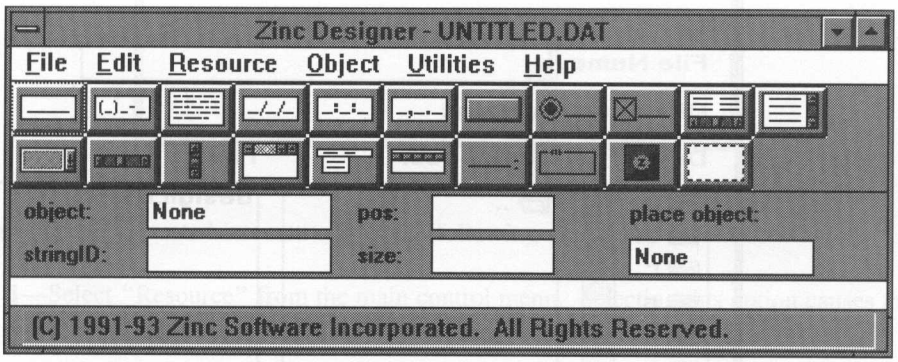
CHAPTER 3 – USING ZINC DESIGNER

The third "Hello World!" tutorial lets us take a step back to see how window creation can be accomplished in a manner of minutes (and a single line of code) using Zinc Designer. The code for this tutorial is located in **\ZINC\TUTOR\HELLO\HELLO3.-CPP**.

Zinc Designer lets you create windows interactively and then incorporate them into your program. The interactive designer is located in **\ZINC\BIN\DESIGN.EXE**. To invoke this program, first make sure you have **\ZINC\BIN** in your PATH environment variable, then type:

```
design <Enter>
```

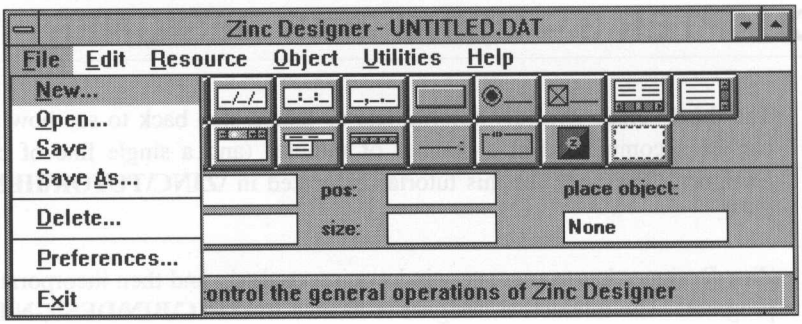
Once the application is running, the following window should be visible on the screen:



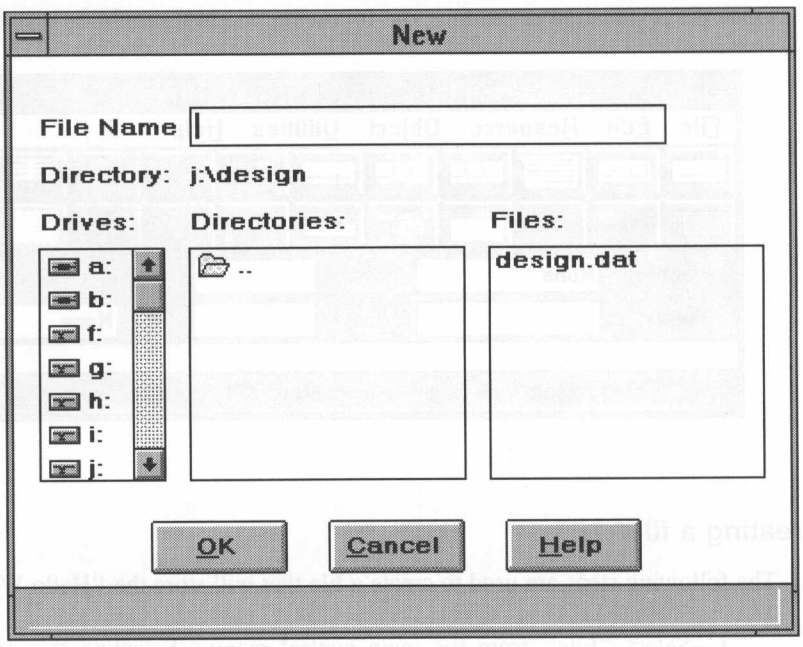
Creating a file

The following steps are used to create a file that will store the "Hello World!" windows:

1—Select "File" from the main control menu. Selecting this option causes the following pop-up menu to be displayed:



2—Select "New.." from the pop-up menu. After you select this item a new window appears:



3—Enter the file name by typing `hello` in the field adjacent to the "File name" prompt.

This is the file name used when the world windows are saved to disk.

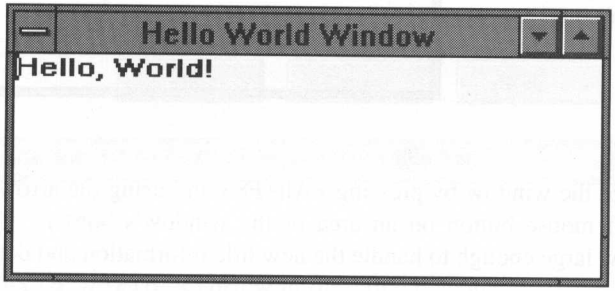
4—Create the file by selecting the "OK" button.

Once the OK button has been selected, Zinc Designer does the following:

- creates a **HELLO.DAT** file that will be used to store the "Hello World!" windows
- removes the "New" window from the screen
- updates the control window's title to reflect the active **HELLO.DAT** file.

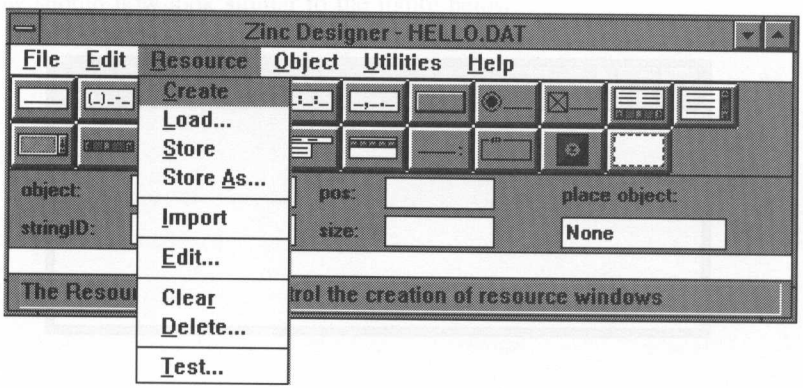
Creating a window

The window we created in the second "Hello World!" tutorial was:

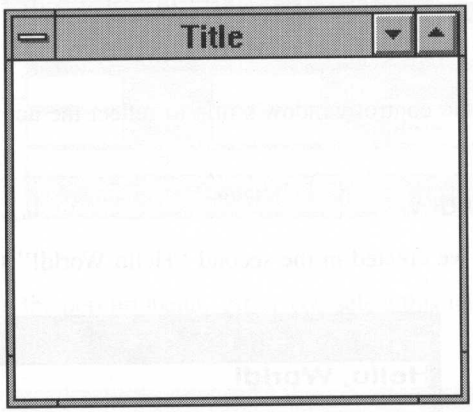


This window is created interactively in the following steps:

1—Select "Resource" from the main control menu. Selecting this option causes the following pop-up menu to be displayed:

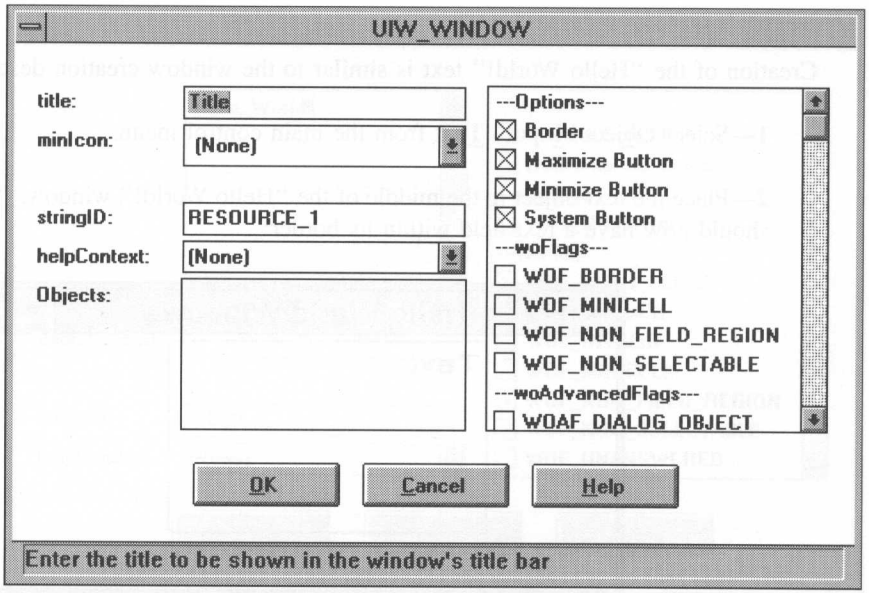


2-Select "Create" from the pop-up window. At this point a generic window appears on the screen:



3—Size the window by pressing <Alt+F8> and using the arrow keys or by pressing the left mouse button on an area of the window's border. 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:

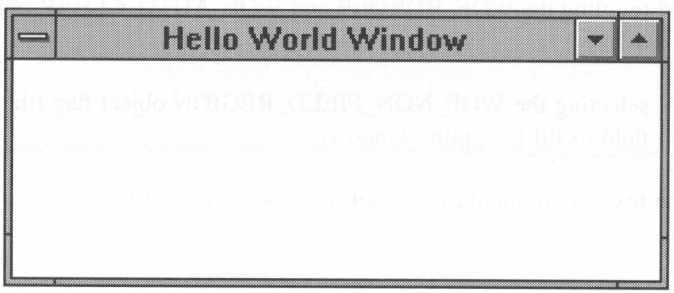


5—Enter `Hello World Window` in the "title:" field.

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

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

Your window should now look similar to the figure below:

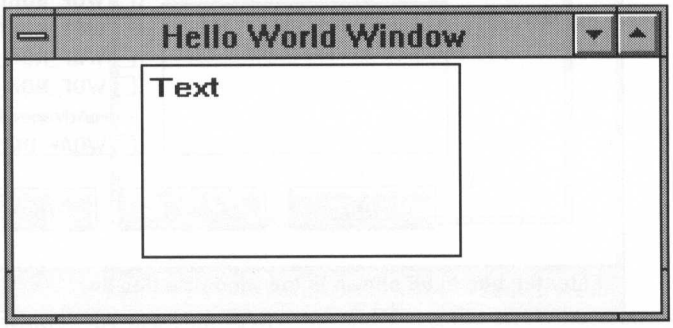


Creating a window object

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

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

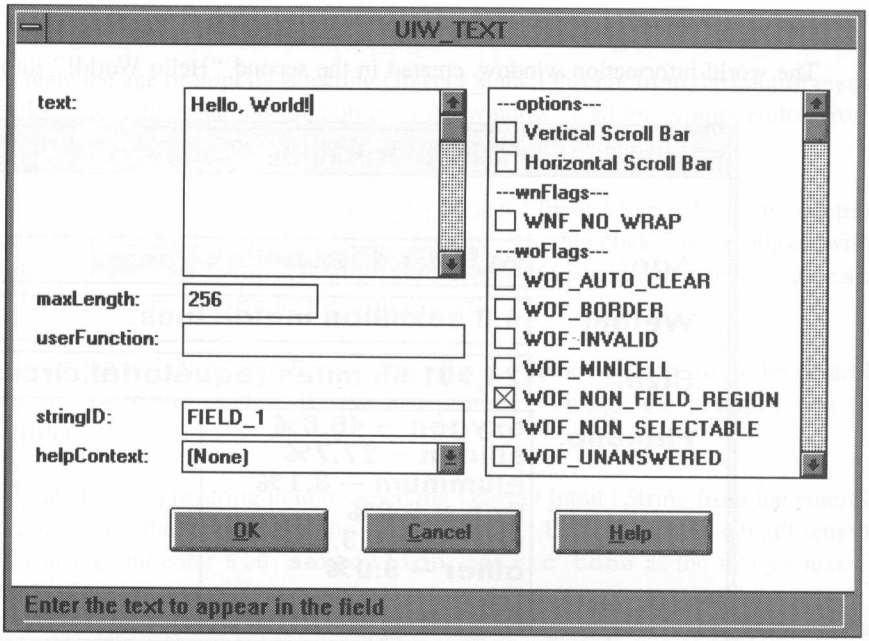
2—Place the text object in the middle of the "Hello World!" window. Your window should now have a text field within its border:



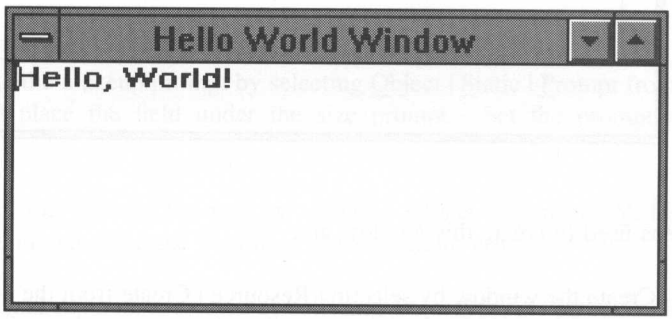
3—Change the default information associated with the text object by:

- calling the text object editor (by double-clicking the left mouse button on the text object)
- typing `Hello World!` in the field adjacent to the "text:" prompt
- typing `256` in the field adjacent to the "maxLength:" prompt
- toggling the WOF_BORDER and WOF_AUTO_CLEAR object flags from on to off
- 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:

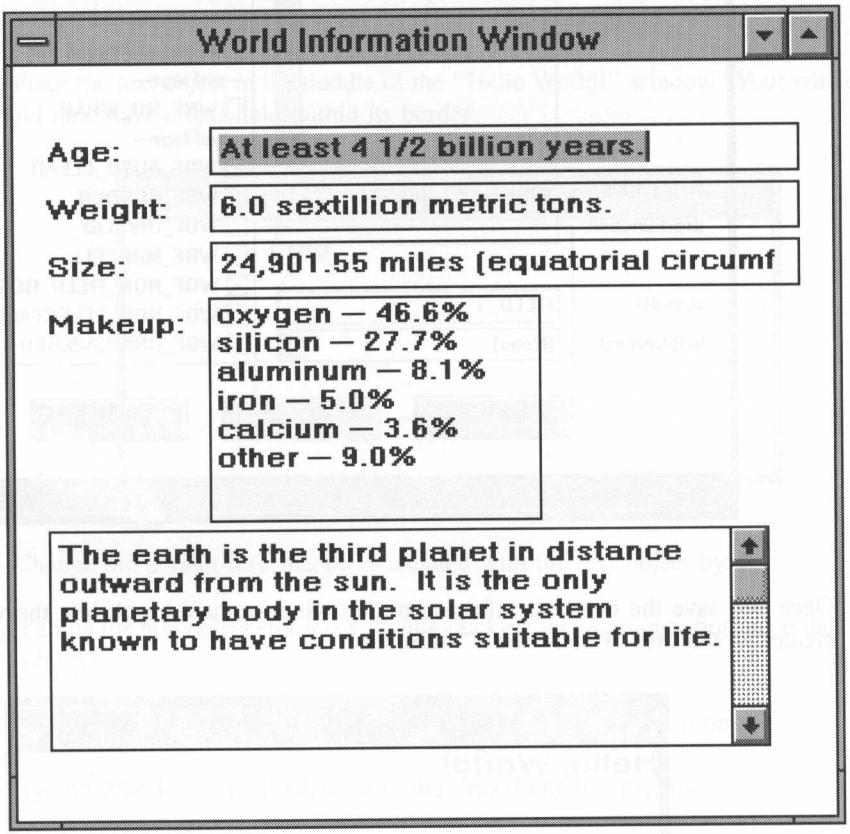


Once you save the new text information, the window should look like the window we created in the first "Hello World!" tutorial:



Creating additional windows

The world information window, created in the second "Hello World!" tutorial was:



The steps used to create this window are:

1—Create the window by selecting Resource | Create from the control menu. Make sure the window is large enough so that the accompanying field information fits within the window's border.

2—Change the window title (in the window editor) to read `World Information Window`.

3—Change the window identification by calling the window editor and entering `WORLD_INFORMATION_WINDOW` as the *stringID*.

4—Select "Ok" to exit the window editor.

5—Create the age prompt by selecting Object | Static | Prompt from the control menu and place the field at the left-top corner of the window. Call the prompt editor, from the window's edit window, and enter **`Age:`** as the prompt's text.

6—Create the age string field by selecting Object | Input | String from the control menu and place the field next to the age prompt. Double click, on the object, with the mouse. Enter **`50`** as the default length for the string field and enter **`At least 4 1/2 billion years`** as the string's text.

7—Create the weight prompt by selecting Object | Static | Prompt from the control menu and place the field under the age prompt. Change the prompt's text to **`Weight:`**.

8—Create the weight string field by selecting Object | Input | String from the control menu and place the field next to the weight prompt. Enter **`50`** as the default length for the string and enter **`6.0 sextillion metric tons`** as the string's text.

9—Create the size prompt by selecting Object | Static | Prompt from the control menu and place the field under the weight prompt. Enter **`Size:`** as the prompt's text.

10—Create the size string by selecting Object | Input | String from the control menu and place the field next to the size prompt. Set the length for this object to be **`50`** and enter **`24,901.55 miles (equatorial circumference)`** as the string's text.

11—Create the makeup prompt by selecting Object | Static | Prompt from the control menu and place the field under the size prompt. Set the prompt's text to be **`Makeup:`**.

12—Create the makeup list box by selecting Object | Control | Vt-List from the control menu and place the field next to the makeup prompt.

13—Each makeup item is created by selecting Object | Input | String from the control menu. After you select this option, place the object anywhere inside the list by clicking on the list box with the mouse. The list automatically provides a default position and size for the newly created item. Additional information can be edited using the string editor.

The only information you need to change is the default text associated with each makeup item. The text for the first item is **`oxygen -- 46.6%`**.

Each additional makeup item should be added in a similar manner. The following items were provided in the original world information window:

- silicon -- 27.7%
- aluminum -- 8.1%
- iron -- 5.0%
- calcium -- 3.6%
- other -- 9.0%

14—Create the world information text field by selecting Object | Input | Text from the control menu, then place the field under the makeup list. The default length for this field is `256` and the default text is `The earth is the third planet in distance outward from the sun. It is the only planetary body in the solar system known to have conditions suitable for life.` To add a vertical scroll bar to the text field, check the "Vertical Scroll Bar" checkbox under the "---options---" heading of the vertical list.

15—Press the "OK" button to complete the changes to the window.

You have now completed the creation of the "Hello World!" information window.

Saving the file

The "Hello World!" windows are saved when you select File | Save from the control menu. Zinc Designer performs the following operations when the windows are saved:

A HELLO.DAT file is updated—This file contains the binary information associated with the objects saved during the design session. You may recall the second tutorial where we created a help file. Help contexts and window objects reside in the same **.DAT** file.

A HELLO.CPP file is created—This file contains the definition for the *objectTable*. This structure provides read access points for objects saved to disk. The entries inside this table depend on the types of objects that were created in the designer.

A HELLO.HPP file is created—This file contains the numeric identifications (ID's associated with those strings you entered next to the "stringID" prompt) and the help context definitions. The string identification for each field within a window is

unique. The items within sub-windows, combo boxes or list boxes have unique numeric identifications within that scope.

Window access

The code used in this tutorial has the same initialization process as each preceding tutorial in that they all follow the same three steps:

- Create the display
- Create the Event Manager and add input devices
- Create the Window Manager

After the Window Manager is created, however, the program adds the two world information windows to the Window Manager:

```
*windowManager
    + new UIW_WINDOW("hello.dat~HELLO_WORLD_WINDOW")
    + new UIW_WINDOW("hello.dat~WORLD_INFORMATION_WINDOW");
```

In the code above, HELLO_WORLD_WINDOW and WORLD_INFORMATION_-WINDOW are retrieved from the **HELLO.DAT** data file and then are added to the Window Manager.

An alternative way of reading the objects from disk is shown below:

```
*windowManager
    + UI_WINDOW_OBJECT::New("hello.dat~HELLO_WORLD_WINDOW")
    + UI_WINDOW_OBJECT::New("hello.dat~WORLD_INFORMATION_WINDOW");
```

This method allows for error correction. If, for example, one of the windows was not found in the file, **New( )** will return a NULL value. When a NULL value is added to the Window Manager, no change is made.

The cleanup associated with this program is the same as that of the previous tutorials.

As you may recall, the designer created a **HELLO.CPP** code file. This file must be compiled and linked with the Hello3 program. It contains an essential object table which is used by window object constructors to read class object information from the data file.

Run-time features

The run-time features associated with this tutorial are the same as that of previous tutorials. The persistent window objects contain all the information necessary to ensure that the application runs as if each object were created with the code shown in previous tutorials.

This concludes the final tutorial program in this section. The next section describes the transition from C programming to C++ programming and how to implement C++ programs using Zinc Application Framework.

SECTION II
DICTIONARY

CHAPTER 4 – WHAT IS THE OBJECT OF C++?

Now that you have completed the "Hello World" tutorials, let's step back and take a look at what an object-oriented language (i.e., C++) has to offer. Any C program may be a C++ program since C++ is a superset of C, but any C++ program is not a C program. C is a very powerful language with proven strong points. C++ utilizes these strong points and combines them with the advantages of an object oriented paradigm.

A simple dictionary program has been created, first in C and then in C++, to illustrate the differences between the two languages. **WORD1A.EXE** is the C version and **WORD1B.-EXE** is the C++ version. (**NOTE:** These two programs are DOS only programs). After completing this tutorial, you should be able to understand:

- classes
- data hiding
- constructors and destructors
- inheritance and deriving classes
- function overloading
- operator overloading
- local variable declaration
- dynamic variable initialization.

The code for the "Word" tutorial programs is located in **\ZINC\TUTOR\WORD**. (See "Chapter 1—Initializing the Library" for information on compiling for each Zinc-supported platform.)

Once the Word1 programs have been compiled, they can be run by typing the program name followed by the word to be looked up in the dictionary. Since this is a tutorial, there are only four words available: **bad**, **begin**, **end** and **good**. To run these programs type the following at the command line:

```
WORD1A good
```

or

```
WORD1B good
```

and then press return. You should see the following printed on the screen:

```
good - Having positive or desirable qualities.
       synonyms - generous, kind, honest.
        antonyms - bad, poor, adverse.
```

Discovering objects

The purpose of any object-oriented language is to provide a logical means of code and data encapsulation. In C++, this is accomplished with an object known as a class. A class is a user defined type structure. Although it may seem strange to think of a type structure as containing code, this is the heart of C++, and it is very powerful. Before we study classes, let's take a look at the file **WORD1A.H**, the header file from the C program.

```
typedef struct
{
    char string[64];
} SYNONYM, ANTONYM;

typedef struct WORD_STRUCT
{
    char string[64];
    char definition[1024];
    int synonymCount;
    SYNONYM synonym[10];
    int antonymCount;
    ANTONYM antonym[10];
} WORD;
```

The preceding C structure declarations are useful in that they encapsulate the data for the dictionary word entries. The problem with this type of programming is that there is no functionality directly associated with the data in this structure. Now let's take a look at the C++ version:

```
class D_WORD : public UI_ELEMENT
{
public:
    char *string;
    char *definition;
    WORD_LIST antonymList;
    WORD_LIST synonymList;

    D_WORD(FILE *file);
    ~D_WORD(void) { delete string; delete definition; }
    D_WORD *Next(void) { return ((D_WORD *)next); }
    void Print(void);
};
```

Notice that the C++ version of the word structure uses the keyword class. The first line of the class declaration gives the class name, D_WORD, and the inheritance list. The inheritance list will be described later on in this chapter. Classes provide a more logical association between code and data since they are both members of the same structure.

The functions listed in a class declaration are actually just function prototypes. When one of these member functions is implemented, it must specify that it is part of the class. For example, consider the implementation of the function **Print( )**:

```
void D_WORD::Print(void)
{
    .
    .
    .
}
```

The first line of the function **Print** lists the following items: the return type, the class name, the scope resolution operator (i.e., ::), the function name and the parameter list. The class name followed by the :: is listed to tell the compiler that the function is a member of the D_WORD class.

Data hiding

After the opening curly brace, in the declaration of the class D_WORD, the line public: appears. The keyword, public, denotes the level of data hiding. Members within a class may be declared as public, protected or private. Since the default data hiding level is private, public has been specified in order to make the data and functions accessible outside of the class.

Public class members are available to any function that has a pointer to an instance of the D_WORD class. Protected class members may only be used by functions within the same class and by functions within derived classes. Private members may only be used by other member functions and members of friend classes.

Constructors and destructors

When an instance of a class is created, it is sometimes desirable to initialize certain member variables. This is done with a special type of member function called a constructor. A constructor automatically gets called when an instance of the class is created. Although the constructor may receive parameters, in C++ it cannot have a return value. A class constructor is easily identified, because it has the same name as that class. The class D_WORD is a good example.

```
class D_WORD : public UI_ELEMENT
{
public:
    char *string;
    char *definition;
    .
    .
    .

    D_WORD(FILE *file);
    ~D_WORD(void) {delete string; delete definition;}
    .
    .
    .
};
```

The complement of a constructor is the destructor. This function is automatically called when an instance of a class is deleted. Destructors are useful functions, because they allow actions to occur, such as freeing memory, when the class is destroyed. A destructor can be easily identified, because it has the same name as the class with the exception that it is preceded by a '~'.

Why use classes?

The following loop, taken from inside the main function in the file **WORD1A.C**, is used to read a word from the dictionary, compare it to the search word and print it out.

```
while (!feof(file))
{
    ReadWord(file, &word);
    if (!strcmp(word.string, argv[1]))
    {
        PrintWord(&word);
        break;
    }
}
```

The loop must be present in order for the operation to work correctly. While this implementation does the job and is fairly easy to read, C++ allows you to construct objects (i.e., classes) that contain the data and functions necessary to find and print a word. Consider the following C++ version of the above code:

```
// Create the dictionary.
DICTIONARY dictionary;
.
.
.

// Search for a word match.
D_WORD *word = dictionary.Get(argv[1]);
if (word)
    word->Print();
else
    printf("The word \"%s\" could not be found.\n");
```

The DICTIONARY class has a member function called **Get( )** that is used to find a word that matches its character string parameter. **Get( )** returns a pointer to a D_WORD class that uses one of its member functions, **Print( )**, to print the appropriate data on the screen. This way D_WORD knows how to print itself, and you, as the programmer, just need to tell it to do so. By using classes, a program can be more logically organized into objects that utilize member functions to perform specific tasks.

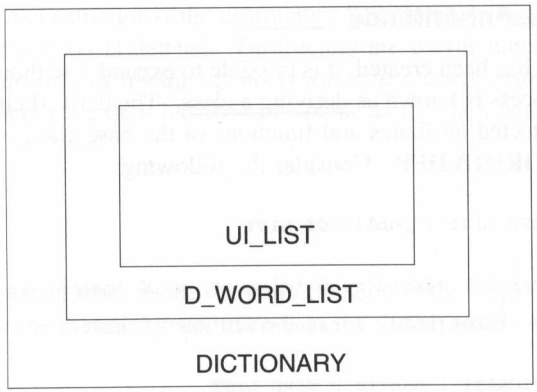
Deriving classes/inheritance

Once an object has been created, it is possible to expand it without modifying the original class. This process is known as deriving a class. The derived class will inherit all of the public and protected attributes and functions of the base class. A good example of this is found in **WORD1B.HPP**. Consider the following:

```
class D_WORD_LIST : public UI_LIST
{
public:
    static int FindWord(void *element, void *matchData)
        {...}
    D_WORD *First(void) { return ((D_WORD *)first); }
};

class DICTIONARY : public D_WORD_LIST
{
public:
    int found;

    DICTIONARY(void);

    D_WORD *Get(const char *word) {...}
};
```

An inheritance list is specified, after the ":", on the first line of the class definition. The class DICTIONARY has been derived from the base class D_WORD_LIST. DICTIONARY is said to inherit D_WORD_LIST. When the inheritance list is declared, the same levels of data hiding apply as in the body of the class definition. In this case, the base class was declared as public, which means that the public members of the base class will be publicly accessible in the derived class.

Derived classes are very useful, since they can be tailored to fit special situations without having to create a whole new class or modifying the original. In the classes declared above, the class D_WORD_LIST is a special type of UI_LIST and the class DICTIONARY is a special type of D_WORD_LIST. The following diagram illustrates the relationship between these three classes:



Function overloading

C++ supports polymorphism, also known as overloading, which allows one name to be used for different, yet similar purposes. Overloaded functions are functions that have the same name, but different parameters. For example, let's take a look at the **Get( )** functions in the UI_LIST class from the file **LIST.HPP**:

```
UI_ELEMENT *Get(int index);
UI_ELEMENT *Get(int (*findFunction)(void *element1, void
    *matchData), void *matchData);
```

These two functions both return a pointer to a particular UI_ELEMENT, but the method that they use and the parameters they require are different. The compiler distinguishes overloaded functions by their parameter lists. Overloading functions allows you to create a generic type of operation while the individual functions define the exact method to be used.

Operator overloading

The operators in C++ can be overloaded in much the same way that functions can. One use for overloaded operators is with a linked list class. The program **WORD1B.EXE** uses a linked list class called UI_LIST that implements overloaded operators. For example:

```
UI_LIST &operator+(UI_ELEMENT *element) { ... };
UI_LIST &operator-(UI_ELEMENT *element) { ... };
```

The '+' and '-' operators have been overloaded to perform add and subtract operations on the linked list. It is important to note that the original operators have not been disabled. For example, the line of code "int count = 3 + 9;" will still perform a

summation and assign the result to the variable count. As with function overloading, the operands will allow the compiler to differentiate which function is called. Here is an example of how to use these operators:

```
UI_LIST list;
UI_ELEMENT element;
.
.
.

list + &element;
```

This use of overloaded operators provides for a much more intuitive piece of code.

Local variables

Local variables in C must be declared at the start of the current block. As an example, let's look at the function main in the C file **WORD1A.C**:

```
main(int argc, char *argv[])
{
    WORD word;
    FILE *file;


    // Make sure there is a word.
    if (argc < 2)
    {
        printf("Usage: WORD1A <word>\n");
        return(0);
    }

    // Make sure the dictionary exists.
    file = fopen("word.dct", "rt");
    .
    .
    .

}
```

Notice that the variable *file* must be declared at the start of the current block even though it is not used until later. If this variable were declared in the middle of the block where it is used, it would cause an error in C.

In C++, variables may be declared as they are needed. This conforms more closely to the idea of data encapsulation that was mentioned earlier. Now let's examine the C++ main function, taken from **WORD1B.CPP**:

```
main(int argc, char *argv[])
{
    // Make sure there is a word.
    if (argc < 2)
    {
        printf("Usage: WORD1B <word>\n");
        return(0);
```

```
        }

        // Create the dictionary.
        DICTIONARY dictionary;
        if (!dictionary.found)
        {
            .
            .
            .

        }

        // Search for a word match.
        D_WORD *word = dictionary.Get(argv[1]);
        .
        .
        .

    }
```

This C++ example shows two very important concepts: local variable declaration and dynamic initialization. As has already been mentioned, variables in C++ may be declared where they are used and not just at the top of the current block.

Any local and global variables in C++ can be dynamically initialized using any valid expression. For example, the variable *word*, from the above piece of code, is initialized, at run-time, with the return value from a call to **dictionary.Get( )**. This is very different from C which requires that a variable's initial value be known at compile time. Dynamic initialization will allow a variable to be initialized based on the value of another variable or on the return value of a function.

Conclusion

You should now be familiar with the major differences between C and C++. If you implement the ideas that were discussed in this chapter, you will be on your way to writing concise, easily maintainable and powerful code.

CHAPTER 5 – EVENT FLOW

This tutorial demonstrates how Zinc Application Framework can be used to enhance an existing C++ program and how events are handled throughout the system. When you are finished, you should understand:

- how a window and its fields are created
- the use of window objects to display information and receive input from the user
- the use of user functions to check data input
- how events are handled throughout the system

In this tutorial, we will examine a modified version of the dictionary program that was discussed in the previous chapter. The program **WORD2.EXE** will be used to demonstrate these new concepts.

The source code associated with this program is located in **\ZINC\TUTOR\WORD**. It contains the following files:

WORD2.CPP—This file contains the main program loop (i.e., **UI_APPLICATION::Main( )**) as well as the implementation of the DICTIONARY_WINDOW, DICTIONARY and D_WORD classes.

WORD2.HPP—This file contains the declarations for the DICTIONARY_WINDOW, DICTIONARY and D_WORD classes.

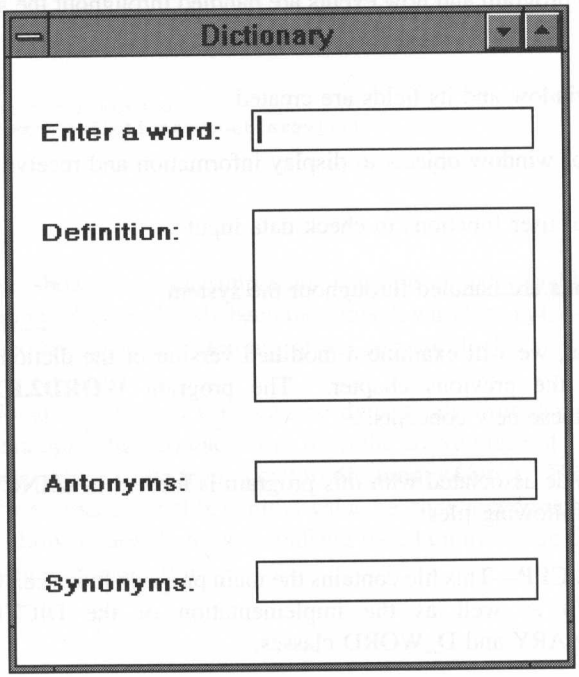
WORD.DCT—This file is the dictionary database file.

.DEF**, ***.RC**—These files are the environment specific definition and resource files required when compiling for Windows or OS/2. (**NOTE:** The **W*. files are for Windows and the **O*.*** files are for OS/2.)

***.MAK**—These files are the compiler-dependent makefiles associated with the Word program. (See "Chapter 1—Initializing the Library" for information on compiling for each Zinc-supported platform.)

Program execution

The operation of the dictionary program can be seen by compiling and running the application **WORD2.EXE**. The following should appear on the screen:



To look up a word, position the cursor on the "Enter a word" field by either clicking on it with the left mouse button or pressing <TAB> until the cursor appears in the field. Once the field becomes current, simply type a word and press enter. If the word is in the dictionary, the definition, antonyms and synonyms will be displayed. If the word is not in the dictionary and cannot be displayed, an error message will appear saying "That word was not found in the dictionary." Remember, as in the previous tutorial, that good, bad, begin and end are the only words available and the search is case-sensitive.

When you are finished using the dictionary, exit the program by either selecting "Close" from the system button's pop-up menu or by pressing <Alt+F4>.

Class definitions

The dictionary window is implemented with a class called DICTIONARY_WINDOW. The actual dictionary is comprised of the classes: DICTIONARY, D_WORD and

WORD_ELEMENT. The definition for the DICTIONARY_WINDOW class is given below:

```
class DICTIONARY_WINDOW : public UIW_WINDOW
{
public:
    DICTIONARY_WINDOW(void);
    ~DICTIONARY_WINDOW(void);

    int dictionaryOpened;

private:
    DICTIONARY *dictionary;
    UIW_STRING *inputField;
    UIW_TEXT *definitionField;
    UIW_STRING *antonymField;
    UIW_STRING *synonymField;

    static EVENT_TYPE LookUpWord(UI_WINDOW_OBJECT *string, UI_EVENT &event,
        EVENT_TYPE ccode);
};
```

DICTIONARY_WINDOW uses the following member variables:

- *dictionaryOpened* is a variable that tells if the dictionary was successfully opened. Since constructors cannot return values, we must set a flag to denote the dictionary status. This value is public so that the controlling program can verify that the dictionary was created.

- *dictionary* is the pointer to the dictionary itself. The instance of DICTIONARY that is pointed to by this pointer is allocated in the constructor for DICTIONARY_-WINDOW. This variable is only used by the DICTIONARY_WINDOW class and therefore is made private.

- *inputField* is a pointer to the UIW_STRING field that is used to collect the input word from the user. This variable is only used by the DICTIONARY_WINDOW class and therefore is made private.

- *definitionField* is a pointer to the UIW_TEXT field that is used to display the definition for the input word. This variable is only used by the DICTIONARY_-WINDOW class and therefore is made private.

- *antonymField* is a pointer to the UIW_STRING field that is used to display the antonyms for the input word. This variable is only used by the DICTIONARY_-WINDOW class and therefore is made private.

- *synonymField* is a pointer to the UIW_STRING field that is used to display the synonyms for the input word. This variable is only used by the DICTIONARY_-WINDOW class and therefore is made private.

The definition for the DICTIONARY class is given below:

```
class DICTIONARY : public UI_LIST
{
public:
    int opened;

    DICTIONARY(char *name);

    static int FindWord(void *element, void *matchData);
    D_WORD *First(void);
    D_WORD *Get(const char *word);
};
```

DICTIONARY uses the following member variable:

- *opened* is a variable that tells if the dictionary was successfully opened. Since constructors cannot return values, we must set a flag to denote the dictionary status. This value is public so that the controlling program can verify that the dictionary was created.

The definition for the D_WORD class is given below:

```
class D_WORD : public UI_ELEMENT
{
public:
    char *string;
    char *definition;
    UI_LIST antonymList;
    UI_LIST synonymList;

    D_WORD(FILE *file);
    ~D_WORD(void);

    D_WORD *Next(void);
};
```

D_WORD uses the following member variables:

- *string* is a variable that contains the actual word entry in the dictionary.
- *definition* is a variable that contains the definition string of the word.
- *antonymList* is a list of antonyms that apply to the dictionary entry.
- *synonymList* is a list of synonyms that apply to the dictionary entry.

The definition for the WORD_ELEMENT class is given below:

```
class WORD_ELEMENT : public UI_ELEMENT
{
public:
    char *string;
```

```
    WORD_ELEMENT(const char *a_string);
    ~WORD_ELEMENT(void);

    WORD_ELEMENT *Next(void);
};
```

WORD_ELEMENT uses the following member variables:

- *string* is a variable that contains a character string. In this example, it is used to hold either antonyms or synonyms.

Creating the window

In this version of the dictionary program, we will create a specialized window class called DICTIONARY_WINDOW that will be derived from the Zinc window class UIW_WINDOW. Instead of using the existing UIW_WINDOW class, we will derive one that will not only handle the I/O with the window fields, but will also maintain the communication with the dictionary itself.

When the DICTIONARY_WINDOW constructor is called, the window itself is automatically created since UIW_WINDOW was declared as the base class. Once inside the constructor, each of the fields is created and then added to the window. Objects are added to the window using the C++ reserved word *this* and the overloaded + operator. The DICTIONARY_WINDOW constructor is shown below:

```
DICTIONARY_WINDOW::DICTIONARY_WINDOW(void) : UIW_WINDOW(16, 6, 41, 14)
{
    .
    .
    .
    if (dictionaryOpened)
    {
        // Create the window fields.
        inputField = new UIW_STRING(17, 1, 20, "", 40, STF_NO_FLAGS,
            WOF_BORDER | WOF_AUTO_CLEAR, DICTIONARY_WINDOW::LookUpWord);
        definitionField = new UIW_TEXT(17, 3, 20, 4,"", 100, TXF_NO_FLAGS,
            WOF_BORDER);
        antonymField = new UIW_STRING(17, 8, 20, "", 50, TXF_NO_FLAGS,
            WOF_BORDER);
        synonymField = new UIW_STRING(17, 10, 20, "", 50, TXF_NO_FLAGS,
            WOF_BORDER);

        *this
            + new UIW_BORDER
            + new UIW_MAXIMIZE_BUTTON
            + new UIW_MINIMIZE_BUTTON
            + new UIW_SYSTEM_BUTTON
            + new UIW_TITLE("Dictionary")
            + new UIW_PROMPT(2, 1, "Enter a word:")
            + inputField
            + new UIW_PROMPT(2, 3, "Definition:")
            + definitionField
            + new UIW_PROMPT(2, 8, "Antonyms:")
            + antonymField
```

```
            + new UIW_PROMPT(2, 10, "Synonyms:")
            + synonymField;
        .
        .
        .

    }
}
```

The necessary objects are added to the window inside the constructor so that when the DICTIONARY_WINDOW class is created, only a few lines are required to create it and display it on the screen. Examine the following piece of code taken from the main function in the **WORD2.CPP** file:

```
// Create the dictionary window.
DICTIONARY_WINDOW *dictionary = new DICTIONARY_WINDOW();

// If the dictionary was opened, add it to the window manager.
if (dictionary->dictionaryOpened)
    *windowManager + dictionary;
else
{
    dictionary->errorSystem->ReportError(windowManager, -1,
        "The dictionary file 'WORD.DCT' was not found.");
    delete dictionary;
}
```

If the objects were not added in the constructor but were added to the newly created instance of the DICTIONARY_WINDOW class, then each time the class was created, there would be a significant duplication of code. Adding the objects inside the constructor provides a stronger encapsulation of data and code.

The user function

This version of the dictionary tutorial allows the user to type a word in the "Enter a word" field and press <ENTER> to display either the word's data or an error message. This is done through the use of a user function. We will use the user function to compare the data entered into the object's field to the words in the dictionary. User functions can be assigned to any window object through the object's constructor. Look at the DICTIONARY_WINDOW constructor:

```
inputField = new UIW_STRING(17, 1, 20, "", 40, STF_NO_FLAGS,
    WOF_BORDER | WOF_AUTO_CLEAR, DICTIONARY_WINDOW::LookUpWord);
```

When the UIW_STRING field is constructed, the last parameter references the user function. Adding a user function allows the UIW_STRING object to call this function whenever the string field is made current, non-current or when the <ENTER> key is pressed.

In order for the compiler to generate an address, user functions must be declared as static. The user function **LookUpWord( )** has the following parameters (required for all user functions):

- $returnValue_{out}$ is the result of the operation. Most often *ccode* is the value returned. However, if -1 is returned, the calling window object will be informed that some error occurred and the text is restored to its previous value.
- $object_{in}$ is a UI_WINDOW_OBJECT pointer to the object that invoked this function. In this case, the calling object is a UIW_STRING field whose parent is a DICTIONARY_WINDOW object. This pointer must be typecast by the programmer if object specific information is needed.
- $event_{in}$ is the event that caused this function to be called.
- $ccode_{in}$ is the logical interpretation of the event that caused this function to be called.

Consider the implementation of **LookUpWord( )**:

```
EVENT_TYPE DICTIONARY_WINDOW::LookUpWord(UI_WINDOW_OBJECT *object,
    UI_EVENT &event, EVENT_TYPE ccode)
{
    .
    .
    .

}
```

Since the user function is called when the string field receives the S_CURRENT, S_NON_CURRENT or L_SELECT messages, the first step is to determine if the ccode is S_CURRENT. In the dictionary tutorial, if the input string field is just becoming current, a new word has not been entered and the function returns without doing anything. Examine the initial check in **LookUpWord( )**:

```
// Return if the field is just becoming current.
if (ccode == S_CURRENT)
    return errorCode;
```

If the input field is becoming non-current, the dictionary must be called to verify the input word. To do this, it must have a pointer to the current dictionary object. Note that the input string and the dictionary pointer are both members of the DICTIONARY_WINDOW class. Therefore, it is easy to get a pointer to the correct instance of DICTIONARY_-WINDOW, since the object's parent is the DICTIONARY_WINDOW. The following code segment demonstrates how to get a pointer to the parent, DICTIONARY_WINDOW:

```
DICTIONARY_WINDOW *dictionaryWindow = (DICTIONARY_WINDOW *)object->parent;
```

With the *dictionaryWindow* pointer, access can be made to the public variables and functions of the DICTIONARY_WINDOW class, including the variable *dictionary*. In order to see if the word is in the dictionary, the user function calls the function **DICTIONARY::Get( )** by using the *dictionaryWindow* pointer that was initialized above. This function will either return a NULL, if the word is not found, or a pointer to a D_WORD structure that contains the input word and its associated information. If the return value is a valid pointer, the word's information is written to the appropriate window fields by calling each field's **DataSet( )** function. In the event of error, an error message is displayed. The return value for the user function is 0 upon success or -1 upon error.

Following events

Now that a windowing system has been added to the dictionary program, let's study how events are passed through the system. For example, what happens between the time that a user types a character (e.g., 'g') in the "Enter a word" field and the time that the letter appears on the screen? In this section, we will examine event flow in DOS and in the Windows environment.

NOTE: Currently, Zinc supports two additional GUI environments (i.e., OS/2 and Motif). Although the messages and their meanings differ, the OS/2 and Motif environments pass messages in the same way as Windows.

Event flow—DOS

When a key is pressed, the character is placed in the computer's BIOS keyboard buffer. This is done by the computer and is independent of any application software that is running. The "do" loop in the main function of the program controls the dispatching of events.

```
EVENT_TYPE ccode;
UI_EVENT event;
do
{
    // Get input from the user.
    eventManager->Get(event);

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

When **eventManager->Get( )** is called, each of the devices attached to the Event Manager is polled. If a device, such as the keyboard, has an event waiting, a UI_EVENT structure is created, filled with the proper data and put on the end of the event queue. Let's assume that there were no other events on the queue when the 'g' key event was put on the queue. The *event* variable passed to the **Get( )** function is filled with the next

event in the event queue (e.g., the 'g' key event). When program control returns from the **Get( )** function, the event is passed to the Window Manager with the call **windowManager->Event( )**.

Once the Window Manager has control, it sends the event to the current window object. Each time an object gets the event, it passes it to its current child object. It continues to do this until it gets to the bottom of the hierarchy. Once the control gets to the bottom-most object, the object tries to interpret the event. If it can, it does and then returns a control code. If it cannot, it returns an S_UNKNOWN message to its parent and its parent tries to interpret the event, and so on. In this manner, the events are interpreted from the bottom up. If an S_UNKNOWN message is returned to the Window Manager and the event carries a specified region (such as with a mouse click), the Window Manager checks to see if another object should become current. If so, that object is made current and the event is passed to the current object. If no window can handle the event, the Window Manager just returns an S_UNKNOWN message and the event is ignored.

In the case of the 'g' key in the dictionary example, the Window Manager's current object is the dictionary window. The window receives the event and sends it to its own current object which is the UIW_STRING field. The string's **Event( )** function receives the event and calls **UI_WINDOW_OBJECT::LogicalEvent( )** to try to find a logical mapping of the event. Once it determines that the event is a keystroke and that it contains a 'g' character, the character is copied into the string's memory buffer. A call is made to **UIW_STRING::Redisplay( )**, which in turn calls **display->Text( )** to actually paint the character on the screen. A control code is then returned to the object's parent and finally to the Window Manager which returns to the main do loop, where the sequence starts over again.

Event flow—Microsoft Windows

The Microsoft Windows version of Zinc Application Framework is somewhat simpler than the DOS version. This is due to the fact that Windows does the I/O itself and Zinc only handles the resulting messages. When a UIW_STRING field is created, Zinc creates an actual Windows string object. In the Windows version, Zinc serves as a layer between the existing Windows system and the user application that was written using Zinc. This model allows programs to be easily ported to any of the environments that Zinc supports.

In order to follow an event through the Zinc system while running under Windows, it is necessary to explain something about the way in which Windows passes messages. First of all, Windows messages are put on a Windows message queue where they can be dispatched directly to the current field on the current object. Messages are passed to an object via a special member function known as a "callback" function. A callback function is a Window's function used for sending messages.

Now, let's consider the example of the 'g' key being pressed while a UIW_STRING field is current. First, Windows creates a message and puts it on the Windows message queue. Look at the "do" loop in the function **UI_APPLICATION::Main( )**:

```
EVENT_TYPE ccode;
UI_EVENT event;
do
{
    // Get input from the user.
    eventManager->Get(event);

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

When **eventManager->Get( )** is called, it doesn't return until Windows has generated a message. Once this is done, the call to **windowManager->Event( )** instructs Windows to dispatch the message. When a message is dispatched, Windows calls the appropriate object's callback function (i.e., UIW_STRING in this case) saying that the character 'g' was pressed. In this case, the string object's callback function sends the message to the string object's jump procedure which in turn calls **UIW_STRING::Event( )**. At this point, the event continues in the same manner as with the DOS version. After Zinc handles the message, it is passed back to Windows so that the character may be painted on the screen.

Conclusion

You should now understand how a window and its fields are created, how window objects are used to display information and receive input from the user, how user functions can be used to check data input, and how events are handled throughout the system. If you wish to interface with a separate database you can use this program as a template and, instead of using the class DICTIONARY, you will make the appropriate calls to your database program.

CHAPTER 6 – THE ZINC DATA FILE

This tutorial demonstrates database interaction with Zinc Application Framework and the use of the Zinc data file. After finishing this tutorial, you should be able to understand:

- the use of the Zinc data file
- how to add and delete user-defined objects within the Zinc data file.

In this tutorial, we will examine a new version of the dictionary program that has been used in previous chapters. The program **WORD3.EXE** will be used to demonstrate these new concepts.

The source code associated with this program is located in **\ZINC\TUTOR\WORD**. It contains the following files:

WORD3.CPP—This file contains the main program loop (i.e., **UI_APPLICATION::Main( )**) as well as the implementation of the DICTIONARY_WINDOW, DICTIONARY and D_WORD classes.

WORD3.HPP—This file contains the declarations for the DICTIONARY_WINDOW, DICTIONARY and D_WORD classes.

WORD_WIN.CPP—This file contains the object table for the objects that were created in the designer.

WORD_WIN.DAT—This file is the data file that was created by the Designer. It contains the data information necessary to create the dictionary window and its fields.

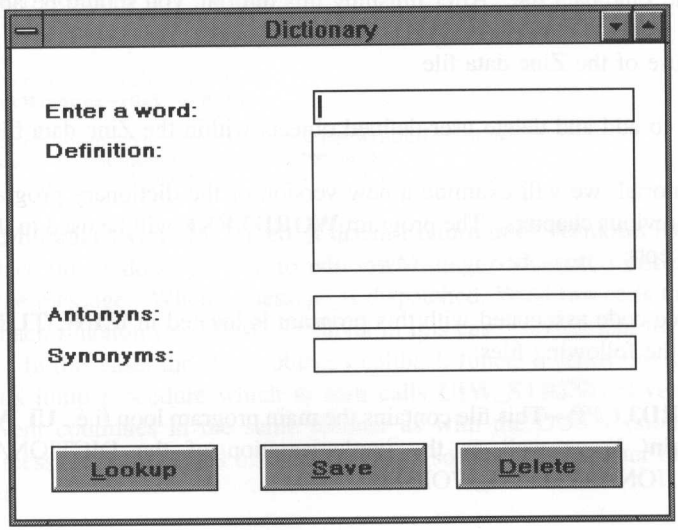
WORD_WIN.HPP—This file contains the header information for the **WORD_WIN.DAT** file. This file contains the #define directives for the numberID's of the file objects in the data file. It also contains the help file header information for the **WORD_WIN.DAT** file.

.DEF**, ***.RC**—These files are the environment specific definition and resource files required when compiling for Windows or OS/2. (**NOTE:** The **W*. files are for Windows and the **O*.*** files are for OS/2.)

***.MAK**—These files are the compiler-dependent makefiles associated with the Word program. (See "Chapter 1—Initializing the Library" for information on compiling for each Zinc-supported platform.)

Program execution

The operation of this version of the dictionary program can be seen by running the application **WORD3.EXE**. The following should appear on the screen:



At this point, the dictionary database will be empty. To add words to the dictionary, simply type the word, definition, antonyms and synonyms in the appropriate fields and press the "Save" button on the bottom of the window. To look up a word, type the word in the "Enter a word:" field and press the "Lookup" button. To delete a word, type the word in the "Enter a word:" field and press the "Delete" button.

When you are finished using the dictionary, exit the program by either selecting "Close" from the system button's pop-up menu or by pressing <Shift+F4>.

Class definitions

The dictionary window is implemented with a class called DICTIONARY_WINDOW. The definition of the DICTIONARY_WINDOW class is given below:

```
class EXPORT DICTIONARY_WINDOW : public UIW_WINDOW
{
public:
    DICTIONARY_WINDOW(char *dictionaryName);
    ~DICTIONARY_WINDOW(void);

    EVENT_TYPE Event(const UI_EVENT &event);
```

```
private:
    DICTIONARY *dictionary;
    UIW_STRING *inputField;
    UIW_TEXT *definitionField;
    UIW_STRING *antonymField;
    UIW_STRING *synonymField;

    static EVENT_TYPE ButtonFunction(UI_WINDOW_OBJECT *item,
        UI_EVENT &event, EVENT_TYPE ccode);
};
```

DICTIONARY_WINDOW uses the following member variables:

- *dictionary* is the pointer to the dictionary itself. The instance of DICTIONARY that is pointed to by this pointer is allocated in the constructor for DICTIONARY_-WINDOW. This variable is only used by the DICTIONARY_WINDOW class and therefore is made private.

- *inputField* is a pointer to the UIW_STRING field that is used to collect the input word from the user. This variable is only used by the DICTIONARY_WINDOW class and therefore is made private.

- *definitionField* is a pointer to the UIW_TEXT field that is used to display the definition for the input word. This variable is only used by the DICTIONARY_-WINDOW class and therefore is made private.

- *antonymField* is a pointer to the UIW_STRING field that is used to display the antonyms for the input word. This variable is only used by the DICTIONARY_-WINDOW class and therefore is made private.

- *synonymField* is a pointer to the UIW_STRING field that is used to display the synonyms for the input word. This variable is only used by the DICTIONARY_-WINDOW class and therefore is made private.

The definition for the DICTIONARY is as follows:

```
class EXPORT DICTIONARY : public UI_STORAGE
{
public:
    DICTIONARY(char *name) :
        UI_STORAGE(name, UIS_OPENCREATE | UIS_READWRITE);
    ~DICTIONARY();

    D_WORD *Get(const char *word);
};
```

The definition for the D_ENTRY class is as follows:

```
class D_ENTRY
{
public:
    int wasLoaded;
    char *word;
    char *definition;
    char *antonym;
    char *synonym;

    D_ENTRY(const char *name, UI_STORAGE *file,
        UIS_FLAGS sFlags = UIS_READ);
    ~D_ENTRY(void);

    static D_ENTRY *New(const char *name, UI_STORAGE *file,
        UIS_FLAGS sFlags = UIS_READ);
    void Save(void);
};
```

D_ENTRY uses the following member variables:

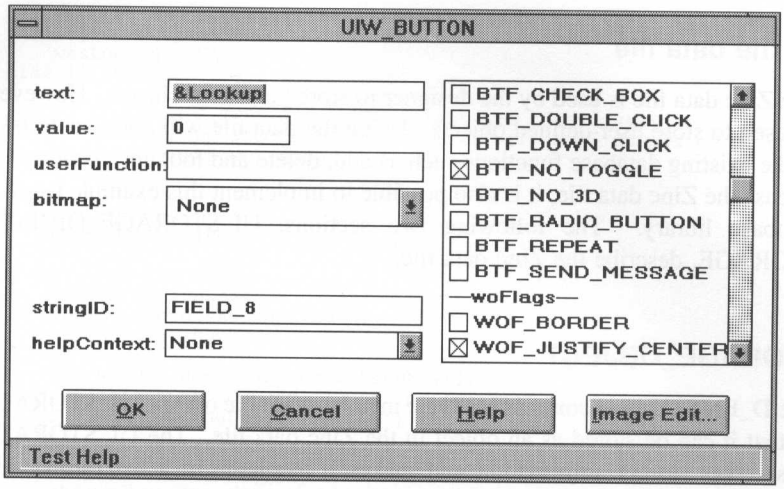
- *wasLoaded* is a flag used to denote whether or not the word entry was loaded.
- *word* is a variable that contains the actual word entry in the dictionary.
- *definition* is a variable that contains the definition string of the word.
- *antonym* is a list of antonyms that apply to the dictionary entry.
- *synonym* is a list of synonyms that apply to the dictionary entry.

Creating the window

The window for this program was created using Zinc Designer and is contained in the file **WORD_WIN.DAT**. You may recreate this window by starting Zinc Designer and build it as it appears in the Program execution section above. (See "Chapter 3—Using Zinc Designer" for steps on using the designer.)

When a field is created, the designer gives it a default *stringID*. A string identification is a label that is used to uniquely identify each object. The default *stringID*'s are of the form FIELD_1, FIELD_2, etc. In most cases, the default *stringID* is sufficient. However, in order to access a particular field, it is helpful to specify a new *stringID*. In the designer, an object's string identification can be changed by bringing up the object's edit window and entering a new string identification in the "stringID:" field. For example, to change the "Lookup" button's stringID, make the button current and bring up its edit window. The new stringID for the "Lookup" button should be **LOOKUP_BUTTON** and

LOOKUP_BUTTON should be entered in the "stringID" field of the button's edit window (shown below):



Now that the window has been set up, it is necessary to connect the "Lookup" button to the function that will look up the word. This is done by assigning a function to the button's *userFunction* member variable. To get a pointer to the button, first create the window that contains the button. In this example, the window is created when the DICTIONARY_WINDOW constructor is called. Then call the window's **Information( )** function with the GET_STRINGID_OBJECT request and the stringID that was assigned to the button when it was created in the Designer. This will return a void pointer that points to the button, so it will need to be cast as a UIW_BUTTON *. These steps are shown by the following piece of code:

```
DICTIONARY_WINDOW::DICTIONARY_WINDOW(char * dictionaryName) :
    UIW_WINDOW("word_win.dat~WINDOW_DICTIONARY")
{
    .
    .
    .

    // Set the user functions to the buttons.
    UIW_BUTTON *button;
    button = (UIW_BUTTON *)Get(DCT_LOOKUP_BUTTON);
    button->userFunction = DICTIONARY_WINDOW::ButtonFunction;
    .
    .
    .

}
```

With a pointer to the button, **ButtonFunction( )** can be assigned as the *userFunction*. **ButtonFunction( )** is a generic function that all the DICTIONARY_WINDOW buttons

call. This way they can be dispatched through a single static function instead of having a user function for each of the buttons on the window.

Using the data file

The Zinc data file is used by the designer to store persistent objects. However, it can also be used to store user-defined objects. Using the data file will allow us to take advantage of the existing database functions such as add, delete and lookup. Although we will only discuss the Zinc data file, it is also possible to implement this example using a third party database library. The following two sections, UI_STORAGE_OBJECT and UI_-STORAGE, describe the Zinc data file.

UI_STORAGE_OBJECT

The D_ENTRY class contains a private member variable of type UI_STORAGE_OBJECT so that it can be stored as an object in the Zinc data file. The UI_STORAGE_OBJECT class takes storage information and makes it available in a list format. It is used in conjunction with the UI_STORAGE class to identify an object's location within a file.

Although D_ENTRY contains a UI_STORAGE_OBJECT member variable, there are three functions that must be set up properly in order for it to function as a persistent object. These functions are: **constructor**, **New( )** and **Save( )**.

The constructor

The constructor for the D_ENTRY class takes the following three parameters:

- *name* is the name of the storage object.
- *file* is the file containing the object. If the object is not found in the file, the member *wasLoaded* is set to FALSE. Otherwise, *wasLoaded* is set to true and the object is retrieved from the data file.
- *flags* indicate whether the object is to be loaded or created. If the entry is found and the UIS_CREATE flag is set, the entry will be deleted from the file in preparation for the new entry being saved.

```
D_ENTRY::D_ENTRY(const char *name, UI_STORAGE *file, UIS_FLAGS flags) :
    word(NULL), definition(NULL), antonym(NULL), synonym(NULL)
{
    object = new UI_STORAGE_OBJECT(*file, name, ID_DICTIONARY_ENTRY, flags);

    // Check to see if object was found in the file.
    if (object->objectError)
        wasLoaded = FALSE;
    else if (FlagSet(flags, UIS_CREATE))
    {
        // If the UIS_CREATE option is set, the record will be
        // overwritten and the previous entry (if any) should be deleted.
        file->DestroyObject(name);
    }
    else
    {
        wasLoaded = TRUE;

        // Load the word.
        USHORT stringLength;
        object->Load(&stringLength);
        if (stringLength)
        {
            word = new char[stringLength+1];
            object->Load(word, 1, stringLength);
            word[stringLength] = '\0';
        }

        // Load the definition.
        object->Load(&stringLength);
        if (stringLength)
        {
            definition = new char[stringLength+1];
            object->Load(definition, 1, stringLength);
            definition[stringLength] = '\0';
        }

        // Load the antonyms.
        object->Load(&stringLength);
        if (stringLength)
        {
            antonym = new char[stringLength+1];
            object->Load(antonym, 1, stringLength);
            antonym[stringLength] = '\0';
        }

        // Load the synonyms.
        object->Load(&stringLength);
        if (stringLength)
        {
            synonym = new char[stringLength+1];
            object->Load(synonym, 1, stringLength);
            synonym[stringLength] = '\0';
        }
    }
}
```

If the word entry is found in the file, each of the object's fields are loaded. Each field is preceded by an unsigned short that gives the size, in bytes, of the field to follow. Then the number of bytes comprising the field itself are read in. Since each object stores its own fields, the constructor knows how many fields to read in.

The New function

When a word is looked up in the dictionary and its related information is read in, a function called **D_ENTRY::New( )** is called. The **New( )** function discussed here is a member of the class and not the **new** operator of C++.

In this tutorial, only one type of object is stored in the data file, and the benefits of the **New( )** function are not apparent. However, **New( )** is included as a matter of form. The reason for having a static **New( )** function in a class is to be able to take the address of a function that will call the constructor. A good example of this can be seen in the implementation of an object table. Examine the following object table taken from the file **WORD_WIN.CPP**:

```
UI_ITEM *UI_WINDOW_OBJECT::userTable = _userTable;

static UI_ITEM _objectTable[] =
{
    { ID_BORDER, VOIDF(UIW_BORDER::New), "BORDER", 0 },
    { ID_BUTTON, VOIDF(UIW_BUTTON::New), "BUTTON", 0 },
    { ID_MAXIMIZE_BUTTON, VOIDF(UIW_MAXIMIZE_BUTTON::New),
      "MAXIMIZE_BUTTON", 0 },
    { ID_MINIMIZE_BUTTON, VOIDF(UIW_MINIMIZE_BUTTON::New),
      "MINIMIZE_BUTTON", 0 },
    { ID_PROMPT, VOIDF(UIW_PROMPT::New), "PROMPT", 0 },
    { ID_STRING, VOIDF(UIW_STRING::New), "STRING", 0 },
    { ID_SYSTEM_BUTTON, VOIDF(UIW_SYSTEM_BUTTON::New), "SYSTEM_BUTTON",
      0 },
    { ID_TEXT, VOIDF(UIW_TEXT::New), "TEXT", 0 },
    { ID_TITLE, VOIDF(UIW_TITLE::New), "TITLE", 0 },
    { ID_WINDOW, VOIDF(UIW_WINDOW::New), "WINDOW", 0 },
    { ID_END, NULL, NULL, 0 }
};
UI_ITEM *UI_WINDOW_OBJECT::objectTable = _objectTable;
```

This object table is the one generated when the window for DICTIONARY_WINDOW was created in the designer. The designer automatically creates an object table adding an entry for each type of object used. If you desire to create persistent objects without using the designer, you will need to create a similar object table.

When an object is read in, the object's type is loaded and checked against the object table to see which object is to be created. If the object's type is ID_WINDOW, for example, and there is an entry for it in the object table, the **UIW_WINDOW::New( )** will be called.

The Save function

The purpose of the **Save( )** function is to save the object into a file. Before each of the object's members are stored, an unsigned short containing the length of each member is

saved. This allows the correct number of bytes to be loaded when the object is later retrieved. The following listing shows how members and their lengths are stored:

```
void D_ENTRY::Save()
{
    // Store the word.
    object->Store((USHORT)ui_strlen(word));
    object->Store(word, 1, ui_strlen(word));

    // Store the word definition.
    object->Store((USHORT)ui_strlen(definition));
    object->Store(definition, 1, ui_strlen(definition));

    // Store the antonyms.
    object->Store((USHORT)ui_strlen(antonym));
    object->Store(antonym, 1, ui_strlen(antonym));

    // Store the synonyms.
    object->Store((USHORT)ui_strlen(synonym));
    object->Store(synonym, 1, ui_strlen(synonym));
}
```

When **Save( )** is called, **object->Store( )** is called to write the data to storage. UI_STORAGE actually writes the data to a temporary file and not to the actual data file. In order to "commit" the object to permanent storage, **UI_STORAGE::Save( )** must be called. In this program, this call is made in the destructor for the DICTIONARY class.

UI_STORAGE

The class DICTIONARY is derived from UI_STORAGE. The UI_STORAGE class is used to read or write Zinc Application Framework files. It is created as a class so that the file can be treated as an object, which handles file input and output.

The UI_STORAGE class can be thought of as a file system. Thus, one can make directories, change directories and add and delete "files" (i.e., resources) within the file. The main difference between a UI_STORAGE class and a regular file is that the UI_STORAGE file is constructed so that specific objects can be saved and retrieved. These objects can be persistent objects, and the user can store items or objects of different types to the file.

Conclusion

You should now be able to understand how to use the Zinc data file and how to add objects to it. You should also be able to use a window created in the designer within an application and add user functions to buttons on the window. Some enhancement ideas for the DICTIONARY program include creating multiple types of persistent objects for use in the same data file or using a third party database instead of the Zinc data file.

SECTION III
ZINC APPLICATION PROGRAM

CHAPTER 7 – GETTING THE RIGHT DESIGN

The next several tutorials are designed to help you understand Zinc design and coding features, which will help you to write efficient applications. The source code associated with this program is located in **ZINC\TUTOR\ZINCAPP**. It contains the following files:

ZINCAPP.CPP—This file contains the main program loop (i.e., **main( )** or **WinMain( )**).

ZINCAPP.HPP—This file contains the constant definitions for the display, window, event and help messages that are passed through the system when a pop-up item is selected from the main control window. In addition, this file contains the declarations for the ZINCAPP_WINDOW_MANAGER, CONTROL_WINDOW and EVENT_MONITOR classes.

CONTROL.CPP—This file contains the following member functions:

CONTROL_WINDOW::CONTROL_WINDOW( )
CONTROL_WINDOW::Event( )
CONTROL_WINDOW::Message( )
ZINCAPP_WINDOW_MANAGER::Event( )
ZINCAPP_WINDOW_MANAGER::ExitFunction( )

These functions are used to create the main control menu and to handle all main control throughout the application.

SUPPORT.CPP—This file contains the object table that must be compiled with the application since persistent window objects are to be used.

SUPPORT.DAT—This file is the binary data file created by the Designer. It contains the help context and persistent window object information.

SUPPORT.HPP—This file contains the help context constant information used to associate a help context with a window. It also contains the persistent object identification values entered as the *stringID* field for each object in the **.DAT** file.

DISPLAY.CPP—This file contains the **CONTROL_WINDOW::Option_Display( )** member function. This function is used to change the type of display used.

EVENT.CPP—This file contains the **CONTROL_WINDOW::Option_Event( )** and **EVENT_MONITOR( )** member functions. These functions are used to process all

the messages that are produced when an "Event" menu item is selected from the main control window.

HELP.CPP—This file contains the **CONTROL_WINDOW::OptionHelp( )** member function. It processes all of the messages that are produced when a "Help" menu item is selected from the main control window.

WINDOW.CPP—This file contains the **CONTROL_WINDOW::OptionWindow( )** member function. This function invokes the proper window that was selected from the main control window by processing all the messages that are produced when a menu item is selected.

.DEF**, ***.RC**—These files are the environment specific definition and resource files required when compiling for Windows or OS/2. (**NOTE:** The **W*. files are for Windows and the **O*.*** files are for OS/2.)

***.MAK**—These files are the compiler-dependent makefiles associated with the Zincapp program. (See "Chapter 1—Initializing the Library" for information on compiling for each Zinc-supported platform.)

Goals

The first step in designing an effective application—after you have identified your audience and the major objectives you want to achieve—is defining the high-level operation of your program.

The Zinc application program is designed with the following main areas of emphasis:

General control—The goal of this area is to provide a consistent easy-to-use program that will be familiar to users. This goal is accomplished by providing a consistent interface that conforms to the Common User Access (CUA) standards.

Screen features—The goal of this area is to show the flexible and versatile nature of the screen display. This goal is accomplished by letting users switch display modes from text to graphics, or vice versa during the application.

Window objects—The goal of this area is to show Zinc Application Framework as a complete user interface package. This goal is accomplished by showing the many different types of user interface objects that can be created using the library.

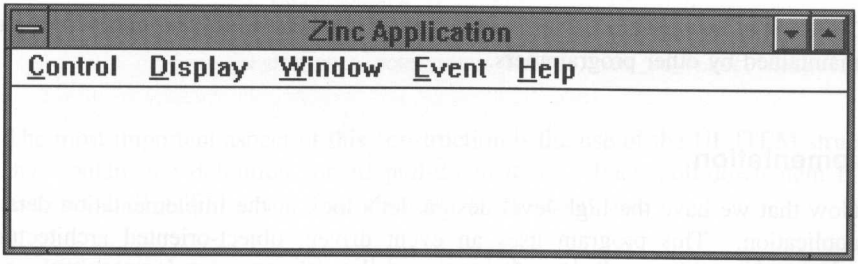
Event information—The goal of this area is to show the flexible nature of input information and the advanced event driven architecture. This goal is accomplished

by showing how input information is entered then processed by objects within the system.

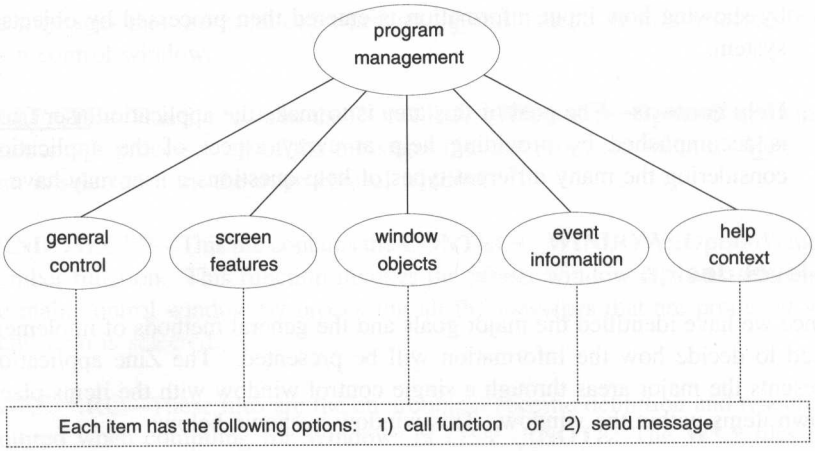
Help contexts—The goal of this area is to make the application user friendly. This is accomplished by providing help at every aspect of the application, and by considering the many different types of help questions a user may have.

High-level design

Once we have identified the major goals and the general methods of implementation, we need to decide how the information will be presented. The Zinc application program presents the major areas through a single control window with the items placed as pull-down items within the window. This window is shown below:



From a conceptual level, the main window serves as the control unit to all of the areas of emphasis we have identified by pull-down items. Each sub-module controls the operation of items within its scope. For example, the main control window may pass control to a screen features control unit. It, in turn, will either send a message through the system, requesting that some action be performed or pass control to some other function where the operation can be performed. The representation of this control can be shown by the figure below:



This model is very consistent, and, when implemented, will be easily understood and maintained by other programmers.

Implementation

Now that we have the high-level design, let's look at the implementation details of the application. This program uses an event driven, object-oriented architecture. The following provides a conceptual overview to how the program is organized:

1—The controlling window is created and is attached to the Window Manager. This window is a class object derived from the base UIW_WINDOW class. Its derivation from a window allows us to override the **Event( )** virtual function to determine what messages are being passed to the window and then lets us dispatch those messages in a clean fashion through class member functions (described later in this chapter).

```
class CONTROL_WINDOW : public UIW_WINDOW
{
public:
    CONTROL_WINDOW(void);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
```

The constructor is used to set up the window and pop-up menu items. A partial listing of this initialization is shown below:

```
CONTROL_WINDOW::CONTROL_WINDOW(void) : UIW_WINDOW(0, 0, 76, 6,
    WOF_NO_FLAGS, WOAF_LOCKED)
{
    // Control menu items.
    static UI_ITEM controlItems[] =
    {
        { S_REDISPLAY,          VOIDF(CONTROL_WINDOW::Message),
            "&Refresh\tShift+F6", MNIF_NO_FLAGS },
```

```
        { 0,    VOIDF(0),    "",          MNIF_NO_FLAGS },// item separator
        { L_EXIT_FUNCTION,       VOIDF(CONTROL_WINDOW::Message),
          "E&xit\tAlt+F4",       MNIF_NO_FLAGS },
        { 0, 0, 0, 0 }  // End of array.
    };
    .
    .
    .

    // Attach the sub-window objects to the control window.
    *this
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON(SYF_GENERIC)
        + new UIW_TITLE("Zinc Application")
        + &(*new UIW_PULL_DOWN_MENU
            + new UIW_PULL_DOWN_ITEM("&Control", WNF_NO_FLAGS,
                controlItems)
            + new UIW_PULL_DOWN_ITEM("&Display", WNF_NO_FLAGS,
                displayItems)
            + &(*new UIW_PULL_DOWN_ITEM("&Window", WNF_NO_FLAGS)
                + controlObjects
                + inputObjects
                + selectObjects)
            + new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS, eventItems)
            + new UIW_PULL_DOWN_ITEM("&Help", WNF_NO_FLAGS, helpItems));
}
```

The most important aspect of this construction is the use of the UI_ITEM structures that contain the definition for all pull-down items. Each pull-down item has an associated UI_ITEM array. The elements of this array are:

- the internal message that will be passed through the system. This is the first field in the UI_ITEM structure. For example, the first Control menu item (Clear) contains the message MSG_CLEAR. This message will be passed through the system whenever the Control | Clear menu item is selected.

- the static member function that is called when the user selects a menu item. All menu items specify **CONTROL_WINDOW::Message( )** as their user-function. This function is responsible for the actual dispatching of messages via the Event Manager.

- the string information to be displayed on the screen. In the case of the Clear menu item this string is "&Clear\tShift+F5". (The "Shift+F5" portion of the string is discussed later in this chapter.)

- the menu item flags. These flags control the presentation and interaction of the menu item. For example, MNIF_CHECK_MARK will cause a check mark character to be displayed to the left of the menu item's text when the item is selected.

2—The Window Manager dispatches all messages to the front window (in our case the main control window). When the main control window receives input, it dispatches the information according to the logical type.

The class members responsible for the first sub-level of control are shown below:

```
class CONTROL_WINDOW : public UIW_WINDOW
{
protected:
    void OptionDisplay(EVENT_TYPE item);
    void OptionEvent(EVENT_TYPE item);
    void OptionHelp(EVENT_TYPE item);
    void OptionWindow(EVENT_TYPE item);
};
```

In our application there are four types of messages that can be received:

Display option messages—These types of messages are generated when a "Display" menu item has been selected from the main control window. They are processed by the **OptionDisplay( )** member function.

Window option messages—These types of messages are generated when a "Window" menu item has been selected from the main control window. They are processed by the **OptionWindow( )** member function.

Event option messages—These types of messages are generated when an "Event" menu item has been selected from the main control window. They are processed by the **OptionEvent( )** member function.

Help option messages—These types of messages are generated when a "Help" menu item has been selected from the main control window. They are processed by the **OptionHelp( )** member function.

All other messages are passed to the **UIW_WINDOW::Event( )** member function for processing.

NOTE: The control option messages are automatically processed by the Window Manager since they represent operations handled by the Window Manager.

3—When an option member function is selected, it has the option of either sending a message back through the system or of calling another member function that is appropriate based on the type of message. For example, the "Display" control function (**OptionDisplay**) sends a message through the system rather than re-setting the display itself:

```
#if defined (ZIL_MSWINDOWS) || defined (ZIL_OS2) || defined (ZIL_MOTIF)
void CONTROL_WINDOW::OptionDisplay(EVENT_TYPE item)
{
}
#else
void CONTROL_WINDOW::OptionDisplay(EVENT_TYPE item)
{
    // Set up the default (i.e., graphics mode) event.
    UI_EVENT event(S_RESET_DISPLAY, TDM_NONE);

    // Decide on the new display type.
    if (item == MSG_25x40_MODE)
        event.rawCode = TDM_25x40;
    else if (item == MSG_25x80_MODE)
        event.rawCode = TDM_25x80;
    else if (item == MSG_43x80_MODE)
        event.rawCode = TDM_43x80;

    // Send a message to reset the display.
    // (Code resides in main program loop).
    eventManager->Put(event);
}
#endif
```

The "Event" control function (**OptionEvent**), on the other hand, creates an event monitor class object and attaches it directly to the Window Manager. No additional messaging is required.

The implementation details of each menu item is given in the next five tutorial chapters. These chapters are organized in the following manner:

"Chapter 8—Control Options" contains information about program flow when one of the "Control" menu items is selected from the main control window.

"Chapter 9—Display Options" contains information about program flow when one of the "Display" menu items is selected from the main control window.

"Chapter 10—Window Options" contains information about program flow when one of the "Window" menu items is selected from the main control window.

"Chapter 11—Event Options" contains information about program flow when one of the "Event" menu items is selected from the main control window.

"Chapter 12—Help Options" contains information about program flow when one of the "Help" menu items is selected from the main control window.

The remaining parts of this chapter address the implementation of accelerator keys and a brief discussion of how structured programming is often used with Zinc Application Framework.

Accelerator keys

There are two accelerator keys defined for this program:

<Shift+F6>—Pressing this key combination causes the Window Manager to clear the screen and to redisplay each window that is attached to the Window Manager's list of window objects.

<Alt+F4>—Pressing this key combination causes the exit application window to appear on the screen.

The accelerator keys are implemented in the **CONTROL_WINDOW::Event( )** function.

```
EVENT_TYPE CONTROL_WINDOW::Event(const UI_EVENT &event)
{
    // Check for an accelerator key.
    EVENT_TYPE ccode = event.type;
    if (ccode == L_EXIT_FUNCTION)
        eventManager->Put(UI_EVENT(L_EXIT_FUNCTION));

    if (ccode == E_KEY)
    {
        // Define the set of accelerator keys.
        static struct ACCELERATOR_PAIR
        {
            RAW_CODE rawCode;
            LOGICAL_EVENT logicalType;
        } acceleratorTable[] =
        {
            { SHIFT_F6,     S_REDISPLAY },
            { ALT_F4,       L_EXIT_FUNCTION },
            { 0, 0 }    // End of array.
        };

        for (int i = 0; acceleratorTable[i].rawCode; i++)
            if (event.rawCode == acceleratorTable[i].rawCode)
            {
                UI_EVENT tEvent(acceleratorTable[i].logicalType);
                eventManager->Put(tEvent);  // Put the accelerator key
                return (ccode);             // into the system.
            }
    }

    // Process the event according to its type.
    if (ccode >= MSG_HELP)
        OptionHelp(event.type);                 // Help menu option selected.
    else if (ccode >= MSG_EVENT)
        OptionEvent(event.type);                // Event menu option selected.
    else if (ccode >= MSG_WINDOW)
        OptionWindow(event.type);               // Window menu option selected.
    else if (ccode >= MSG_DISPLAY)
        OptionDisplay(event.type);              // Display menu option selected.

    else if (ccode >= MSG_CONTROL)
    {
            UI_EVENT tEvent(event.type);
            eventManager->Put(tEvent);          // Put the accelerator key
    }
    else
        ccode = UIW_WINDOW::Event(event);       // Unknown event.
```

```
    // Return the control code.
    return (ccode);
}
```

This implementation is described by the following steps:

1—The **Event( )** function receives all input from the Window Manager.

2—If the event is a normal key the control window searches its list of raw-code/logical type pairs. The definition of the two accelerator keys is given by the *acceleratorTable* static array (shown above).

3—If an accelerator key is detected, its logical value is placed into the Event Manager. This value is later interpreted by the Window Manager, when the main program loop gets the next key using **eventManager->Get( )**.

NOTE: The accelerator keys described above are only available when the main control window is at the front of the screen.

Structured programming

Quite often, structured programming techniques are used to program with Zinc Application Framework. If this program were re-written to incorporate this type of programming, each menu item could be assigned a function that was executed when the item was selected. Here is some sample code that shows how the "Display, Help" options specified in the CONTROL_WINDOW constructor could be re-written to call specific help functions rather than calling a message passing function, as is currently employed. (This code is not contained in any of the ZincApp program files. It is presented as a conceptual alternative.)

```
CONTROL_WINDOW::CONTROL_WINDOW(void) :
    UIW_WINDOW(0, 0, 52, 13, WOF_NO_FLAGS, WOAF_LOCKED)
{
    extern EVENT_TYPE HelpKeyboard(UI_WINDOW_OBJECT *item, UI_EVENT &event,
        EVENT_TYPE ccode);
    extern EVENT_TYPE HelpMouse(UI_WINDOW_OBJECT *item, UI_EVENT &event,
        EVENT_TYPE ccode);
    extern EVENT_TYPE HelpCommands(UI_WINDOW_OBJECT *item, UI_EVENT &event,
        EVENT_TYPE ccode);
    extern EVENT_TYPE HelpProcedures(UI_WINDOW_OBJECT *item,
        UI_EVENT &event, EVENT_TYPE ccode);
    extern EVENT_TYPE HelpHelp(UI_WINDOW_OBJECT *item, UI_EVENT &event,
        EVENT_TYPE ccode);
    extern EVENT_TYPE HelpZincApp(UI_WINDOW_OBJECT *item, UI_EVENT &event,
        EVENT_TYPE ccode);

    static UI_ITEM helpItems[] =
    {
        { MSG_HELP_KEYBOARD, VOIDF(CONTROL_WINDOW::Message), "&Keyboard",
            MNIF_NO_FLAGS },
```

```
            { MSG_HELP_MOUSE, VOIDF(CONTROL_WINDOW::Message), "&Mouse",
              MNIF_NO_FLAGS },
            { MSG_HELP_COMMANDS, VOIDF(CONTROL_WINDOW::Message),"&Commands",
              MNIF_NO_FLAGS },
            { MSG_HELP_PROCEDURES, VOIDF(CONTROL_WINDOW::Message),
              "&Procedures", MNIF_NO_FLAGS },
            { MSG_HELP_OBJECTS, VOIDF(CONTROL_WINDOW::Message), "&Objects",
              MNIF_NO_FLAGS },
            { MSG_HELP_HELP, VOIDF(CONTROL_WINDOW::Message),
              "&Using help", MNIF_NO_FLAGS },
            { 0, VOIDF(0), "", MNIF_SEPARATOR },
            { MSG_HELP_ZINCAPP, VOIDF(About),
              "&About ...", MNIF_NO_FLAGS },
            { 0, 0, 0, 0 }  // End of array.
        };
        .
        .
        .
    }
```

You can see how each item could have an associated function that performed a particular operation based on the type of menu item that was selected. To implement this design throughout the program, we would need to define functions for each of the menu items specified in the main control window. Here is an example of how the **HelpKeyboard( )** function might be implemented:

```
EVENT_TYPE HelpKeyboard(UI_WINDOW_OBJECT *item, UI_EVENT &event,
    EVENT_TYPE ccode)
{
    item->helpSystem->DisplayHelp(item->windowManager, HELP_KEYBOARD);
}
```

While this method of implementation works, it has several drawbacks:

1—It results in duplicate definitions and operations. You can see from the help example above that it would take seven functions to do the work the **CONTROL_-WINDOW::OptionHelp( )** function did. This wastes compiler time and executable space.

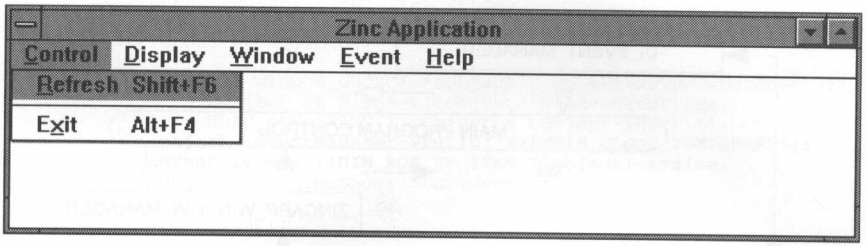
2—It forces you back into a structured method of programming. Learning an event driven architecture takes time. It can become very confusing if the application you write contains elements of an event driven system and elements of structured programming methods.

3—It doubles the effort of Zinc Application Framework. Since Zinc is based on an event driven architecture, a structured functions approach implements a second type of design architecture. This increases the amount of time and effort involved in creating and debugging your applications.

There are many advantages to the object-oriented, event driven architecture employed by Zinc Application Framework. As you work with the library, you will begin to see how these features combine to make a powerful, consistent library architecture.

CHAPTER 8 – CONTROL OPTIONS

The ZincApp program's control options are shown under the "Control" menu item:

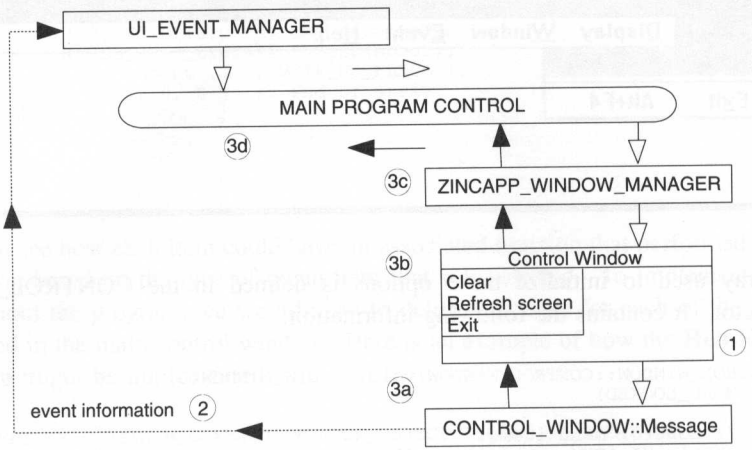


The array used to initialize these options is defined in the CONTROL_WINDOW constructor. It contains the following information:

```
CONTROL_WINDOW::CONTROL_WINDOW(void) : UIW_WINDOW(0, 0, 76, 6, WOF_NO_FLAGS,
    WOAF_LOCKED)
{
    // Control menu items.
    static UI_ITEM controlItems[] =
    {
        { S_REDISPLAY,      VOIDF(Message), "&Refresh\tShift+F6",
          MNIF_NO_FLAGS },
        { 0,                VOIDF(0),       "", 0 },     // item separator
        { L_EXIT_FUNCTION,  VOIDF(Message), "E&xit\tAlt+F4",
          MNIF_NO_FLAGS },
        { 0, 0, 0, 0 }  // End of array.
    };
    .
    .
    .

    // Attach the sub-window objects to the control window.
    *this
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON(SYF_GENERIC)
        + new UIW_TITLE("Zinc Application")
        + &(*new UIW_PULL_DOWN_MENU
          + new UIW_PULL_DOWN_ITEM("&Control", WNF_NO_FLAGS,
            controlItems)
          + new UIW_PULL_DOWN_ITEM("&Display", WNF_NO_FLAGS,
            displayItems)
          + &(*new UIW_PULL_DOWN_ITEM("&Window", WNF_NO_FLAGS)
            + controlObjects
            + inputObjects
            + selectObjects)
          + new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS, eventItems)
          + new UIW_PULL_DOWN_ITEM("&Help", WNF_NO_FLAGS, helpItems));
}
```

Control program flow

When a control option is selected, it is handled in five major steps. A complete explanation of these steps follows (the corresponding steps are shown by the circled numbers in the figure):



1—The user function, **CONTROL_WINDOW::Message( )**, is called by the **UIW_POP_UP_ITEM::Event( )** function. (The pop-up item inherits the code below from the UIW_BUTTON class.)

```
EVENT_TYPE UIW_BUTTON::Event(const UI_EVENT &event)
{
    .
    .
    .
    case L_SELECT:
    case L_END_SELECT:
        UI_EVENT tEvent;
        tEvent = event;
        if (userFunction)
            (*userFunction)(this, tEvent, ccode);
```

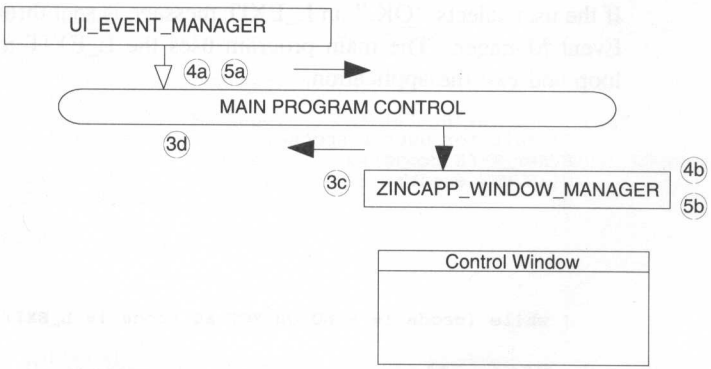
The arguments passed to **Message( )** are a pointer to the selected control option (*this*), a copy of the event that caused the user function to be called (*tEvent*), and the logical interpretation (*ccode*) of the event that caused **Event( )** to be called. (**NOTE:** the variable *tEvent* needs to be a copy of *event* since *event* is a constant variable whose values cannot be modified.)

2—The **CONTROL_WINDOW::Message( )** function sends a request to remove the temporary control options menu by sending an S_CLOSE_TEMPORARY message through the system via the Event Manager. It then sends the control request through

the system by setting *event.type* to be the menu item's value (i.e., the S_REDISPLAY or L_EXIT values defined in the *controlOptions* array) and then by sending another message through the system.

```
EVENT_TYPE CONTROL_WINDOW::Message(UI_WINDOW_OBJECT *data,
    UI_EVENT &event, EVENT_TYPE ccode)
{
    if (ccode == L_SELECT)
    {
        for (UI_WINDOW_OBJECT *tObject = object->windowManager->First();
            tObject && FlagSet(tObject->woAdvancedFlags,
            WOAF_TEMPORARY); tObject = tObject->Next());
        object->eventManager->Put(UI_EVENT(S_CLOSE_TEMPORARY));
        event.type = ((UIW_POP_UP_ITEM *)object)->value;
        object->eventManager->Put(event);
    }
    return (ccode);
}
```

3—Control returns to the main loop by first exiting **CONTROL_WINDOW::-Message( )** and then by exiting the UIW_POP_UP_ITEM, CONTROL_WINDOW and ZINCAPP_WINDOW_MANAGER classes' **Event( )** virtual functions.

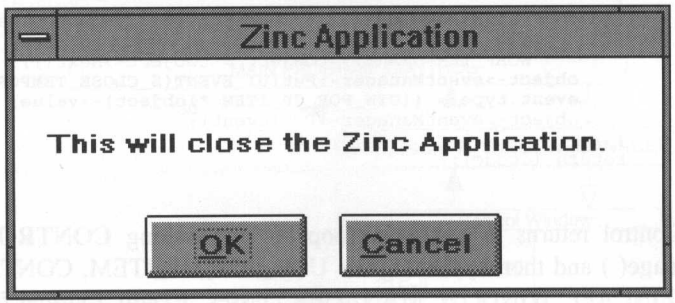


4—The main loop picks up the program generated messages by calling **event-Manager->Get( )**. The first message received is S_CLOSE_TEMPORARY. This message is handled by the Window Manager and causes the control options menu to be removed from the screen.

5—The second message received is the control message determined by the selected menu item. This message is passed to the Window Manager by calling **window-Manager->Event( )**. The Window Manager performs the following actions according to the type of message:

S_REDISPLAY—Causes the Window Manager to clear the screen and redisplay each window that is attached to the Window Manager's list of window objects.

L_EXIT_FUNCTION—The Window Manager calls the **CONTROL_WINDOW::ExitFunction( )** function which displays an exit window on the screen. A picture of this window is shown below:



If the user selects "OK," an L_EXIT message is sent through the system via the Event Manager. The main program uses the L_EXIT to break from its main loop and exit the application.

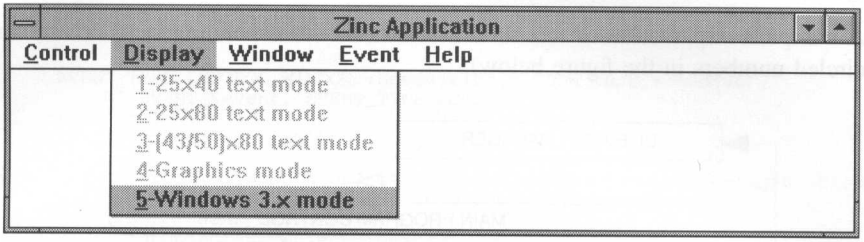
```
// Wait for user response.
EVENT_TYPE ccode;
UI_EVENT event;
do
{
    .
    .
    .
} while (ccode != S_NO_OBJECT && ccode != L_EXIT);
```

Since the Window Manager recognizes and processes all of these messages, no control is passed to the control window; rather, program flow returns to the main loop.

The most interesting part of the flow information discussed above is how **CONTROL_WINDOW::Message( )** generates an event that is later interpreted by the Window Manager and that the message requires no special handling on the application's part. This control works correctly because the events are passed through the system via the Event Manager.

CHAPTER 9 – DISPLAY OPTIONS

The ZincApp program's display options are shown under the "Display" menu item:



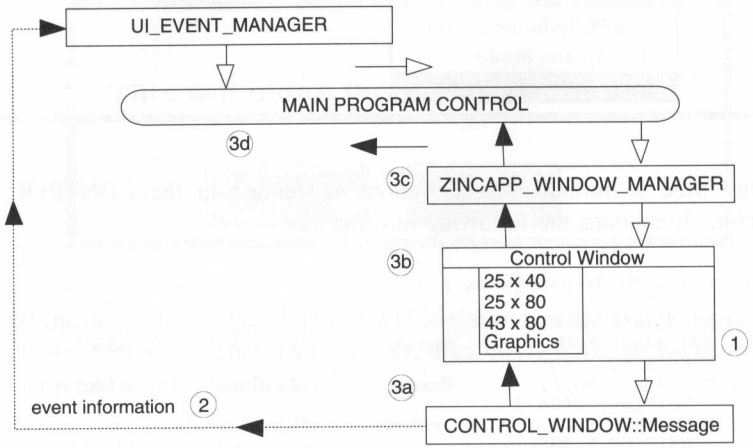
The array used to initialize these options is defined in the CONTROL_WINDOW constructor. It contains the following information:

```
static UI_ITEM displayItems[] =
{
#if defined (ZIL_MSDOS)
    { MSG_25x40_MODE,       Message,    "&1-25x40 text mode",
        MNIF_NO_FLAGS },
    { MSG_25x80_MODE,       Message,    "&2-25x80 text mode",
        MNIF_NO_FLAGS },
    { MSG_43x80_MODE,       Message,    "&3-(43/50)x80 text mode",
        MNIF_NO_FLAGS },
    { MSG_GRAPHICS_MODE,    Message,    "&4-Graphics mode", MNIF_NO_FLAGS },
    { MSG_WINDOWS_MODE,     Message,    "&5-Windows 3.X mode",
        MNIF_NON_SELECTABLE },
#endif
    { 0, 0, 0, 0 }  // End of array.
};
.
.
.

// Attach the sub-window objects to the control window.
*this
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON(SYF_GENERIC)
    + new UIW_TITLE("Zinc Application")
    + &(*new UIW_PULL_DOWN_MENU
        + new UIW_PULL_DOWN_ITEM("&Control", WNF_NO_FLAGS, controlItems)
        + new UIW_PULL_DOWN_ITEM("&Display", WNF_NO_FLAGS, displayItems)
        + &(*new UIW_PULL_DOWN_ITEM("&Window", WNF_NO_FLAGS)
            + controlObjects
            + inputObjects
            + selectObjects)
        + new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS, eventItems)
        + new UIW_PULL_DOWN_ITEM("&Help", WNF_NO_FLAGS, helpItems));
```

Display program flow

When a display option is selected, initial program flow is handled the same way that the control options are handled. At the fifth step however, program flow is directed to the **OptionsDisplay( )** member function.

A complete explanation of this flow follows (the corresponding steps are shown by the circled numbers in the figure below):



1—The **CONTROL_WINDOW::Message( )** function is called by the **UIW_POP_-UP_ITEM::Event( )** function. (The pop-up item inherits the code below from the UIW_BUTTON class.)

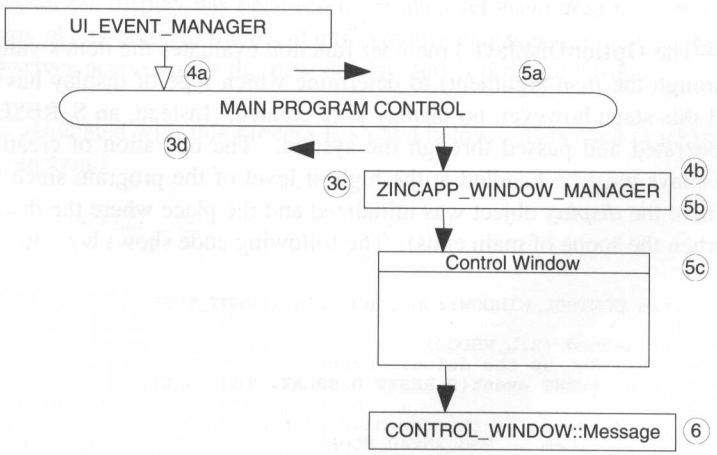
```
EVENT_TYPE UIW_BUTTON::Event(const UI_EVENT &event)
{
    .
    .
    .
    case L_SELECT:
    case L_END_SELECT:
        UI_EVENT tEvent = event;
        if (userFunction)
            (*userFunction)(this, tEvent, ccode);
```

The arguments passed to **Message( )** are a pointer to the selected display option (*this*), a copy of the event that caused the user function to be called (*tEvent*), and the logical interpretation (*ccode*) of the event that caused **Event( )** to be called. (**NOTE:** the variable *tEvent* needs to be a copy of *event* since *event* is a constant variable whose values cannot be modified.)

2—The **CONTROL_WINDOW::Message( )** function sends a request to remove the temporary display options menu by sending an S_CLOSE_TEMPORARY message through the system via the Event Manager. It then sends the display request through the system by setting *event.type* to be the menu item's value (i.e., one of the MSG_DISPLAY values defined in the *displayOptions* array) and sending this message through the system.

```
EVENT_TYPE CONTROL_WINDOW::Message(UI_WINDOW_OBJECT *object,
    UI_EVENT &event, EVENT_TYPE ccode)
{
    if (ccode == L_SELECT)
    {
        for (UI_WINDOW_OBJECT *tObject = object->windowManager->First();
            tObject && FlagSet(tObject->woAdvancedFlags,
            WOAF_TEMPORARY);
        tObject = tObject->Next())
        object->eventManager->Put(UI_EVENT(S_CLOSE_TEMPORARY));
        event.type = ((UIW_POP_UP_ITEM *)object)->value;
        object->eventManager->Put(event);
    }
    return (ccode);
}
```

3—Control returns to the main loop by first exiting **CONTROL_WINDOW::-Message( )** and then by exiting the UIW_POP_UP_ITEM, CONTROL_WINDOW and ZINCAPP_WINDOW_MANAGER classes' **Event( )** virtual functions.



4—The main loop picks up the program generated messages by calling **event-Manager->Get( )**. The first message received is S_CLOSE_TEMPORARY. This message is handled by the Window Manager and causes the display options menu to be removed from the screen.

5—The second message received is the display message determined by the selected menu item. This message is passed by the main loop to the Window Manager, then is dispatched by the Window Manager to **CONTROL_WINDOW::Event( )** since the control window is the front window on the screen. The control window evaluates *event.type* (in this case a MSG_DISPLAY message)—resulting in the **OptionDisplay( )** member function being called.

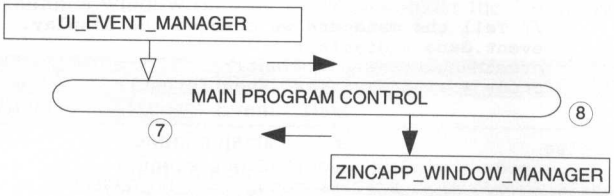
The code responsible for this control is shown below:

```
EVENT_TYPE CONTROL_WINDOW::Event(const UI_EVENT &event)
{
    .
    .
    .
    // Process the event according to its type.
    if (ccode >= MSG_HELP)
        OptionHelp(event.type);          // Help menu option selected.
    else if (ccode >= MSG_EVENT)
        OptionEvent(event.type);         // Event menu option selected.
    else if (ccode >= MSG_WINDOW)
        OptionWindow(event.type);        // Window menu option selected.
    else if (ccode >= MSG_DISPLAY)
        OptionDisplay(event.type);       // Display menu option selected.
    else
        ccode = UIW_WINDOW::Event(event);    // Unknown event.

    // Return the control code.
    return (ccode);
}
```

6—The **OptionDisplay( )** member function evaluates the item's value (passed down through the *item* argument) to determine which type of display has been requested. At this stage however, no display is re-created. Instead, an S_RESET_DISPLAY is generated and passed through the system. The operation of creating and deleting displays must be handled at the highest level of the program since that is the place where the *display* object was initialized and the place where the display is destroyed (when the scope of main ends). The following code shows how this message is sent:

```
void CONTROL_WINDOW::OptionDisplay(EVENT_TYPE item)
{
#if defined (ZIL_MSDOS)
    // Set up the default event.
    UI_EVENT event(S_RESET_DISPLAY, TDM_NONE);

    // Decide on the new display type.
    if (item == MSG_25x40_MODE)
        event.rawCode = TDM_25x40;
    else if (item == MSG_25x80_MODE)
        event.rawCode = TDM_25x80;
    else if (item == MSG_43x80_MODE)
        event.rawCode = TDM_43x80;

    // Send a message to reset the display.
    // (Code resides in main program loop).
    eventManager->Put(event);
#endif
}
```

7—Control returns once again to the main program loop by exiting the associated **Event( )** functions (see step 3).



8—The main loop picks up the S_RESET_DISPLAY message by calling **eventManager->Get( )**. This message causes the program to:

A—Tell the event and window managers that the old display is about to be deleted. This allows the managers to un-initialize any display dependent information they may have.

B—The new display is constructed. The type of display is determined by *event.rawCode*.

C—After the display has been reset, we must set *event.data* to point to the new display object and call the event and window managers so they can re-initialize themselves according to the new display and coordinate system.

The code associated with this process is shown below. (This code is taken from the **main( )** function.)

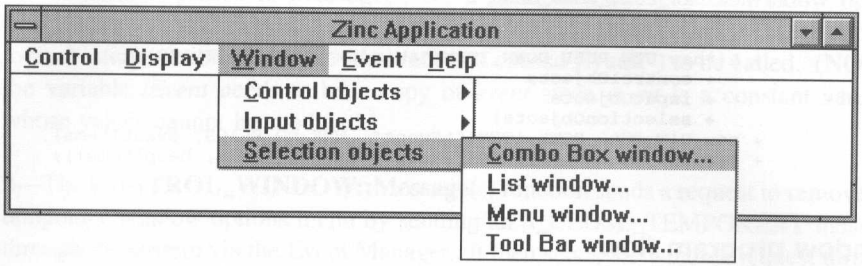
```
    // Wait for user response.
    EVENT_TYPE ccode;
    UI_EVENT event;
    do
    {
        // Get input from the user.
        eventManager->Get(event);

        // Check for a screen reset message.
        if (event.type == S_RESET_DISPLAY)
        {
#if defined(ZIL_MSDOS)
            event.data = NULL;

            // Tell the managers we changed the display.
            windowManager->Event(event);
            eventManager->Event(event);

            delete display;
            display = new UI_GRAPHICS_DISPLAY;
```

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

            // Tell the managers we changed the display.
            event.data = display;
            eventManager->Event(event);
            ccode = windowManager->Event(event);
#endif
        }
        else
            ccode = windowManager->Event(event);
    } while (ccode != L_EXIT && ccode != S_NO_OBJECT);
```

If you carefully examine the **CONTROL_WINDOW::OptionDisplay( )** member function and the code in the main program loop, you may recognize that we could have removed the **OptionDisplay( )** function if we were to intercept all MSG_DISPLAY messages in the main loop. The reason we did not put the display code in the main loop is mainly an issue of consistency. Up until this point, we have let the control window and associated member functions handle the program specific messages. In this case we are generating a system message from the display member function, then intercepting the request at the main level before letting the Window Manager process it.

CHAPTER 10 – WINDOW OPTIONS

The ZincApp program's window options are shown under the "Window" menu item:



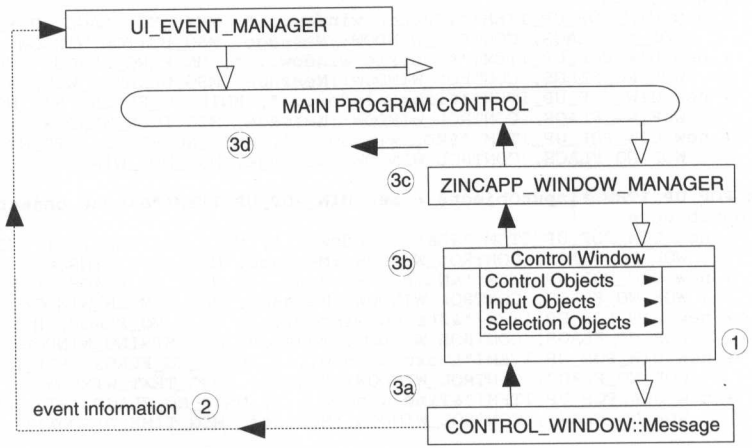
The array used to initialize these options is defined in the CONTROL_WINDOW constructor. It contains the following information:

```
// Create the objects submenu.
UIW_POP_UP_ITEM *controlObjects = new UIW_POP_UP_ITEM("&Control objects");
*controlObjects
    + new UIW_POP_UP_ITEM("&Button window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
        WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_BUTTON_WINDOW)
    + new UIW_POP_UP_ITEM("&Generic window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
        WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_GENERIC_WINDOW)
    + new UIW_POP_UP_ITEM("&Icon window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
        WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_ICON_WINDOW)
    + new UIW_POP_UP_ITEM("&MDI window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
        WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_MDI_WINDOW);

UIW_POP_UP_ITEM *inputObjects = new UIW_POP_UP_ITEM("&Input objects");
*inputObjects
    + new UIW_POP_UP_ITEM("&Date window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
        WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_DATE_WINDOW)
    + new UIW_POP_UP_ITEM("&Number window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
        WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_NUMBER_WINDOW)
    + new UIW_POP_UP_ITEM("&String window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
        WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_STRING_WINDOW)
    + new UIW_POP_UP_ITEM("&Text window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
        WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_TEXT_WINDOW)
    + new UIW_POP_UP_ITEM("&Time window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
        WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_TIME_WINDOW);

UIW_POP_UP_ITEM *selectObjects = new UIW_POP_UP_ITEM("&Selection objects");
*selectObjects
    + new UIW_POP_UP_ITEM("&Combo Box window...", MNIF_NO_FLAGS,
        BTF_NO_FLAGS, WOF_NO_FLAGS, CONTROL_WINDOW::Message,
        MSG_COMBO_BOX_WINDOW)
    + new UIW_POP_UP_ITEM("&List window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
        WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_LIST_WINDOW)
    + new UIW_POP_UP_ITEM("&Menu window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
        WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_MENU_WINDOW)
    + new UIW_POP_UP_ITEM("&Tool Bar window...", MNIF_NO_FLAGS,
        BTF_NO_FLAGS, WOF_NO_FLAGS, CONTROL_WINDOW::Message,
        MSG_TOOL_BAR_WINDOW);
.
.
.
```

```
// Attach the sub-window objects to the control window.
*this
+ new UIW_BORDER
+ new UIW_MAXIMIZE_BUTTON
+ new UIW_MINIMIZE_BUTTON
+ new UIW_SYSTEM_BUTTON(SYF_GENERIC)
+ new UIW_TITLE("Zinc Application")
+ &(*new UIW_PULL_DOWN_MENU
    + new UIW_PULL_DOWN_ITEM("&Control", WNF_NO_FLAGS, controlItems)
    + new UIW_PULL_DOWN_ITEM("&Display", WNF_NO_FLAGS, displayItems)
    + &(*new UIW_PULL_DOWN_ITEM("&Window", WNF_NO_FLAGS)
        + controlObjects
        + inputObjects
        + selectionObjects)
    + new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS, eventItems)
    + new UIW_PULL_DOWN_ITEM("&Help", WNF_NO_FLAGS, helpItems));
```

Window program flow

When a window option is selected, initial program flow is handled the same way that the display options are handled. At the fifth step however, program flow is directed to the **OptionWindow( )** member function. A complete explanation of this flow follows. (The corresponding steps are shown by the circled numbers in the figure.)



1—The **CONTROL_WINDOW::Message( )** function is called by **UIW_POP_UP_-ITEM::Event( )**. (The pop-up item inherits this calling sequence from the UIW_-BUTTON class.)

```
EVENT_TYPE UIW_BUTTON::Event(const UI_EVENT &event)
{
    .
    .
    .
```

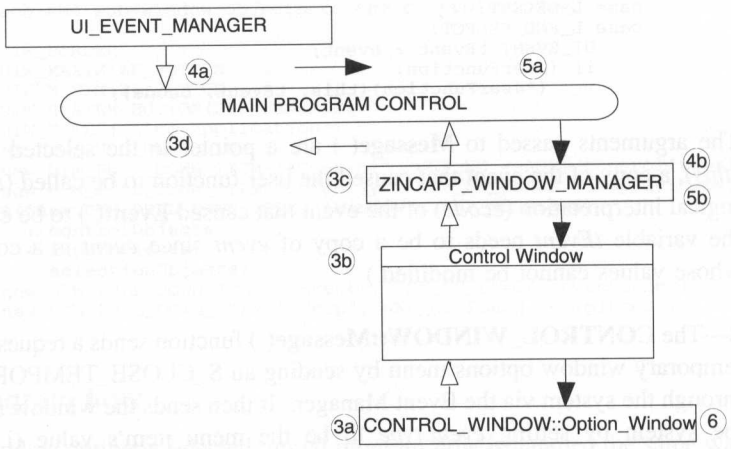
```
        case L_SELECT:
        case L_END_SELECT:
            UI_EVENT tEvent = event;
            if (userFunction)
                (*userFunction)(this, tEvent, ccode);
```

The arguments passed to **Message( )** are a pointer to the selected window option (*this*), a copy of the event that caused the user function to be called (*tEvent*), and the logical interpretation (*ccode*) of the event that caused **Event( )** to be called. (**NOTE:** the variable *tEvent* needs to be a copy of *event* since *event* is a constant variable whose values cannot be modified.)

2—The **CONTROL_WINDOW::Message( )** function sends a request to remove the temporary window options menu by sending an S_CLOSE_TEMPORARY message through the system via the Event Manager. It then sends the window request through the system by setting *event.type* to be the menu item's value (i.e., one of the ZINCAPP_WINDOW values defined in the *windowOptions* array) and then by sending another message through the system.

```
EVENT_TYPE CONTROL_WINDOW::Message(UI_WINDOW_OBJECT *data,
    UI_EVENT &event, EVENT_TYPE ccode)
{
    if (ccode == L_SELECT)
    {
        for (UI_WINDOW_OBJECT *tObject = object->windowManager->First();
             tObject && FlagSet(tObject->woAdvancedFlags,
             WOAF_TEMPORARY);
        tObject = tObject->Next())
        object->eventManager->Put(UI_EVENT(S_CLOSE_TEMPORARY));
        event.type = ((UIW_POP_UP_ITEM *)object)->value;
        object->eventManager->Put(event);
    }
    return (ccode);
}
```

3—Control returns to the main loop by first exiting **CONTROL_WINDOW::-Message( )** and then by exiting the UIW_POP_UP_ITEM, CONTROL_WINDOW and ZINCAPP_EVENT_MANAGER classes' **Event( )** virtual functions.



4—The main loop picks up the program generated messages by calling **eventManager->Get( )**. The first message received is S_CLOSE_TEMPORARY. This message is handled by the Window Manager and causes the window options menu to be removed from the screen.

5—The second message received is the window request determined by the selected menu item. This message is passed by the main loop to the Window Manager and is then dispatched by the Window Manager to **CONTROL_WINDOW::Event( )** since the control window is the front window on the screen. The control window evaluates *event.type* (in this case a MSG_WINDOW message)—resulting in the **OptionWindow( )** member function being called.

The code responsible for this control is shown below:

```
EVENT_TYPE CONTROL_WINDOW::Event(const UI_EVENT &event)
{
    .
    .
    .

    // Process the event according to its type.
    if (ccode >= MSG_HELP)
        OptionHelp(event.type);          // Help menu option selected.
    else if (ccode >= MSG_EVENT)
        OptionEvent(event.type);         // Event menu option selected.
    else if (ccode >= MSG_WINDOW)
        OptionWindow(event.type);        // Window menu option selected.
    else if (ccode >= MSG_DISPLAY)
        OptionDisplay(event.type);       // Display menu option selected.
    else
        ccode = UIW_WINDOW::Event(event);    // Unknown event.

    // Return the control code.
    return (ccode);
}
```

6—The **OptionWindow( )** member function evaluates the item's value (passed down through the *item* argument) to determine which type of window has been requested. It then calls the associated member function that constructs the window. The new window is attached to the Window Manager using the + operator overload. The following code shows how this is done:

```
void CONTROL_WINDOW::OptionWindow(EVENT_TYPE item)
{
    // Get the specified window.
    UI_WINDOW_OBJECT *object = NULL;
    switch(item)
    {
    case MSG_DATE_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_DATE");
        break;

    case MSG_GENERIC_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_GENERIC");
        break;

    case MSG_ICON_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_ICON");
        break;

    case MSG_LIST_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_LIST");
        break;

    case MSG_COMBO_BOX_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_COMBO_BOX");
        break;

    case MSG_MENU_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_MENU");
        break;

    case MSG_NUMBER_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_NUMBER");
        break;

    case MSG_STRING_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_STRING");
        break;

    case MSG_TEXT_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_TEXT");
        break;

    case MSG_TIME_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_TIME");
        break;

    case MSG_BUTTON_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_BUTTON");
        break;

    case MSG_TOOL_BAR_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_TOOL_BAR");
        break;

    case MSG_MDI_WINDOW:
        object = UIW_WINDOW::New("support.dat~WINDOW_MDI");
        break;
    }
```

```
        // Add the window object to the window manager.
        if (object)
            *windowManager + object;
    }
```

You may have noticed that the *object* variable is defined to be a UI_WINDOW_-OBJECT pointer instead of a UIW_WINDOW pointer. This generic declaration allows us to expand the program to attach other non-window objects (e.g., an icon).

At this point the new window is displayed on the screen and it becomes the front window of the application. All subsequent events are processed by the new window until a change is requested by the end-user. A description of the types of windows presented in this menu item follows:

Generic—This window shows the basic window objects that are usually provided as default objects to a window. These objects include:

- the window's border (UIW_BORDER),
- the maximize button (UIW_MAXIMIZE_BUTTON),
- the minimize button (UIW_MINIMIZE_BUTTON),
- the system button (UIW_SYSTEM_BUTTON) and
- the title bar (UIW_TITLE).

In this function, the window is created by loading it from the **.DAT** file. If the window were not loaded from the **.DAT** file, it could have been created by calling **UIW_WINDOW::Generic( )**.

Button—This window shows the different types of buttons that can be used: regular buttons, radio buttons, check boxes and bitmapped buttons.

Combo box—*This window shows two combo box objects. One of the combo boxes* was implemented with string objects and the other with bitmapped buttons.

Date—This window shows the many variations available with the date class object.

Icon—This window shows several types of icon images that can either be attached to a parent window, or directly to the screen.

List—This window shows the implementation of both a horizontal and a vertical list.

MDI window—This window was implemented as an MDI parent window that contains some MDI child windows.

Menu—This window shows the use of pull-down menus. The source code shows you how to create and attach pull-down and pop-up items into pull-down menus.

Number—This window shows several implementations of the UIW_BIGNUM, UIW_INTEGER and UIW_REAL class objects.

String—This window shows several types of string objects that can be created with Zinc Application Framework. These objects include the basic UIW_STRING object, two types of UIW_FORMATTED_STRING class objects, and a multi-line text field (UIW_TEXT) that only occupies part of its parent window.

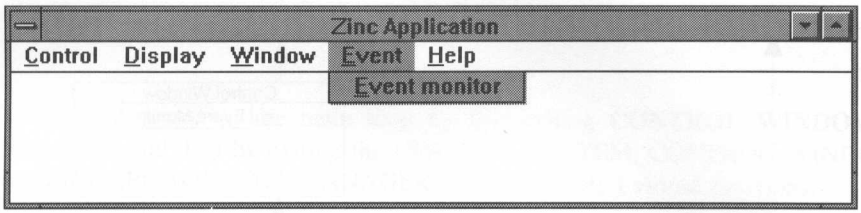
Text—This window shows a full-window implementation of a UIW_TEXT object and an associated vertical scroll bar.

Time—This window shows the many variations that can be used with the UIW_-TIME class object.

Tool bar—This window shows a tool bar object that contains various window objects.

CHAPTER 11 – EVENT OPTIONS

The ZincApp program's event option is shown under the "Event" menu item:



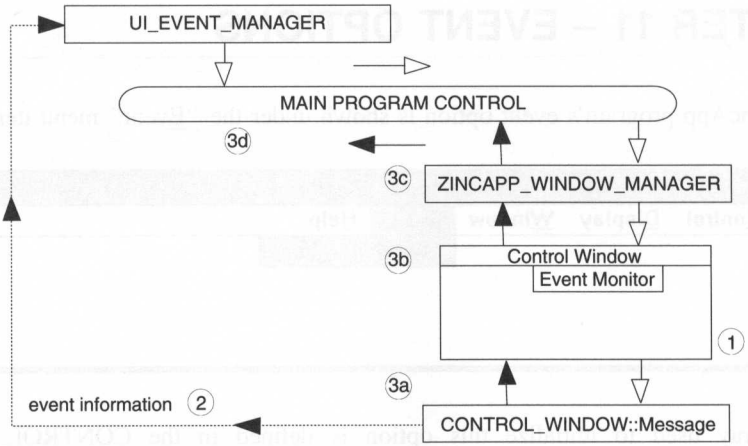
The array used to initialize this option is defined in the CONTROL_WINDOW constructor. It contains the following information:

```
static UI_ITEM eventItems[] =
{
    { MSG_EVENT_MONITOR,     VOIDF(CONTROL_WINDOW::Message),
        "&Event monitor"     MNIF_NO_FLAGS },
    { 0, 0, 0 }              // end of array
};
.
.
.

// Attach the sub-window objects to the control window.
*this
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON(SYF_GENERIC)
    + new UIW_TITLE("Zinc Application")
    + &(*new UIW_PULL_DOWN_MENU
        + new UIW_PULL_DOWN_ITEM("&Control", WNF_NO_FLAGS, controlItems)
        + new UIW_PULL_DOWN_ITEM("&Display", WNF_NO_FLAGS, displayItems)
        + &(*new UIW_PULL_DOWN_ITEM("&Window", WNF_NO_FLAGS)
            + controlItems
            + inputItems
            + selectItems)
        + new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS, eventItems)
        + new UIW_PULL_DOWN_ITEM("&Help", WNF_NO_FLAGS, helpItems));
```

Event program flow

When the event option is selected, initial program flow is handled the same way that the window options are handled. At the fifth step however, program flow is directed to the **OptionEvent( )** member function.

A complete explanation of this flow follows. (The corresponding steps are shown by the circled numbers in the figure.)



1—The **CONTROL_WINDOW::Message( )** function is called by the **UIW_POP_-UP_ITEM::Event( )** function. (The pop-up item inherits the code below from the UIW_BUTTON class.)

```
EVENT_TYPE UIW_BUTTON::Event(const UI_EVENT &event)
{
    .
    .
    .

    case L_SELECT:
    case L_END_SELECT:
        UI_EVENT tEvent;
        tEvent = event;
        if (userFunction)
            (*userFunction)(this, tEvent, ccode);
```

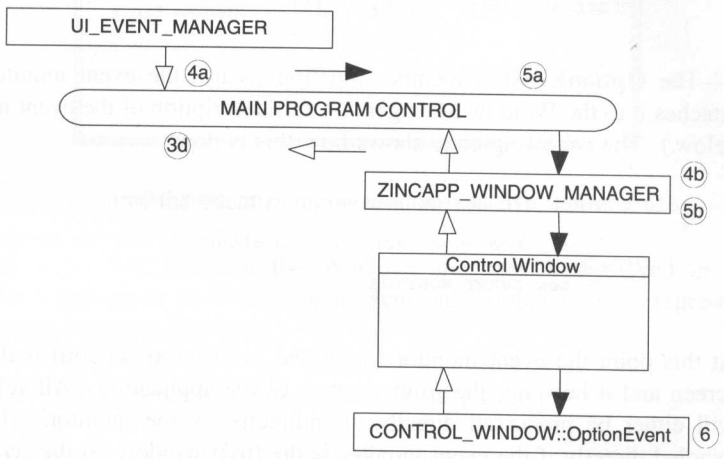
The arguments passed to **Message( )** are a pointer to the selected event option (*this*), a copy of the event that caused the user function to be called (*tEvent*), and the logical interpretation (*ccode*) of the event that caused **Event( )** to be called. (**NOTE:** the variable *tEvent* needs to be a copy of *event* since *event* is a constant variable whose values cannot be modified.)

2—The **CONTROL_WINDOW::Message( )** function sends a request to remove the temporary event option menu by sending an S_CLOSE_TEMPORARY message through the system via the Event Manager. It then sends the event request through the system by setting *event.type* to be MSG_EVENT and then by sending another message through the system.

```
EVENT_TYPE CONTROL_WINDOW::Message(UI_WINDOW_OBJECT *data,
    UI_EVENT &event, EVENT_TYPE ccode)
{
```

```
        if (ccode == L_SELECT)
        {
            for (UI_WINDOW_OBJECT *tObject = object->windowManager->First();
                tObject && FlagSet(tObject->woAdvancedFlags,
                WOAF_TEMPORARY);
            tObject = tObject->Next())
            object->eventManager->Put(UI_EVENT(S_CLOSE_TEMPORARY));
            event.type = ((UIW_POP_UP_ITEM *)object)->value;
            object->eventManager->Put(event);
        }
        return (ccode);
    }
```

3—Control returns to the main loop by first exiting **CONTROL_WINDOW::-Message( )** and then by exiting the UIW_POP_UP_ITEM, CONTROL_WINDOW and ZINCAPP_WINDOW_MANAGER classes' **Event( )** virtual functions.



4—The main loop picks up the program generated messages by calling **event-Manager->Get( )**. The first message received is S_CLOSE_TEMPORARY. This message is handled by the Window Manager and causes the event option menu to be removed from the screen.

5—The second message received is the event message MSG_EVENT. This message is passed by the main loop to the Window Manager, then is dispatched by the Window Manager to **CONTROL_WINDOW::Event( )** since the control window is the front window on the screen. The control window evaluates *event.type* (in this case the MSG_EVENT message)—resulting in the **OptionEvent( )** member function being called.

The code responsible for this control is shown below:

```
EVENT_TYPE CONTROL_WINDOW::Event(const UI_EVENT &event)
{
    .
    .
    .
    // Process the event according to its type.
    if (ccode >= MSG_HELP)
        OptionHelp(event.type);          // Help menu option selected.
    else if (ccode >= MSG_EVENT)
        OptionEvent(event.type);         // Event menu option selected.
    else if (ccode >= MSG_WINDOW)
        OptionWindow(event.type);        // Window menu option selected.
    else if (ccode >= MSG_DISPLAY)
        OptionDisplay(event.type);       // Display menu option selected.
    else
        ccode = UIW_WINDOW::Event(event);   // Unknown event.

    // Return the control code.
    return (ccode);
}
```

6—The **OptionEvent( )** member function creates the event monitor window and attaches it to the Window Manager. (A full description of the event monitor is given below.) The following code shows how this is done.

```
void CONTROL_WINDOW::OptionEvent(EVENT_TYPE item)
{
    // Create the event monitor and attach it to the window manager.
    *windowManager
        + new EVENT_MONITOR;
}
```

At this point the event monitor (described in the next section) is displayed on the screen and it becomes the front window of the application. All subsequent events will either be processed directly or indirectly by the monitor. (Events are only handled directly if the event monitor is the front window on the screen.)

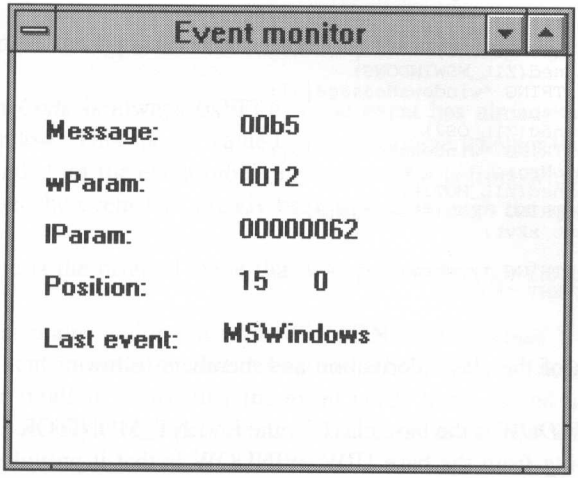
Monitoring library events

Monitoring events in the ZincApp program requires the definition and use of two derived classes: EVENT_MONITOR and ZINCAPP_WINDOW_MANAGER.

Event Monitor

The event monitor window is used to show the type of messages being processed by the library.

A picture of this window is shown below:



The Windows version of the event monitor window has five sections:

Message—The hex value of the Windows message is displayed in this field. (**NOTE:** A translation table could be implemented so that a textual representation of the message could be displayed.)

wParam—The *wParam* value of the event is displayed.

lParam—The *lParam* value of the event is displayed.

Position—The *Position* value of the event is displayed.

Last event—The last event section shows the interpreted value of the last event. This can be any of the Zinc Application Framework system or logical messages, or the interpreted keyboard or mouse code.

This window is implemented through a class called EVENT_MONITOR. The definition of this class is contained in **ZINCAPP.HPP**. Its members are shown below:

```
class EVENT_MONITOR : public UIW_WINDOW
{
public:
    EVENT_MONITOR(void);
    EVENT_TYPE Event(const UI_EVENT &event);
```

```
private:
#if defined (ZIL_MSDOS)
    UIW_STRING *keyboard[3];
    UI_EVENT kEvent;
    UIW_STRING *mouse[3];
    UI_EVENT mEvent;
#elif defined(ZIL_MSWINDOWS)
    UIW_STRING *windowsMessage[5];
    MSG wMsg;
#elif defined(ZIL_OS2)
    UIW_STRING *windowsMessage[5];
    QMSG oMsg;
#elif defined(ZIL_MOTIF)
    UIW_STRING *motifMessage[3];
    XEvent xEvt;
#endif
    UIW_STRING *system;
    UI_EVENT sEvent;
};
```

A description of the class' derivation and members follows:

- *UIW_WINDOW* is the base class for the EVENT_MONITOR class. The main reason for deriving from the base UIW_WINDOW is that it provides a very clean way of attaching a window to the screen that can receive message information, and a clean way of removing the window and monitoring capability once it is removed from the screen.

- *EVENT_MONITOR( )* is the constructor. When the event monitor window is attached to ZincApp's window manager (described below), it receives all events that pass through the system, after the front window has processed the event. This allows the front window to process the event normally, then for the event monitor to look at the type of action that was performed. If we were to derive the event monitor from UI_-DEVICE (such as the MACRO_HANDLER discussed in a later section) we would only receive raw input information. By positioning ourselves in the window manager, we are able to see how raw events are handled by an object. For example, pressing the left-mouse button on the title bar produces a series of messages ending in "Move." Pressing the left-mouse button in a text field however, produces the "Begin mark" message. If this class were positioned in the Event Manager, it would only interpret a left-down click for both types of events.

- *Event( )* is the function that processes the logical event. There are two types of events the **EVENT_MONITOR::Event( )** function can receive. The first type is messages passed to the window in the normal course of operation. These messages would be passed to the window if it were the front window on the screen, or if a mouse message overlapped the window's screen region. The second type of message is sent to the event monitor as a result of it being a monitor type window. These messages are received after they have been processed by the window manager. In addition, these special events are packaged by the window manager into a new event

and passed to the member function by the window manager. The window manager packages these events in the following fashion:

event.type is the logical event returned by the receiving object.

event.rawCode is always 0xFFFF if the event has already been passed to the front window. This special value lets us determine whether the original message was intended for the event monitor window (if it is front window on the screen) or whether the event has already been passed through the system.

event.data is the original event that was passed through the system.

There are four main sections to **EVENT_MONITOR::Event( )**. The first section sets up the event information and determines whether the event is intended for the window interpretation, or whether the event needs to be passed to the base UIW_-WINDOW class object for processing. The code associated with this section is shown below:

```
EVENT_TYPE EVENT_MONITOR::Event(const UI_EVENT &event)
{
    .
    .
    .

    // See if it is a normal event.                          (section 1)
    if (event.rawCode != 0xFFFF)
        return (UIW_WINDOW::Event(event));

    // Check for new keyboard event.                         (section 2)
    UI_EVENT *tEvent = (UI_EVENT *)event.data;
#if defined (ZIL_MSDOS)
    if (tEvent->type == E_KEY)
    .
    .
    .

    // Check for new mouse event.                            (section 3)
    else if (tEvent->type == E_MOUSE)
    .
    .
    .

#elif defined (ZIL_MSWINDOWS)
    if (tEvent->type == E_MSWINDOWS)
    .
    .
    .

#elif defined(ZIL_OS2)
    if (tEvent->type == E_OS2)
    .
    .
    .
```

```
#elif defined (ZIL_MOTIF)
    if (tEvent->type == E_MOTIF)
    .
    .
    .
#endif

    // Check for new logical event.                          (section 4)
    if (sEvent.type != event.type)
    .
    .
    .
    // Return the logical event.
    return (event.type);
}
```

- *keyboard* and *kEvent* contain information about the last key that was pressed. (See the "Last key" description above.) These variables are available in the DOS environment only. The variable *kEvent* keeps track of the last event for optimization so that only those parts of the key that have changed will be updated. When the **EVENT_MONITOR::Event( )** routine is called, these variables are changed to reflect the new event (passed as an argument to the event monitor's **Event( )** function). The code responsible for this change is shown below:

```
EVENT_TYPE EVENT_MONITOR::Event(const UI_EVENT &event)
{
    .
    .
    .
    UI_EVENT *tEvent = (UI_EVENT *)event.data;
    .
    .
    .
    // Check for new keyboard event.
    if (tEvent->type == E_KEY)
    {
        char string[32];
        if (kEvent.rawCode != tEvent->rawCode)
        {
            sprintf(string, "%04x", tEvent->rawCode);
            keyboard[0]->Information(SET_TEXT, string);
        }
        if (kEvent.key.shiftState != tEvent->key.shiftState)
        {
            sprintf(string, "%02x", tEvent->key.shiftState);
            keyboard[1]->Information(SET_TEXT, string);
        }
        if (kEvent.key.value != tEvent->key.value)
        {
            sprintf(string, "%c", tEvent->key.value);
            keyboard[2]->Information(SET_TEXT, string);
        }
        kEvent = *tEvent;
    }
```

- *mouse* and *mEvent* contain information about the last mouse event. These variables are available in the DOS environment only. They work just like the keyboard

variables *keyboard* and *kEvent* except that the information is maintained for the mouse. The variable *mevent* keeps track of the last event for optimization so that only those parts of the mouse event that have changed will be updated. When the **EVENT_MONITOR::Event( )** routine is called, these variables are changed to reflect the new event (passed as an argument to the event monitor's **Event( )** function). The code responsible for this change is shown below:

```
EVENT_TYPE EVENT_MONITOR::Event(const UI_EVENT &event)
{
    UI_EVENT *tEvent = (UI_EVENT *)event.data;
    .
    .
    .

    // Check for new mouse event.
    else if (tEvent->type == E_MOUSE)
    {
        char string[32];
        if (mEvent.rawCode != tEvent->rawCode)
        {
            sprintf(string, "%04x", tEvent->rawCode);
            mouse[0]->Information(SET_TEXT, string);
        }
        if (mEvent.position.column != tEvent->position.column)
        {
            sprintf(string, "%03d", tEvent->position.column);
            mouse[1]->Information(SET_TEXT, string);
        }
        if (mEvent.position.line != tEvent->position.line)
        {
            sprintf(string, "%03d", tEvent->position.line);
            mouse[2]->Information(SET_TEXT, string);
        }
        mEvent = *tEvent;
    }
```

- *windowsMessage* and *wMsg* contain the information from the last event that was received by the event monitor in the Windows environment. *windowsMessage* and *oMsg* contain the information from the last event that was received by the event monitor in the OS/2 environment. *motifMessage* and *xEvt* contain the information from the last event that was received by the event monitor in the Motif environment. The variables *wMsg*, *oMsg* and *xEvt* keep track of the last event for optimization so that only those parts of the event that have changed will be updated. When the **EVENT_MONITOR::Event( )** routine is called, these variables are changed to reflect the new event (passed as an argument to the event monitor's **Event( )** function). For example, the code responsible for this change in Windows is shown below:

```
EVENT_TYPE EVENT_MONITOR::Event(const UI_EVENT &event)
{
    .
    .
    .
```

```
    // See if it is a normal event.
    if (event.rawCode != 0xFFFF)
        return (UIW_WINDOW::Event(event));

    // Check for new keyboard event.
    UI_EVENT *tEvent = (UI_EVENT *)event.data;

#if defined(ZIL_MSDOS)
    .
    .
    .

#elif defined(ZIL_MSWINDOWS)
    if (tEvent->type == E_MSWINDOWS)
    {
        MSG msg = event.message;
        char string[32];
        if (wMsg.message != msg.message)
        {
            sprintf(string, "%04x", msg.message);
            windowsMessage[0]->Information(SET_TEXT, string);
        }
        if (wMsg.wParam != msg.wParam)
        {
            sprintf(string, "%04x", msg.wParam);
            windowsMessage[1]->Information(SET_TEXT, string);
        }
        if (wMsg.lParam != msg.lParam)
        {
            sprintf(string, "%08x", msg.lParam);
            windowsMessage[2]->Information(SET_TEXT, string);
        }
        if (wMsg.pt.x != msg.pt.x)
        {
            sprintf(string, "%d", msg.pt.x);
            windowsMessage[3]->Information(SET_TEXT, string);
        }
        if (wMsg.pt.y != msg.pt.y)
        {
            sprintf(string, "%d", msg.pt.y);
            windowsMessage[4]->Information(SET_TEXT, string);
        }
        wMsg = msg;
    }
#elif defined(ZIL_OS2)
    .
    .
    .
#elif defined(ZIL_MOTIF)
    .
    .
    .
#endif
    .
    .
    .
    // Return the logical event.
    return (event.type);
}
```

- *system* and *sEvent* contain information about the last interpreted event that was returned by the window object. These variables work just like the mouse variables *mouse* and *mEvent* except that the information is maintained for the logical or system

event. The variable *sEvent* keeps track of the last event for optimization so that only changes in the event cause the event field to be updated. When the **EVENT_-MONITOR::Event( )** routine is called, these variables are changed to reflect the new event (passed as an argument to the event monitor's **Event( )** function). The code responsible for this change is shown below (only a partial list of the event/string pair table is shown):

```
EVENT_TYPE EVENT_MONITOR::Event(const UI_EVENT &event)
{
    UI_EVENT *tEvent = (UI_EVENT *)event.data;

    // Declare the event type/name pairs.
    static struct EVENT_PAIR
    {
        int type;
        char *name;
    } eventTable[] =
    {
        { E_KEY,                "Key" },            // Raw events.
        { E_MOUSE,              "Mouse" },
        { E_CURSOR,             "Cursor" },
        { E_DEVICE,             "Device" },

        { S_ERROR,              "Error" },          // System events.
        { S_MINIMIZE,           "Minimize" },
        { S_MAXIMIZE,           "Maximize" },
        .
        .
        .
        { L_EXIT,               "Exit" },           // Logical events.
        { L_VIEW,               "View" },
        { L_SELECT,             "Select" },
        .
        .
        .

        { MSG_25x40_MODE,       "25x40 Text Mode" },// ZINCAPP events
        { MSG_25x80_MODE,       "25x80 Text Mode" },
        { MSG_43x80_MODE,       "43x80 Text Mode" },
        { MSG_GRAPHICS_MODE,    "Graphics Mode" },
        .
        .
        .

        { 0, 0 }                // End of array.
    };
    .
    .
    .

    // Check for new logical event.
    if (sEvent.type != event.type)
    {
        char *name = "<Unknown>";
        for (int i = 0; eventTable[i].type; i++)
            if (event.type == eventTable[i].type)
            {
                name = eventTable[i].name;
                break;
            }
        system->Information(name);
        sEvent = event;
    }
```

Window Manager

The event monitor (described previously) receives all interpreted messages by attaching itself to a Zinc Application window manager class called ZINCAPP_WINDOW_-MANAGER.

The definition of the ZINCAPP_WINDOW_MANAGER class is defined in **ZINCAPP.-HPP**. Its definition is shown below:

```
class ZINCAPP_WINDOW_MANAGER : public UI_WINDOW_MANAGER
{
public:
    ZINCAPP_WINDOW_MANAGER(UI_DISPLAY *display,
        UI_EVENT_MANAGER *eventManager) :
        UI_WINDOW_MANAGER(display, eventManager,
        ZINCAPP_WINDOW_MANAGER::ExitFunction) { }
    virtual EVENT_TYPE Event(const UI_EVENT &event);

private:
    static EVENT_TYPE ExitFunction(UI_DISPLAY *display,
        UI_EVENT_MANAGER *eventManager, UI_WINDOW_MANAGER *windowManager);
};
```

A description of the class' derivation and members follows:

- *UI_WINDOW_MANAGER* is the base class for the ZINCAPP_WINDOW_-MANAGER class. The derivation from this class allows us to get all interpreted messages before they are passed to the main control loop and to send the event information to any event monitor windows.

- *ZINCAPP_WINDOW_MANAGER( )* is the ZincApp window manager constructor. It calls the base UI_WINDOW_MANAGER with the display and eventManager supplied by its arguments but also provides an exitFunction pointer that is the **ZINCAPP_WINDOW_MANAGER::ExitFunction( )** static member function (described below). The ZincApp window manager class is constructed in the main section of our program, just the way a normal window manager would be constructed. The code below shows how this is done:

```
.
.
.
// Initialize the ZincApp window manager and add the control window.
ZINCAPP_WINDOW_MANAGER *windowManager =
    new ZINCAPP_WINDOW_MANAGER(display, eventManager);

UI_WINDOW *window = new CONTROL_WINDOW;
*windowManager
    + new window;
```

- *Event( )* is the function that processes the event information. It contains two major sections:

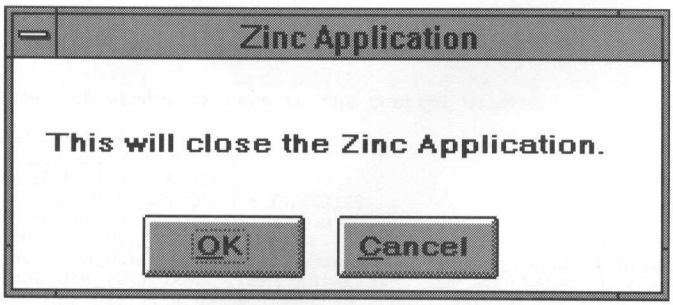
```
EVENT_TYPE ZINCAPP_WINDOW_MANAGER::Event(const UI_EVENT &event)
{
    // Allow the base window manager to process the event.
    EVENT_TYPE ccode = UI_WINDOW_MANAGER::Event(event);        (section 1)

    // Send the event to any event monitor windows.
    for (UI_WINDOW_OBJECT *object = First(); object;
        object = object->Next())
        if (object->userFlags == MSG_EVENT_MONITOR &&          (section 2)
            event.type != S_RESET_DISPLAY)
        {
            UI_EVENT tEvent(event.type, 0xFFFF);
            tEvent.data = (void *)&event;
            object->Event(tEvent);
        }

    // Return the control code.
    return (ccode);
}
```

The first section calls **UI_WINDOW_MANAGER::Event** so that it can dispatch the message to the proper window.

The second section is used to dispatch the interpreted message to any event monitoring windows. It determines these windows by looking at the object's *userFlags*. If the flag is set to be MSG_EVENT_MONITOR (by **EVENT_-MONITOR::Event( )**) and if the event type is not S_RESET_DISPLAY, the message is sent to the device. This event is modified to contain the logical code in *event.type*, the value 0xFFFF in *event.rawCode*, and the raw event is pointed to by *event.data*.

- *ExitFunction( )* is a function that displays a modal exit window to the screen.

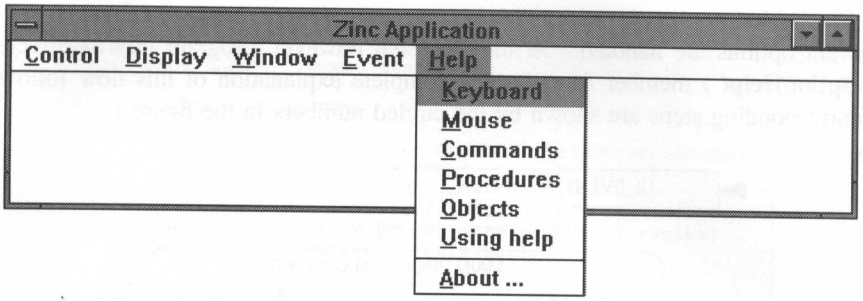
A picture of this window is shown below:



If the user selects "OK" an L_EXIT message is passed through the system via the Event Manager, and program execution ceases. Otherwise, the window is removed from the screen and program flow continues in a normal fashion.

CHAPTER 12 – HELP OPTIONS

The ZincApp program's help options are shown under the "Help" menu item:



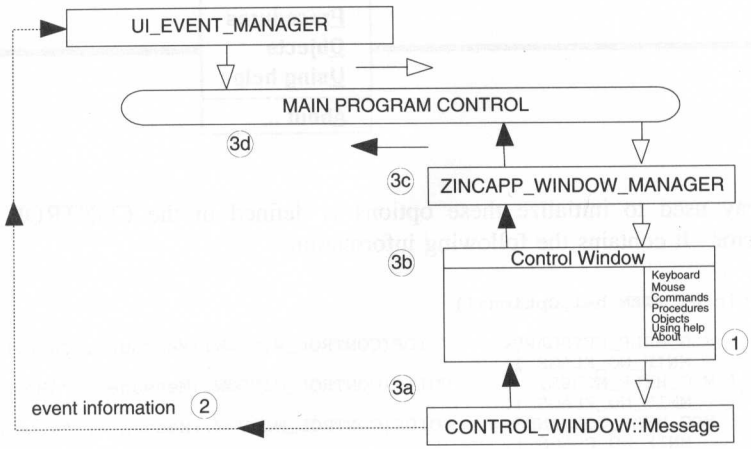
The array used to initialize these options is defined in the CONTROL_WINDOW constructor. It contains the following information:

```
static UI_ITEM helpOptions[] =
{
    { MSG_HELP_KEYBOARD,     VOIDF(CONTROL_WINDOW::Message), "&Keyboard",
      MNIF_NO_FLAGS },
    { MSG_HELP_MOUSE,        VOIDF(CONTROL_WINDOW::Message), "&Mouse",
      MNIF_NO_FLAGS },
    { MSG_HELP_COMMANDS,     VOIDF(CONTROL_WINDOW::Message), "&Commands",
      MNIF_NO_FLAGS },
    { MSG_HELP_PROCEDURES,   VOIDF(CONTROL_WINDOW::Message), "&Procedures",
      MNIF_NO_FLAGS },
    { MSG_HELP_OBJECTS,      VOIDF(CONTROL_WINDOW::Message), "&Objects",
      MNIF_NO_FLAGS },
    { MSG_HELP_HELP,         VOIDF(CONTROL_WINDOW::Message), "&Using help",
      MNIF_NO_FLAGS },
    { 0,                     0,                              "",
      MNIF_SEPARATOR },
    { MSG_HELP_ZINCAPP,      VOIDF(CONTROL_WINDOW::Message), "&About ...",
      MNIF_NO_FLAGS },
    { 0, 0, 0, 0 }  // End of array.
};
.
.
.
// Attach the sub-window objects to the control window.
*this
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON(SYF_GENERIC)
    + new UIW_TITLE("Zinc Application")
    + &(*new UIW_PULL_DOWN_MENU
        + new UIW_PULL_DOWN_ITEM("&Control", WNF_NO_FLAGS, controlItems)
        + new UIW_PULL_DOWN_ITEM("&Display", WNF_NO_FLAGS, displayItems)
        + &(*new UIW_PULL_DOWN_ITEM("&Window", WNF_NO_FLAGS)
            + controlItems
            + inputItems
            + selectionItems)
```

```
        + new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS, eventItems)
        + new UIW_PULL_DOWN_ITEM("&Help", WNF_NO_FLAGS, helpItems));
```

Help program flow

When a help option is selected, initial program flow is handled the same way that the event options are handled. At the fifth step however, program flow is directed to the **OptionHelp( )** member function. A complete explanation of this flow follows. (The corresponding steps are shown by the circled numbers in the figure.)



1—The **CONTROL_WINDOW::Message( )** function is called by **UIW_POP_UP_-ITEM::Event( )**. (The pop-up item inherits the code below from UIW_BUTTON.)

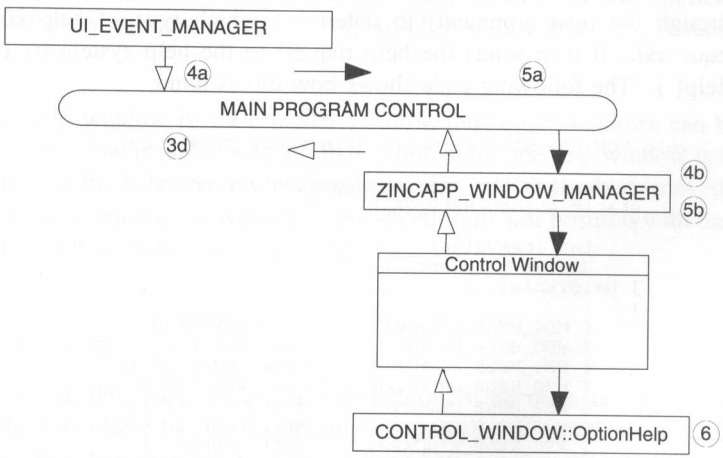
```
EVENT_TYPE UIW_BUTTON::Event(const UI_EVENT &event)
{
    .
    .
    .

    case L_SELECT:
    case L_END_SELECT:
        UI_EVENT tEvent;
        tEvent = event;
        if (userFunction)
            (*userFunction)(this, tEvent, ccode);
```

The arguments passed to **Message( )** are a pointer to the selected help option (*this*) and a copy of the event that caused the user function to be called (*tEvent*). (**NOTE:** the variable *tEvent* needs to be a copy of *event* since *event* is a constant variable whose values cannot be modified.)

2—The **CONTROL_WINDOW::Message( )** function sends a request to remove the temporary help options menu by sending an S_CLOSE_TEMPORARY message through the system via the Event Manager. It then sends the help request through the system by setting *event.type* to be the menu item's value (i.e., one of the MSG_HELP values defined in the *helpOptions* array).

```
EVENT_TYPE CONTROL_WINDOW::Message(UI_WINDOW_OBJECT *object,
    UI_EVENT &event, EVENT_TYPE ccode)
{
    if (ccode == L_SELECT)
    {
        for (UI_WINDOW_OBJECT *tObject = object->windowManager->First();
            tObject && FlagSet(tObject->woAdvancedFlags,
            WOAF_TEMPORARY); tObject = tObject->Next());
            object->eventManager->Put(UI_EVENT(S_CLOSE_TEMPORARY));
        event.type = ((UIW_POP_UP_ITEM *)object)->value;
        object->eventManager->Put(event);
    }
    return (ccode);
}
```

3—Control returns to the main loop by first exiting **CONTROL_WINDOW::-Message( )** and then by exiting the UIW_POP_UP_ITEM, CONTROL_WINDOW and ZINCAPP_EVENT_MANAGER classes' **Event( )** virtual functions.



4—The main loop picks up the program generated messages by calling **event-Manager->Get( )**. The first message received is S_CLOSE_TEMPORARY. This message is handled by the Window Manager and causes the help options menu to be removed from the screen.

5—The second message received is the help message determined by the selected menu item. This message is passed by the main loop to the Window Manager, then is dispatched by the Window Manager to **CONTROL_WINDOW::Event( )** since

the control window is the front window on the screen. The control window evaluates *event.type* (in this case a MSG_HELP message)—resulting in the **OptionHelp( )** member function being called. The code responsible for this control is shown below:

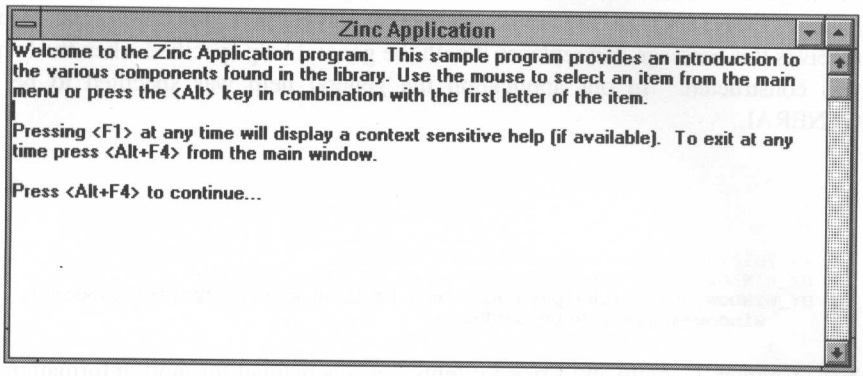
```
EVENT_TYPE CONTROL_WINDOW::Event(const UI_EVENT &event)
{
    EVENT_TYPE ccode = event.type;
    .
    .
    .

    // Process the event according to its type.
    if (ccode >= MSG_HELP)
        OptionHelp(event.type);                 // Help option.
    else if (ccode >= MSG_EVENT)
        OptionEvent(event.type);                // Event option.
    else if (ccode >= MSG_WINDOW)
        OptionWindow(event.type);               // Window option.
    else if (ccode >= MSG_DISPLAY)
        OptionDisplay(event.type);              // Display option.
    else
        ccode = UIW_WINDOW::Event(event);       // Unknown event.

    // Return the control code.
    return (ccode);
}
```

6—The **OptionHelp( )** member function evaluates the item's value (passed down through the *item* argument) to determine which type of help context has been requested. It then sends the help request to the help system by calling **DisplayHelp( )**. The following code shows how this is done:

```
void CONTROL_WINDOW::OptionHelp(EVENT_TYPE item)
{
    // Declare the help message/context pairs.
    static struct HELP_PAIR
    {
        int itemValue;
        USHORT helpContext;
    } helpTable[] =
    {
        { MSG_HELP_KEYBOARD,     HELP_KEYBOARD },
        { MSG_HELP_MOUSE,        HELP_MOUSE },
        { MSG_HELP_COMMANDS,     HELP_COMMANDS },
        { MSG_HELP_PROCEDURES,   HELP_PROCEDURES },
        { MSG_HELP_OBJECTS,      HELP_OBJECTS },
        { MSG_HELP_HELP,         HELP_HELP },
        { MSG_HELP_ZINCAPP,      HELP_GENERAL },
        { 0, 0 }     // End of array.
    };

    // Get the help context then call the help system.
    USHORT helpContext = NO_HELP_CONTEXT;
    for (int i = 0; helpTable[i].itemValue; i++)
        if (item == helpTable[i].itemValue)
        {
            helpContext = helpTable[i].helpContext;
            break;
        }
    helpSystem->DisplayHelp(windowManager, helpContext);
}
```

Once the help system's **DisplayHelp( )** function has been called the help window is attached to the Window Manager.

For example, the help request MSG_HELP_ZINCAPP causes the following help window to appear:



At this point the help window becomes the front window of the application. All subsequent events are processed by the help window until a change is requested by the end-user.

NOTE: The help window is not a modal window, thus other windows can be selected while the help window is on the screen. In addition, only one help window is defined for an application. If the help window is already present, or if it has been moved and sized by a previous help request, the window is presented in its last position with the new help information shown in its title and text fields.

General library help

In addition to the help information provided through the main control menu, context sensitive help is available by simply pressing <F1> during the application. Each window created in the ZincApp program has a pre-defined help context. This context is specified when the window is constructed. For example, the main control window has HELP_-MAIN_CONTROL specified as its help context. The code below shows where this context is specified:

```
CONTROL_WINDOW::CONTROL_WINDOW(void) :
    UIW_WINDOW(0, 0, 52, 13, WOF_NO_FLAGS, WOAF_LOCKED, HELP_MAIN_CONTROL)
{
    .
    .
    .
}
```

In general, window help is managed by the **UI_WINDOW_OBJECT::Event( )** function. This control is similar to that shown in the steps above. After the <F1> key is pressed the Window Manager dispatches the message to the front window. If the window has an accompanying help context, the help system is called with the type of help associated with the window. (In the case of the control window it would be a request for the HELP_-MAIN_CONTROL help context.) Otherwise, general help is requested by sending NO _HELP_CONTEXT to the **helpSystem->DisplayHelp( )** function. The help system receives this message and replaces it with the general help specified when the help system was constructed. In our application the general help was specified to be HELP_-GENERAL.

```
.
.
.

// Initialize the help and error systems.
UI_WINDOW_OBJECT::errorSystem = new UI_ERROR_SYSTEM;
UI_WINDOW_OBJECT::helpSystem = new UI_HELP_WINDOW_SYSTEM("support",
    windowManager, HELP_GENERAL);
```

See "Chapter 2—Help and Error Systems" of this manual for more information on using the help system.

SECTION IV
DERIVED CLASSES

CHAPTER 13 – MACRO DEVICE

This tutorial shows you how to create a keyboard macro input device. When we are finished, you should understand:

- the design used to implement a simple keyboard macro
- the basic design rules that control the operation of input devices within Zinc Application Framework
- the type of information needed to initialize the UI_DEVICE base class

The source code associated with this program is located in the **\ZINC\TUTOR\MACRO** subdirectory. It contains the following files:

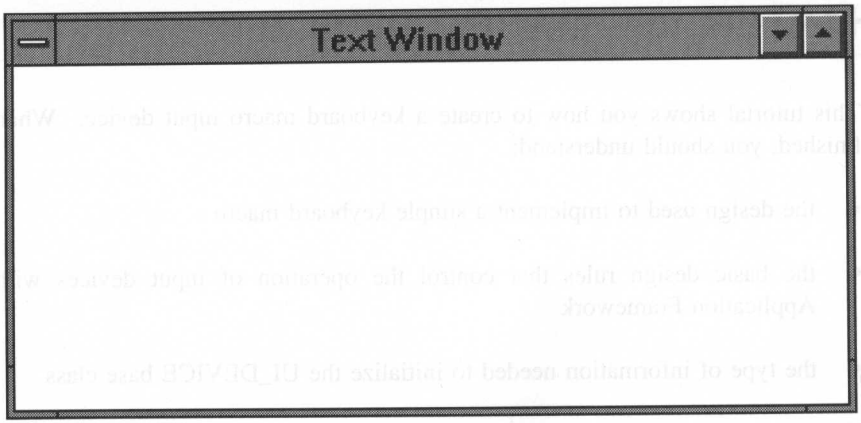
MACRO.CPP—This file contains the macro device member functions **MACRO_HANDLER::Event( )** and **MACRO_HANDLER::Poll( )**, as well as the main program loop (**UI_APPLICATION::Main( )**).

.DEF**, ***.RC**—These files are the environment specific definition and resource files required when compiling for Windows or OS/2. (**NOTE:** The **W*. files are for Windows and the **O*.*** files are for OS/2.)

***.MAK**—These files are the compiler-dependent makefiles associated with the Macro program. (See "Chapter 1—Initializing the Library" for information on compiling for each Zinc-supported platform.)

Program execution

Let's begin by looking at how the keyboard macro operates in a sample application. To do this, compile and run the application **MACRO.EXE**. The following window should appear on the screen:



The current object in the window is a text object. (It is a non-field region so it takes up the entire region within the window.) You should be able to type text information into this window. In addition, four simple macro keys have been implemented:

Pressing <F5> causes the text "Macro #1" to be entered into the text window.

Pressing <F6> causes the text "Macro #2" to be entered into the text window.

Pressing <F7> causes the text "Macro #3" to be entered into the text window.

Pressing <F8> causes the text "Macro #4" to be entered into the text window.

When you are finished experimenting with the program, exit by either selecting "Close" from the system button's pop-up menu, or by pressing <Alt+F4>.

Class definition

The macro keys described above are implemented as a single input device called MACRO_HANDLER. This device is created and attached to the Event Manager using the + operator. The following code shows this implementation:

```
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
*eventManager
    + new UID_KEYBOARD
    + new UID_MOUSE
    + new UID_CURSOR;
*eventManager
    + new MACRO_HANDLER(macroTable);
```

The definition of the macro device is given below:

```
const EVENT_TYPE E_MACRO = 89;

struct MACRO_PAIR
{
    RAW_CODE rawCode;
    char *macro;
};

class MACRO_HANDLER : public UI_DEVICE
{
public:
    MACRO_HANDLER(MACRO_PAIR *_macroTable) : UI_DEVICE(E_MACRO, D_OFF),
        macroTable(_macroTable) { installed = TRUE; }
    EVENT_TYPE Event(const UI_EVENT &event);
private:
    MACRO_PAIR *macroTable;
    MACRO_PAIR *currentMacro;
    int offset;

    void Poll(void);
};
```

This class uses the following definitions and member variables:

- *E_MACRO* is a constant value that is used to uniquely identify the macro device. Zinc Application Framework pre-defines the values for the keyboard, mouse and cursor devices but leaves other values open for programmer-defined input devices. The significance of the value 89 will be discussed later in this chapter.

- *MACRO_PAIR* is a structure that allows you to define a keyboard/macro equivalent pair. The definition of the four macro keys we used in our sample program is sho wn below:

```
MACRO_PAIR macroTable[] =
{
    { F5, "Macro #1." },
    { F6, "Macro #2." },
    { F7, "Macro #3." },
    { F8, "Macro #4." },
    { 0, NULL }
};
```

  The entry { 0, NULL } is used as an end-of-array indicator. In addition, the use of F5, F6, F7 and F8 in the array above requires us to define a constant value called USE_RAW_KEYS. This definition allows us to have access to the raw DOS scan codes defined in **UI_EVT.HPP**.

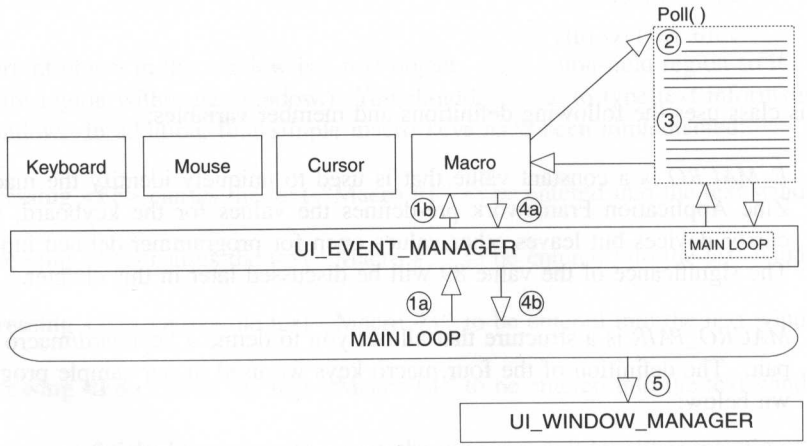
- *macroTable* is a pointer to the table that contains the *rawCode/macro* pairs to be matched. In our program this table is *macroTable* (shown above).

- *currentMacro* is a pointer to the current, or active, macro (if any). This value is reset whenever a new macro key is pressed.

- *offset* is a value that gives the position of the current keyboard input within the *currentMacro->macro* character array. It is used when the macro device feeds keyboard information into the Event Manager's input queue. (The terms "input queue" and "event queue" are synonymous.)

Conceptual operation

The conceptual operation of the macro device, after it has been attached to the Event Manager, is shown in the figure below:



This operation can be described through the following steps. (The corresponding steps are shown by the circled numbers in the figure.)

1—The device is polled (i.e., its **Poll( )** routine is called) whenever the programmer calls **eventManager->Get( )**. If a macro key has just been pressed, the process goes to the second step. If the macro device is currently enabled (i.e., feeding information into the input queue) the process goes to the third step. Otherwise, program flow returns to the main. The code associated with this step is shown below. (**NOTE:** The step identifications to the right are not part of the actual code.)

```
void MACRO_HANDLER::Poll(void)                                    (step 1)
{
    // See if any events are in the event manager's input queue.
    UI_EVENT event;
    static int emptyQueue = TRUE;
```

```
        if (emptyQueue)
            emptyQueue = eventManager->Get(event,
                Q_NO_POLL | Q_NO_BLOCK | Q_NO_DESTROY | Q_BEGIN);

        // Check for environment-specific keyboard events.          (step 2)
    #if defined (ZIL_MSWINDOWS)
        if (state == D_OFF && !emptyQueue && event.type == E_MSWINDOWS &&
            event.message.message == WM_KEYDOWN)
        {
            .
            .
            .
    #elif defined (ZIL_OS2)
        if (!emptyQueue && event.type == E_OS2 &&
            event.message.msg == WM_CHAR)
        {
            .
            .
            .
    #elif defined (ZIL_MOTIF)
        if (!emptyQueue && event.type == E_MOTIF &&
            event.message.type == KeyPress)
        {
            .
            .
            .
    #endif
            .
            .
            .
        }

        // See if the event is a macro key.                         (step 3)
        if (state == D_OFF && !eventQueue && event.type == E_KEY)
        {
            .
            .
            .
        }

        // Put macro information into the input queue.              (step 4)
        if (state == D_ON && emptyQueue)
        {
            .
            .
            .
        }
    }
```

You may have noticed that **eventManager->Get( )** is called with several parameters. Since we are getting input while in an input device function, we must be very careful not to recursively call **eventMangager->Get( )**. The way we protect against further recursion is to set the Q_NO_POLL flag. This prevents the Event Manager from calling any other input devices. The Q_NO_BLOCK flag prevents the Event Manager from stopping program execution until an event is detected. We set this since we only want to "check" the input queue to see if an event is available. (A value of 0 is returned if there is an event in the queue. Otherwise, a negative value is returned.)

Next, we do not want to destroy the contents of the queue since we are only looking for special keyboard events. The way this is done is by setting the Q_NO_DESTROY flag. This allows us to obtain a copy of the event information without removing it from the queue. The Q_BEGIN flag is used to get information from the beginning of the queue, rather than from the end.

2—The second step is to check for events that are specific to a particular environment. If these types of events are received, they are translated to the generic Zinc event format for processing.

3—The third step is only executed if a new macro key has been pressed and the key has been entered into the input queue by the UID_KEYBOARD device. In this step, the type of macro is determined. If a valid macro key has been entered, all other input devices are shut off so that they won't feed additional information into the queue while we are putting in our macro events. Next, the original macro key is removed from the Event Manager's input queue and the macro device is enabled. The first character of the new macro is placed into the input queue by continuing to the third step (i.e., setting the *emptyQueue* flag to be TRUE causes step 3 to be executed). The code below shows how this step is implemented:

```
void MACRO_HANDLER::Poll(void)                                    (step 1)
{
    // See if any events are in the event manager's input queue.
    UI_EVENT event;
    int emptyQueue = eventManager->Get(event,
        Q_BEGIN | Q_NO_DESTROY | Q_NO_BLOCK | Q_NO_POLL);

    // Check for environment-specific keyboard events.              (step 2)
#if defined (ZIL_MSWINDOWS)
    if (state == D_OFF && !emptyQueue && event.type == E_MSWINDOWS &&
        event.message.message == WM_KEYDOWN)
    {
        .
        .
        .

#elif defined (ZIL_OS2)
    if (!emptyQueue && event.type == E_OS2 &&
        event.message.msg == WM_CHAR)
    {
        .
        .
        .

#elif defined (ZIL_MOTIF)
    if (!emptyQueue && event.type == E_MOTIF &&
        event.message.type == KeyPress)
    {
        .
        .
        .

#endif
```

```
        .
        .
        .
    }

    // See if the event is a macro key.                          (step 3)
    if (state == D_OFF && !emptyQueue && event.type == E_KEY)
    {
        for (int i = 0; macroTable[i].rawCode && !emptyQueue; i++)
            if (event.rawCode == macroTable[i].rawCode)
            {
                // Turn off all other devices while we feed the macro.
                eventManager->DeviceState(E_DEVICE, D_OFF);
                eventManager->Get(event, Q_BEGIN | Q_NO_POLL);
                currentMacro = &macroTable[i];
                offset = 0;
                state = D_ON;
                // Set emptyQueue to be TRUE so we go to the next step.
                emptyQueue = TRUE;
                break;
            }
    }

    // Put macro information into the input queue                 (step 4)
    if (state == D_ON && emptyQueue)
    {
        .
        .
        .

    }
}
```

4—The fourth step is only executed if the macro device has been enabled. Once the macro device is enabled, it feeds one event into the input queue each time its **Poll( )** routine is called, but only if there are no other events waiting to be processed by the Event Manager. Once the macro device runs out of input information, it changes its *state* to D_OFF. This prevents the third step from being executed until another macro key is pressed.

```
void MACRO_HANDLER::Poll(void)                                  (step 1)
{
    // See if any events are in the event manager's input queue.
    UI_EVENT event;
    int emptyQueue = eventManager->Get(event,
        Q_BEGIN | Q_NO_DESTROY | Q_NO_BLOCK | Q_NO_POLL);

    // Check for environment-specific keyboard events.          (step 2)
#if defined (ZIL_MSWINDOWS)
    if (state == D_OFF && !emptyQueue && event.type == E_MSWINDOWS &&
        event.message.message == WM_KEYDOWN)
    {
        .
        .
        .

#elif defined (ZIL_OS2)
    if (!emptyQueue && event.type == E_OS2 &&
        event.message.msg == WM_CHAR)
    {
        .
        .
        .
```

```
#elif defined (ZIL_MOTIF)
    if (!emptyQueue && event.type == E_MOTIF &&
        event.message.type == KeyPress)
    {
        .
        .
        .
#endif
        .
        .
        .
    }

    // See if the event is a macro key.                          (step 3)
    if (state == D_OFF && !emptyQueue && event.type == E_KEY)
    {
        .
        .
        .
    }

    // Put macro information into the input queue.                (step 4)
    if (state == D_ON && emptyQueue)
    {
        event.type = E_KEY;
        event.rawCode = currentMacro->macro[offset];
        event.key.value = event.rawCode;
        event.key.shiftState = 0;
        eventManager->Put(event, Q_END);
        if (!currentMacro->macro[++offset])
        {
            eventManager->DeviceState(E_DEVICE, D_ON);
            state = D_OFF;
        }
    }
}
```

5—Program flow is returned to the programmer in two stages. First, control returns to the Event Manager when the input devices return from their **Poll( )** functions, then if an event is present in the input queue, program control returns to the main loop.

6—The main program loop processes all event information, including the macro key expansions, by calling **windowManager->Event( )**. The main program loop then exits if the L_EXIT message is received, or it returns to the first step to get the next event.

Class information

The MACRO_HANDLER class constructor is defined as an in-line function.

```
class MACRO_HANDLER : public UI_DEVICE
{
public:
    MACRO_HANDLER(MACRO_PAIR *_macroTable) : UI_DEVICE(E_MACRO, D_OFF),
        macroTable(_macroTable) { installed = TRUE; }
```

Base class initialization

The base UI_DEVICE class constructor is called before any class specific information is set. It requires the specification of the device's type (E_MACRO) and its initial state (D_OFF).

The Event Manager uses the input device *type* to determine the device's order in the device list. Input devices are arranged in the device list in ascending *type* order. Thus, the order of the four input devices we attached to the Event Manager is:

UID_KEYBOARD—Its value is 10, the number associated with the constant variable E_KEY.

UID_MOUSE—Its value is 30, the number associated with the constant variable E_MOUSE.

UID_CURSOR—Its value is 50, the number associated with the constant variable E_CURSOR.

MACRO_HANDLER—We assigned it the value 89, so that it would be the last device in the list.

We need the macro handler to be the last device in the list so that its **Poll( )** function can review any activity that has been performed since the last call to **eventManager->Get( )**. For example, if the user presses <F5>, the keyboard's **Poll( )** function will put the character <F5> into the Event Manager's input queue. Later, the macro device's **Poll( )** function will be called. When it is, the macro handler will find the <F5> value entered by the keyboard. If we assign the macro handler a lower number than that assigned to the keyboard, the macro handler will always check the input queue before the keyboard feeds its information and will never see the <F5> key (i.e., it will be passed to the main control before the macro handler is called again).

The initial state of the macro device needs to be off so that the program doesn't think macro information is being fed into the input queue. The Event Manager does not look at the state of devices, but devices generally use the information internally to determine what types of operations to perform. The macro device can be in one of the following two states:

D_OFF—If the macro device is in this state, no macro information is being entered into the input queue.

D_ON—If the macro device is in this state, it is currently feeding information into the input queue.

The Event Manager and base UI_DEVICE classes set three other variables:

- *enabled* is used as a second-level state indicator. The base device class sets this variable to be TRUE, but it is ignored by the macro device.
- *display* is a pointer to the screen display that was created in the main program loop. This variable is not set until the macro device is attached to the Event Manager. The macro device does not use this pointer.
- *eventManager* is a pointer to the Event Manager where the macro device is attached. The macro device uses this pointer to make queries on and feed information to the input queue.

Member variable initialization

The class member *macroTable* is initialized to point to the constructor argument *_macroTable*. This variable is used as the search table for keyboard/macro expansions. The array specified in this argument must not be destroyed until the class is destroyed by the Event Manager.

The last thing the class constructor does is override the base class member *installed*. The value specified is TRUE. This value is not used by the Event Manager, but it does provide consistency when checking for device installation.

The class members *currentMacro* and *offset* are not set until the state of the device changes to D_ON.

The Poll function

The **MACRO_HANDLER::Poll( )** function was described in the conceptual operation part of this chapter. In general, **Poll( )** functions should be used for the following:

1—To feed information to or get information from the Event Manager's input queue. The keyboard and mouse devices all have poll routines that feed information into the input queue.

2—To pass control to an object on a periodic basis. Many environments do not support multi-tasking. In these environments the use of a poll routine is beneficial because it ensures that all devices will be polled each time the **eventManager->Get( )** function is called. The cursor device uses this method to paint and remove an XOR region to the screen, simulating a blinking cursor. It does this by keeping track of time intervals and blinking the cursor in a consistent fashion.

The macro device feeds information to and gets information from the Event Manager. Information is fed into the input queue when the device is "on" and checks the input when it is "off."

The Event function

The **MACRO_HANDLER::Event( )** function is defined below:

```
class MACRO_HANDLER : public UI_DEVICE
{
public:
    EVENT_TYPE Event(const UI_EVENT &event);
```

This routine must be declared by the macro device since the base UI_DEVICE class declares it to be a pure virtual function (i.e., a function with an = *0* statement at the end).

```
class UI_DEVICE : public UI_ELEMENT
{
    public:
    virtual EVENT_TYPE Event(const UI_EVENT &event) = 0;
```

In general, **Event( )** functions are used to change the state of an input device.

Enhancements

Now that we have discussed the basic design and implementation of a keyboard macro device, let's evaluate some variations you could implement to make the device more powerful. (**NOTE:** The actual implementation of these ideas is left to the reader.)

1—Stuff the input buffer all at once, rather than one character at a time. This could be accomplished by modifying the **Poll( )** routine to put all macro characters into the input queue in one step. The benefits of this method are that it simplifies the process of the macro device and that it prevents the need for disabling all other input devices. The problem with this implementation is two-fold. First, the macro may fill the input buffer, in which case we will have to write code to wait until the buffer is not full. Second, the macro may itself contain a character that is a macro key. This would

require modification to our member variables and may cause recursion of macro events.

2—Modify the static variables *UIW_STRING::pasteBuffer* and *UIW_STRING::pasteLength* to contain the macro, then send an L_PASTE message through the system. This is a slick implementation whose only drawbacks are that it wipes out the old information in the global paste buffer and that the receiving object may not be a simple text field, like the window created in our application.

3—Extend the macro device to enable the addition or deletion of macro pairs. This could be accomplished by overloading the + and – operators for the MACRO_HANDLER class.

4—Extend the macro pair to handle logical, system or normal keyboard information. In this scenario, you would need to modify the definition of *MACRO_PAIR.macro* to support UI_EVENT information rather than simple character values. In addition, you would probably want to write an editor so that the macro could be easily edited and modified. This would require that you set up an edit window (using the UIW_WINDOW class) that contained the macro key, a list of mapping events, and menu-items or buttons that would let you add, delete or modify the contents of the list.

You should now understand the design associated with a macro device and the basic design and implementation rules associated with input devices in general. If you are able to understand this information, you are well on your way to understanding the operation of the Event Manager within Zinc Application Framework and the way in which input devices operate within the library.

CHAPTER 14 – HELP BAR

This tutorial shows you how to create a help bar object. When we are finished, you should understand:

- how window objects can communicate with the help bar class
- the design used to implement an object that displays help text at the bottom of the parent window
- the basic design rules that control the operation of windows and window objects within Zinc Application Framework
- how to derive a new window object from the UI_WINDOW_OBJECT base class
- how to implement a new window object in Microsoft Windows, OS/2 and Motif.

The source code associated with this program is located in **\ZINC\TUTOR\HELPBAR**. It contains the following files:

HELPBAR.CPP—This file contains the main program loop (i.e., **UI_APPLICATION::Main( )**) as well as the static functions **InformationWindow( )**, **SetHelp( )** and **ActionFunction( )**.

HLPBAR.CPP—This file contains the **HELP_BAR** class source code.

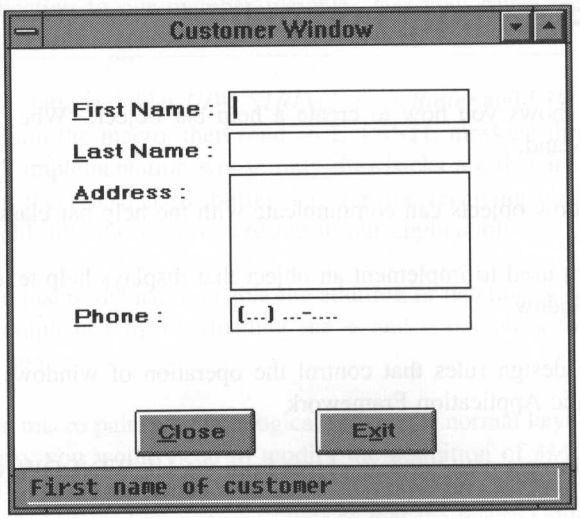
HLPBAR.HPP—This file contains the **HELP_BAR** class definition.

.DEF**, ***.RC**—These files are the environment specific definition and resource files required when compiling for Windows or OS/2. (**NOTE:** The **W*. files are for Windows and the **O*.*** files are for OS/2.)

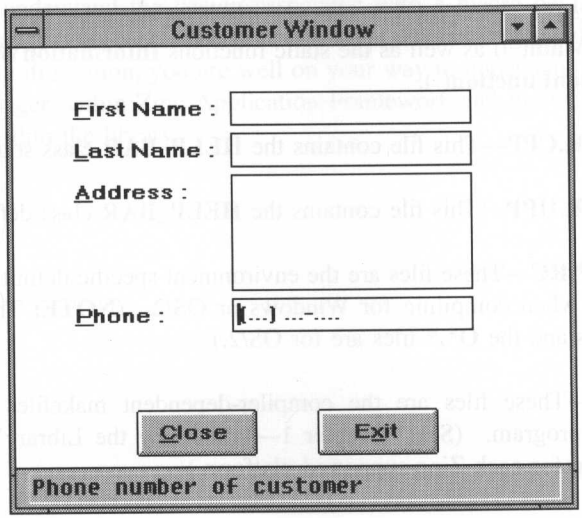
***.MAK**—These files are the compiler-dependent makefiles associated with the Helpbar program. (See "Chapter 1—Initializing the Library" for information on compiling for each Zinc-supported platform.)

Program execution

The operation of help bar objects can be seen by compiling and running the application **HELPBAR.EXE**. Two copies of the following window should appear on the screen:



There is no direct interaction with the help bar object; it is simply used to display help information associated with the current window object. For example, making the "Phone" field current will cause the following text to be displayed on the help bar:



When you are done experimenting with the help bar tutorial program, exit either by selecting the "Exit" button, the "Exit" option from the system button's menu, or by typing <Alt+F4>.

Class definition

The help bar object is implemented with a class called HELP_BAR. The HELP_BAR definition (contained in **HELPBAR.HPP**) is given below:

```
// Help bar objectID.
const OBJECTID ID_HELP_BAR = 3005;

class HELP_BAR : public UI_WINDOW_OBJECT
{
public:
    HELP_BAR(char *text = NULL);
    ~HELP_BAR(void);
    EVENT_TYPE Event(const UI_EVENT &event);

    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);
    static UI_WINDOW_OBJECT *New(const char *, UI_STORAGE *,
        UI_STORAGE_OBJECT *);
    virtual void Store(const char *, UI_STORAGE *, UI_STORAGE_OBJECT *);

protected:
    char *text;

    virtual EVENT_TYPE DrawItem(const UI_EVENT &event, EVENT_TYPE ccode);
};
```

This class uses one member variable:

- *text* is a pointer to the text to be displayed on the help bar field.

Using HELP_BAR

The HELP_BAR class is defined to occupy the bottom line of its parent window. When the parent window is moved or sized, the help bar will also be moved and sized so that it still occupies the bottom line of the window. This feature is achieved by setting the WOF_NON_FIELD_REGION flag on the help bar object. (This will be discussed later on in this chapter.)

The help bar displays textual information when it receives a request via its **Information( )** function. Window objects, in this tutorial, use a user function to send help display requests to the help bar. Each window object calls the following user function (contained in **HELPBAR.CPP**):

```
// Help bar message indices.
enum HELP_BAR_MESSAGE
{
    HELP_FIRST_NAME = 1,
    HELP_LAST_NAME,
    HELP_ADDRESS,
    HELP_PHONE,
    HELP_CLOSE_WINDOW,
    HELP_EXIT
};
```

```
// User function to set help bar information.
EVENT_TYPE SetHelp(UI_WINDOW_OBJECT *object, UI_EVENT &,
    EVENT_TYPE ccode)
{
    // Declare the help message/context pairs.
    static struct HELP_PAIR
    {
        UI_HELP_CONTEXT helpContext;
        char *message;
    } helpMessageTable[] =
    {
        { HELP_FIRST_NAME,      "First name of customer" },
        { HELP_LAST_NAME,       "Last name of customer" },
        { HELP_ADDRESS,         "Address of customer" },
        { HELP_PHONE,           "Phone number of customer" },
        { HELP_CLOSE_WINDOW,    "This will close the window" },
        { HELP_EXIT,            "This will exit the program" },
        { 0, 0 } // End of array.
    };

    // If you are not setting or clearing the help bar then just exit.
    if (ccode != S_CURRENT && ccode != S_NON_CURRENT)
        return (0);

    // Find the parent window.
    for (UI_WINDOW_OBJECT *parentWindow = object; parentWindow->parent; )
        parentWindow = parentWindow->parent;

    // Get the help bar.
    UI_WINDOW_OBJECT *helpBar =
        (UI_WINDOW_OBJECT *)parentWindow->Information(GET_STRINGID_OBJECT,
            "HELP_BAR");

    // If there was a help bar then set or clear its message.
    if (helpBar)
    {
        // Set default message to clear bar.
        char *message = "";

        if (ccode == S_CURRENT)
        {
            // Get the message associated with the help context.
            for (int i = 0; helpMessageTable[i].helpContext; i++)
                if (object->helpContext == helpMessageTable[i].helpContext)
                {
                    message = helpMessageTable[i].message;
                    break;
                }
        }

        // Update the help bar text.
        helpBar->Information(SET_TEXT, message, ID_HELP_BAR);
    }
    return (0);
}
```

The user function should perform the following essential steps:

1—Find the parent window. In order for the calling window object to obtain a pointer to the help bar (without the use of global or special pointers), it is necessary to get a pointer to the parent window. All windows maintain a list of their sub-objects. Since the calling object and the help bar share the same parent window, we can trace the objects' *parent* pointer until it points at the parent window. Getting a

pointer to the parent window in this manner will allow access to the help bar without the use of a global or a special help bar pointer.

2—Get the help bar. With a pointer to the parent window, we can query the window to see if a help bar object has been added. If the window contains a help bar object (i.e., it has the ID_HELP_BAR identification code) the window returns a UI_-WINDOW_OBJECT pointer to it.

3—Get the help information. If the parent window contained a help bar, get the help information associated with the calling window object. In this example, the help text is cleared when the calling window object is becoming non-current. This allows for the help bar to remain blank if the window object that will become current does not have any associated help information.

4—Send the display help request. Using the *helpBar* pointer that we set in step 2, we can set the help bar's text by calling its **Information( )** function. (**NOTE: Information( )** is inherited from UI_WINDOW_OBJECT.)

Event function—DOS

A DOS version of **Event( )** is created to enable the help bar to receive the messages which cause it to be initialized, sized or displayed when the help bar is running in DOS. Since some of the low-level messages are environment-specific, one event function is created for each of the supported environments. Other than this exception, the source code for the help bar is portable across environments. All events are passed to **UI_-WINDOW_OBJECT::Event( )**. The following event messages are also processed by **HELP_BAR::Event( )**:

S_CREATE—When this message is received, it is passed to UI_WINDOW_OBJECT to initialize the object's information. Since the help bar was declared as a non-field region, it will (by default) occupy the entire available window space. At this time, changes are made to the help bar's region (i.e., member variable *true*) so that it only occupies the bottom line of the window. This is demonstrated by the following code:

```
if (display->isText)
    true.top = true.bottom;
else
{
    true.left--; true.right++;
    true.top = ++true.bottom - display->cellHeight + 1;
}
```

S_DISPLAY_ACTIVE and **S_DISPLAY_INACTIVE**—These messages cause the help bar and its associated text, if any, to be displayed on the window. In text mode,

this is simple since it only requires setting the correct color palette and then displaying text. The following code, taken from **HELP_BAR::DrawItem( )**, shows this:

```
UI_REGION region = true;
UI_PALETTE *palette = LogicalPalette(ccode, ID_BUTTON);
DrawText(screenID, region, text, palette, TRUE, ccode);
```

In graphics mode, displaying the help bar is a little more complicated since, in addition to the text, there is also a graphical field to be drawn. The graphical field is similar in appearance to a depressed button. The following code shows how the help bar is drawn:

```
UI_PALETTE *palette = LogicalPalette(ccode, ID_BUTTON);
UI_REGION region = true;
if (FlagSet(woFlags, WOF_BORDER))
    DrawBorder(screenID, region, FALSE, ccode);
display->Rectangle(screenID, region, palette, 0, TRUE, FALSE, &clip);
region.left += display->cellWidth;
region.top += HELP_OFFSET;
region.right -= display->cellWidth;
region.bottom -= (HELP_OFFSET + 1);
palette = LogicalPalette(ccode, ID_DARK_SHADOW);
display->Line(screenID, region.left, region.bottom - 1,
    region.left, region.top, palette, 1, FALSE, &clip);
display->Line(screenID, region.left, region.top,
    region.right - 1, region.top, palette, 1, FALSE, &clip);
palette = LogicalPalette(ccode, ID_WHITE_SHADOW);
display->Line(screenID, region.right, region.top,
    region.right, region.bottom, palette, 1, FALSE, &clip);
display->Line(screenID, region.right, region.bottom,
    region.left, region.bottom, palette, 1, FALSE, &clip);
region.left += HELP_OFFSET; region.top++;
region.right -= HELP_OFFSET; region.bottom--;
palette = LogicalPalette(ccode, ID_BUTTON);
DrawText(screenID, region, text, palette, TRUE, ccode);
woStatus &= ~WOS_REDISPLAY;
```

Event function—Windows

A Windows version of **Event( )** is created for the help bar to enable it to receive the messages which cause it to be initialized, sized or displayed when the help bar is running in Windows mode. Since some of the low-level messages are environment-specific, each environment must have its own **Event( )**. This function and the **DrawItem( )** function (described below) are the only Windows specific code blocks. Otherwise, the source code for the help bar is portable across environments. All events are passed to **UI_WINDOW_OBJECT::Event( )**. The following event messages are also processed by **HELP_BAR::Event( )**:

S_INITIALIZE—This message is used to set the static pointer *_helpbarJumpInstance* to the function **HelpbarJumpProcedure( )**.

S_SIZE and **S_CREATE**—When either of these messages is received, it is passed to UI_WINDOW_OBJECT to set up the object's information. Since the help bar was declared as a non-field region, it will (by default) occupy the entire available window space. At this time, changes are made to the help bar's region (i.e., member variable *true*) so that it only occupies the bottom line of the window.

When objects are used within Windows, they must first be registered by a call to the Windows' function **RegisterObject( )**.

```
RegisterObject("HELP_BAR", "STATIC", &_helpbarOffset,
    &_helpbarJumpInstance, &_helpbarCallback, NULL);
```

The purpose of this call is to set **HelpbarJumpProcedure( )** as the function to be called, by Windows, when the help bar object is passed events. **HelpbarJumpProcedure( )** gets a pointer to the receiving object, creates a Zinc event and then passes the event to the receiving object.

```
long FAR PASCAL _export HelpbarJumpProcedure(HWND hWnd, WORD wMsg,
    WORD wParam, LONG lParam)
{
    HELP_BAR *object = (HELP_BAR *)GetWindowLong(hWnd, _helpbarOffset);
    return (object->Event(UI_EVENT(E_MSWINDOWS, hWnd, wMsg, wParam,
        lParam)));
}
```

Upon return from **object->Event( )**, the default callback function is automatically invoked to pass the message back to the default Windows procedure (i.e., **DefWindowProc**).

In addition to implementing **Event( )**, the Windows version implements a drawing routine called **DrawItem( )**. **DrawItem( )** is a virtual function that is called by the library when drawing needs to occur. This feature is different from the other environments mainly to illustrate an alternative way to provide drawing. Either way is valid and there are no real speed differences. However, using **DrawItem( )** provides a more logical division between routines. The code for **DrawItem( )** is listed below:

```
EVENT_TYPE HELP_BAR::DrawItem(const UI_EVENT &, EVENT_TYPE ccode)
{
    const int HELP_OFFSET = 1;

    if (ccode == S_REDISPLAY)
        InvalidateRect(screenID, NULL, FALSE);
    PAINTSTRUCT ps;
    HDC hDC = BeginPaint(screenID, &ps);
    RECT region;
    GetClientRect(screenID, &region);

    // Fill the background.
    HBRUSH fillBrush = CreateSolidBrush(RGB_LIGHTGRAY);
    FillRect(hDC, &region, fillBrush);
    DeleteObject(fillBrush);
```

```
        // Draw the shadow.
        region.left += display->cellWidth;
        region.top += HELP_OFFSET;
        region.right -= display->cellWidth;
        region.bottom -= (HELP_OFFSET + 1);
        HPEN darkShadow = CreatePen(PS_SOLID, 1,
            GetSysColor(COLOR_BTNSHADOW));
        SelectObject(hDC, darkShadow);
        MoveTo(hDC, region.left, region.bottom - 1);
        LineTo(hDC, region.left, region.top);
        LineTo(hDC, region.right, region.top);
        DeleteObject(darkShadow);
        HPEN lightShadow = GetStockObject(WHITE_PEN);
        SelectObject(hDC, lightShadow);
        LineTo(hDC, region.right, region.bottom);
        LineTo(hDC, region.left - 1, region.bottom);
        DeleteObject(lightShadow);

        // Draw the text.
        region.left += HELP_OFFSET; region.top++;
        region.right -= HELP_OFFSET; region.bottom--;
        SetTextColor(hDC, RGB_BLACK);
        SetBkColor(hDC, RGB_LIGHTGRAY);
        ::DrawText(hDC, (LPSTR)text, strlen(text), &region,
            DT_SINGLELINE | DT_VCENTER | DT_LEFT);
        EndPaint(screenID, &ps);
    }
```

Event function—OS/2

An OS/2 version of **Event( )** is created for the help bar to enable it to receive messages which cause it to be initialized, sized or displayed when the help bar is running under OS/2. Since some of the low-level messages are environment-specific, each environment must have its own **Event( )**. This function and the **DrawItem( )** function (described below) are the only OS/2 specific code blocks. Otherwise, the source code for the help bar is portable across environments. All events are passed to **UI_WINDOW_OBJECT::-Event( )**. The following event messages are also processed by **HELP_BAR::Event( )**:

S_CREATE and **S_SIZE**—When these messages are received, they are passed to UI_WINDOW_OBJECT to initialize the object's information. Since the help bar was declared as a non-field region, it will (by default) occupy the entire available window space. At this time, changes are made to the help bar's region (i.e., member variable *true*) so that it only occupies the bottom line of the window. If the S_CREATE or S_SIZE messages are received, the HELP_BAR object is registered with OS/2. This is demonstrated by the following code:

```
    case S_SIZE:
    case S_CREATE:
        true.top = true.bottom - display->cellHeight + 1;
        woStatus |= WOS_OWNERDRAW;
        font = FNT_SMALL_FONT;
        RegisterObject("UIW_HELPBAR", (char *)WC_STATIC, &baseCallback,
            text);
        break;
```

In addition to implementing **Event( )**, the OS/2 version implements a drawing routine called **DrawItem( )**. **DrawItem( )** is a virtual function that is called by the library when drawing needs to occur. This feature is different from the other environments mainly to illustrate an alternative way to provide drawing. Either way is valid and there are no real speed differences. However, using **DrawItem( )** provides a more logical division between routines. The code for **DrawItem( )** is listed below:

```
EVENT_TYPE HELP_BAR::DrawItem(const UI_EVENT &, EVENT_TYPE ccode)
{
    // Virtualize the display.
    UI_REGION region = true;
    display->VirtualGet(screenID, region);

    // Fill the object region.
    lastPalette = LogicalPalette(ccode, ID_WINDOW_OBJECT);
    display->Rectangle(screenID, region, NULL, 0, TRUE);

    // Draw the outer shadow.
    UI_PALETTE *outline = LogicalPalette(ccode, ID_OUTLINE);
    display->Line(screenID, region.left, region.top, region.right,
        region.top, outline);
    int xOffset = display->cellWidth;
    int yOffset = display->cellWidth / 2;
    region.left += xOffset + 1;
    region.top += yOffset + 1;
    region.right -= xOffset;
    region.bottom -= yOffset;

    // Draw the inner shadow.
    UI_PALETTE *lightShadow = LogicalPalette(ccode, ID_WHITE_SHADOW);
    UI_PALETTE *darkShadow = LogicalPalette(ccode, ID_DARK_SHADOW);
    display->Line(screenID, region.left, region.top + 1, region.left,
        region.bottom - 1, darkShadow, 1, FALSE);
    display->Line(screenID, region.left, region.top, region.right,
        region.top, darkShadow, 1, FALSE);
    display->Line(screenID, region.right, region.top + 1, region.right,
        region.bottom, lightShadow, 1, FALSE);
    display->Line(screenID, region.left, region.bottom, region.right - 1,
        region.bottom, lightShadow, 1, FALSE);
    --region;
    region.left += xOffset;

    // Draw the text.
    DrawText(screenID, region, text, NULL, FALSE, ccode);

    // Update the display.
    display->VirtualPut(screenID);
    return (TRUE);
}
```

Event function—Motif

A Motif version of **Event( )** is created for the help bar to enable it to receive messages which cause it to be initialized, sized or displayed when the help bar is running under Motif. Since some of the low-level messages are environment-specific, each environment must have its own **Event( )**. Other than this exception, the source code for the help bar is portable to the other environments supported by Zinc. All events are passed to **UI_-WINDOW_OBJECT::Event( )**.

Motif provides a text widget with the same appearance as the help bar object that we have created for the other environments. Whenever native widgets can be used, it is usually better to use them since they will be faster and provide a tighter integration with the host environment. To illustrate this integration, the help bar for Motif will be implemented using the Motif text widget. The following event messages are processed by **HELP_BAR::Event( )**:

S_SIZE and **S_CREATE**—When these messages are received, they are passed to UI_WINDOW_OBJECT to initialize the object's information. Since the help bar was declared as a non-field region, it will (by default) occupy the entire available window space. At this time, changes are made to the help bar's region (i.e., member variable *true*) so that it only occupies the bottom line of the window. Then the help bar is registered with Motif. This is demonstrated by the following code (contained in **HELPBAR.CPP**):

```
case S_SIZE:
case S_CREATE:
    true.top = true.bottom - display->cellHeight + 1;
    XtSetArg(args[nargs], XmNeditable, FALSE); nargs++;
    XtSetArg(args[nargs], XmNcursorPositionVisible, FALSE); nargs++;
    XtSetArg(args[nargs], XmNmarginHeight, 2); nargs++;
    XtSetArg(args[nargs], XmNmarginWidth, 2); nargs++;
    XtSetArg(args[nargs], XmNvalue, text); nargs++;
    RegisterObject(NULL, XmCreateText, text, ccode, TRUE);
    break;
```

S_REDISPLAY—This message causes the help bar to be re-displayed on the screen. The following code (contained in **HLPBAR.CPP**) shows how this is done:

```
case S_REDISPLAY:
    nargs = 0;
    XtSetArg(args[nargs], XmNvalue, text); nargs++;
    XtSetValues(screenID, args, nargs);
    break;
```

Enhancements

There are several enhancements that can be made to HELP_BAR to provide a different look or implementation. Some of these ideas are described below. (**NOTE:** The actual implementation of these ideas is left to the reader.)

1—Store the help context information into a **.DAT** file. Using a **.DAT** file would require the use of the UI_STORAGE and UI_STORAGE_OBJECT classes.

2—In addition to the field specific help, general help could be provided. This way, whenever a field does not have its own help context, the help bar will not be blank.

3—Bitmaps could be added to the HELP_BAR class to be displayed in addition to the text information.

4—In this tutorial, the help bar consists of a single line of text information. HELP_BAR could be modified to allow for multiple fields on the same help bar line. Some of these extra fields could be buttons that invoke a hyper-text help window.

CHAPTER 15 – VIRTUAL LIST

This tutorial shows you how to create a virtual list that presents database information to the screen (i.e., a list that gets its information from disk). When we are finished, you should understand:

- the design used to implement a virtual list
- the basic design rules that control the operation of windows and of window objects within Zinc Application Framework
- the type of information needed to initialize the UIW_WINDOW base classes.

The source code associated with this program is located in **\ZINC\TUTOR\VLIST**. It contains the following files:

VLIST.CPP—This file contains the main program loop (i.e., **UI_APPLICATION::-Main( )**) as well as the following member functions:

VIRTUAL_ELEMENT::VIRTUAL_ELEMENT( )
VIRTUAL_ELEMENT::Event( )
VIRTUAL_LIST::VIRTUAL_LIST( )
VIRTUAL_LIST::~VIRTUAL_LIST( )
VIRTUAL_LIST::Event( )
VIRTUAL_LIST::LoadRecord( )

VLIST.TXT—This file contains 100 records that are dynamically read from disk when needed by the virtual list.

VLIST.HPP—This file contains the virtual list and the element class definitions.

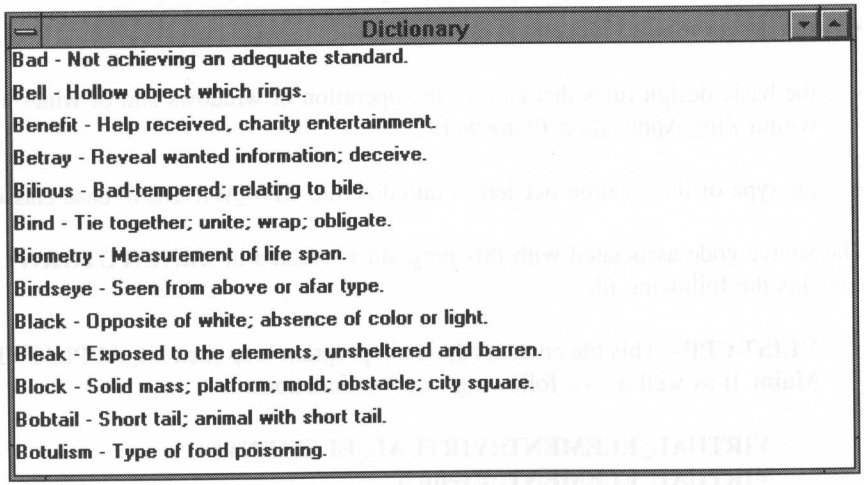
.DEF**, ***.RC**—These files are the environment specific definition and resource files required when compiling for Windows or OS/2. (**NOTE:** The **W*. files are for Windows and the **O*.*** files are for OS/2.)

***.MAK**—These files are the compiler-dependent makefiles associated with the Vlist program. (See "Chapter 1—Initializing the Library" for information on compiling for each Zinc-supported platform.)

Program execution

The operation of this program can be examined by compiling and running the application **VLIST.EXE**.

The following window should appear when you run the program:



The virtual list is actually the entire window itself. Each line of the list contains information about a different record in the database, where each record is comprised of a word and an associated definition.

You should be able to use the following keys to move within the window:

Action	Key	Description
First element	**<Ctrl+Home>**	Moves to the first database element.
Last element	**<Ctrl+End>**	Moves to the last database element.
Previous element	**<Shift+Tab>** **<↑>** **<Gray+↑>**	Moves to the previous database element. If the highlight is positioned on the first element of the window, the previous element is retrieved from the database.
Next element	**<Tab>** **<↓>** **<Gray ↓>**	Moves to the next database element. If the highlight is positioned on the last element of the window, the next element is retrieved from the database.

Page-up	**<PgUp>** **<Gray PgUp>**	Moves up one page in the database.
Page-down	**<PgDn>** **<Gray PgDn>**	Moves down one page in the database.

In addition, the left mouse button can be used to select an object.

When you are finished experimenting with the program, exit by either selecting "Close" from the system button's pop-up menu, or by pressing <Alt+F4>.

Class definitions

The virtual list window is implemented with two classes: VIRTUAL_LIST and VIRTUAL_ELEMENT. The virtual list class controls the presentation of individual virtual elements that are placed in the list. The virtual element objects represent a single database record. They are automatically created and destroyed as needed by the virtual list class. The following code shows how the VIRTUAL_LIST class is added to a parent window, then attached to the Window Manager using the + operator:

```
// Create the virtual list then attach it to the window manager.
*windowManager
    + &(*UIW_WINDOW::Generic(5, 2, 75, 12, "Dictionary")
        + new VIRTUAL_LIST("vlist.txt", 80));
```

The definition for the VIRTUAL_ELEMENT class is given below:

```
class VIRTUAL_ELEMENT : public UIW_STRING
{
    friend class VIRTUAL_LIST;
public:
    int recordNumber;

    void DataSet(VIRTUAL_ELEMENT *element)
        { UIW_STRING::DataSet(element->DataGet());
            recordNumber = element->recordNumber; }
    void DataSet(int _recordNumber, char *string)
        { UIW_STRING::DataSet(string); recordNumber = _recordNumber; }
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    VIRTUAL_ELEMENT *Next(void) { return (VIRTUAL_ELEMENT *)next; }
    VIRTUAL_ELEMENT *Previous(void) { return (VIRTUAL_ELEMENT *)previous; }

private:
    int height;

    VIRTUAL_ELEMENT(int left, int top, int width, int _height, int length) :
        UIW_STRING(left, top, width, "", length, STF_NO_FLAGS,
            WOF_NO_FLAGS), height(_height) {}
};
```

This class uses the following member variables. (Its member functions and conceptual operation will be discussed later in this chapter.)

- *recordNumber* is the number of the record in the database. Record numbers start from the number 0 and increment to one less than the total number of records in the database. For example, if the database has "n" records (e.g., 100), *recordNumber* for the last database record would be "n-1" (e.g., 99).

- *height* is the height of the VIRTUAL_ELEMENT object. *height* is used to position the object within the window.

The VIRTUAL_LIST class definition is:

```
class VIRTUAL_LIST : public UIW_WINDOW
{
public:
    VIRTUAL_LIST(const char *fileName, int _recordLength);
    ~VIRTUAL_LIST(void);

    virtual EVENT_TYPE Event(const UI_EVENT &event);

    VIRTUAL_ELEMENT *Current(void) { return (VIRTUAL_ELEMENT *)current; }
    VIRTUAL_ELEMENT *First(void) { return (VIRTUAL_ELEMENT *)first; }
    VIRTUAL_ELEMENT *Last(void) { return (VIRTUAL_ELEMENT *)last; }

    void LoadRecord(VIRTUAL_ELEMENT *element, int recordNumber);

private:
    FILE *file;
    const int recordLength;
    int numberOfRecords;
    int topRecordShowing;
    int numberShowing;
};
```

This class uses the following member variables:

- *file* is a pointer to the database. This pointer is set when the virtual list class is created.

- *recordLength* is the total number of bytes each record occupies in the database. The database we have implemented is a simple flat file with 80 character fixed-length records.

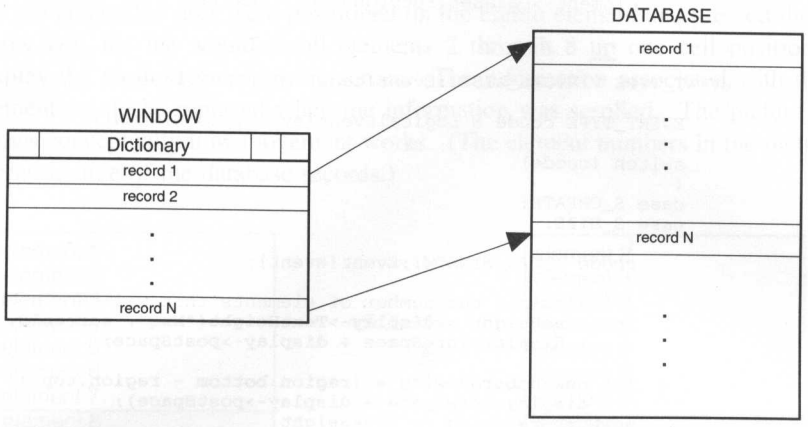
- *numberOfRecords* is the total number of records in the database.

- *topRecordShowing* is the number of the record currently displayed as the first item in the virtual list.

- *numberShowing* is the number of records that are currently visible in the list.

Conceptual operation

The conceptual operation of the virtual list can be illustrated by the following figure:



This operation can be described through the following steps:

1—The virtual list is derived from the UIW_WINDOW base class and reserves the space within its border for the list elements. When the virtual list is created, the database is opened and the total number of records is recorded.

```
VIRTUAL_LIST::VIRTUAL_LIST(const char *fileName, int _recordLength) :
    UIW_WINDOW(0, 0, 0, 0, WOF_NON_FIELD_REGION),
    recordLength(_recordLength), topRecordShowing(0), numberShowing(0)
{
    // Open the database, get the total number of records.
    file = fopen(fileName, "rb");
    fseek(file, 0L, SEEK_END);
    numberOfRecords = (int)(ftell(file) / recordLength);
}
```

2—The list creates virtual elements to fill its window space. When the list is created, it automatically determines the number of elements required to fill the screen, then constructs each element. The information associated with each element is read from disk using the **LoadRecord( )** function. This function is responsible for setting the *UIW_STRING::text* variable associated with the element. The code responsible for this initialization is shown below:

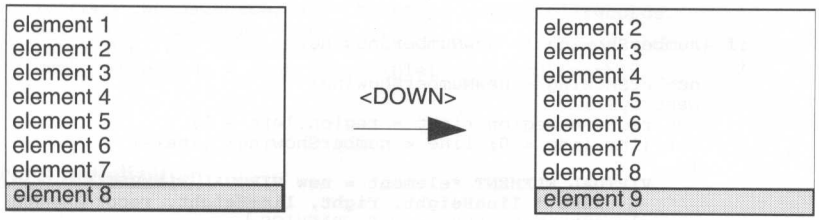
```
void VIRTUAL_LIST::LoadRecord(VIRTUAL_ELEMENT *element,
    int recordNumber)
{
    // Load the record from the file.
    if (recordNumber > numberOfRecords - 1)
        element->DataSet(-1, "");
    else
```

```
        {
            long offset = recordLength * recordNumber;
            fseek(file, offset, SEEK_SET);
            char *text = element->DataGet();
            fgets(text, recordLength - 1, file);
            text[recordLength - 1] = '\0';
            element->DataSet(recordNumber, text);
        }
    }

EVENT_TYPE VIRTUAL_LIST::Event(const UI_EVENT &event)
{
    EVENT_TYPE ccode = LogicalEvent(event, ID_WINDOW);

    switch (ccode)
    {
    case S_CREATE:
    case S_SIZE:
        {
        ccode = UIW_WINDOW::Event(event);

        // Calculate the number of elements that will fit in the list.
        int lineHeight = display->TextHeight("Mxq", screenID, font)
            + display->preSpace + display->postSpace;

        int newNumberShowing = (region.bottom - region.top +
            display->preSpace + display->postSpace);
        newNumberShowing /= lineHeight;
        if (display->isText)
            newNumberShowing++;

        // Make sure that the window is full of records.
        if (numberShowing != newNumberShowing)
        {
            numberShowing = newNumberShowing;
            Destroy();
            int right = region.right - region.left + 1;
            for (int line = 0; line < numberShowing; line++)
            {
                VIRTUAL_ELEMENT *element = new VIRTUAL_ELEMENT(0,
                    line * lineHeight, right, lineHeight,
                    recordLength + 10);
                element->woStatus |= WOS_GRAPHICS;
                LoadRecord(element, topRecordShowing + line);
                Add(element);
            }
            Event(UI_EVENT(S_REDISPLAY));
        }
        }
        break;
        .
        .
        .
    }
```

NOTE: Only those elements that are visible within the window are stored in memory. All other element information is retained on disk.

3—If the user moves to virtual list elements that are visible on the screen, the event is passed by the **VIRTUAL_LIST::Event( )** function to the base **UIW_WINDOW::-Event( )** function for processing.

If the user moves to a record that is not present on the screen, the virtual list "scrolls" all visible record information on the screen (up or down, depending on the new position selected) and reads the new record. The new record is then displayed to the screen. For example, if the first eight records of the database were visible on the screen and the user were positioned on the eighth element and pressed the down arrow key, the list would scroll elements 2 through 8 up one cell position, then display the ninth element on the screen. The information associated with the first element would be replaced when the information was scrolled. The picture below shows conceptually how movement works. (The element numbers in the picture are representative of the database records.)



4—The virtual list information is deleted when the class is deleted. This operation is performed when the window is "closed," or when the application is terminated.

VIRTUAL_ELEMENT

The VIRTUAL_ELEMENT class is derived from the base UIW_STRING class so that it can effectively present information within a window. The VIRTUAL_ELEMENT constructor is defined as private and is an in-line function. (Only the VIRTUAL_LIST class has access to the constructor by virtue of it's friend class status.)

```
class VIRTUAL_ELEMENT : public UIW_STRING
{
    friend class VIRTUAL_LIST;
private:
    VIRTUAL_ELEMENT(int left, int top, int width, int height) :
        UIW_STRING(left, top, width, "", length, STF_NO_FLAGS,
        WOF_NO_FLAGS), height(_height) {}
    .
    .
    .

}
```

The class constructor initializes information as follows:

- *left*, *top*, *width*, *height* and *length* give the size of the object in text or pixel coordinates. Earlier we presented the virtual list code (**VIRTUAL_LIST::Event( )**) that creates virtual elements.

```
// Calculate the number of elements that will fit in the list.
int lineHeight = display->Textheight("Mxq", screenID, font)
    + display->preSpace + display->postSpace;

int newNumberShowing = (region.bottom - region.top + display->preSpace +
    display->postSpace);
newNumberShowing /= lineHeight;
if (display->isText)
    newNumberShowing++;
.
.
.

if (numberShowing != newNumberShowing)
{
    numberShowing = newNumberShowing;
    Destroy();
    int right = region.right - region.left + 1;
    for (int line = 0; line < numberShowing; line++)
    {
        VIRTUAL_ELEMENT *element = new VIRTUAL_ELEMENT(0,
            line * lineHeight, right, lineHeight, recordLength + 10);
        element->woStatus |= WOS_GRAPHICS;
        LoadRecord(element, topRecordShowing + line);
        Add(element);
    }
    Event(UI_EVENT(S_REDISPLAY));
}
.
.
.
```

The actual initialization of the VIRTUAL_ELEMENT's region is handled by the base class UIW_STRING.

The two overloaded **DataSet( )** functions are used by VIRTUAL_LIST to set the information contained within the list. The first overloaded function takes another VIRTUAL_ELEMENT pointer argument. This overloaded function is used when the elements in the list are being scrolled. The second overloaded function receives the record number and a character string as arguments. This function is used by the virtual list when the elements are being read in from disk. Both of these functions are in-line, and both are presented below:

```
void DataSet(VIRTUAL_ELEMENT *element)
    { UIW_STRING::DataSet(element->DataGet());
      recordNumber = element->recordNumber; }
void DataSet(int _recordNumber, char *string)
    { UIW_STRING::DataSet(string); recordNumber = _recordNumber; }
```

VIRTUAL_LIST

The virtual list class is derived from the base class UIW_WINDOW. This derivation allows the list to inherit many of the field movement features implemented by the base class (e.g., moving up and down within the window).

The virtual list constructor initializes the database and base class information. Its definition is shown below:

```
VIRTUAL_LIST::VIRTUAL_LIST(const char *fileName, int _recordLength) :
    UIW_WINDOW(0, 0, 0, 0, WOF_NON_FIELD_REGION),
    recordLength(_recordLength), topRecordShowing(0), numberShowing(0)
{
    // Open the database, get the total number of records.
    file = fopen(fileName, "rb");
    fseek(file, 0L, SEEK_END);
    numberOfRecords = (int)(ftell(file) / recordLength);
}
```

Base class initialization

The base UIW_WINDOW class constructor is called before any class specific information is set. It requires the specification of object boundaries (the first four arguments) and the specification of any special window object flags.

In addition to the boundary information, the base UIW_WINDOW object and window manager classes set several other variables:

- the *UI_WINDOW_OBJECT* part of the class is initialized with the information passed by the UIW_WINDOW class. This includes the boundary arguments specified by our constructor as well as the default arguments specified by the UIW_WINDOW constructor. These arguments are shown below:

```
class EXPORT UIW_WINDOW : public UI_WINDOW_OBJECT, public UI_LIST
{
public:
    UIW_WINDOW(int left, int top, int width, int height,
        WOF_FLAGS woFlags = WOF_NO_FLAGS,
        WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS,
        UI_HELP_CONTEXT helpContext = NO_HELP_CONTEXT,
        UI_WINDOW_OBJECT *minObject = NULL);
    .
    .
    .

};

UIW_WINDOW(int left, int top, int width, int height,
    WOF_FLAGS woFlags, WOAF_FLAGS woAdvancedFlags,
    UI_HELP_CONTEXT helpContext, UI_WINDOW_OBJECT *minObject) :
    UI_WINDOW_OBJECT(left, top, width, height, _woFlags,
    _woAdvancedFlags), UI_LIST(), support(), clipList(),
    wnFlags(WNF_NO_FLAGS)
```

```
{
    .
    .
    .
}
```

Member initialization

The remaining part of the constructor initializes the specific member information associated with the VIRTUAL_LIST class:

- *file* is set to point to the file opened using the **fopen( )** function call. This file is opened for read only access.

- *recordLength* is set to the argument passed down by the constructor. In our example, the record length is 80 bytes.

- *numberOfRecords* is set to the number of fixed-length records stored in the file. This value is achieved by dividing the file length (in bytes) by the size (in bytes) of an individual record.

- *topRecordShowing* is set to 0, since we will begin reading from the front of the file.

- *numberShowing* is set to 0, since no records are currently visible.

The Event function

Various parts of the **VIRTUAL_LIST::Event( )** function were described in previous parts of this chapter. The most important thing to understand about this function is that it only overrides events that cause the presentation of information to change. All other events are passed to UIW_WINDOW for handling. The following list describes how information is overridden by the **VIRTUAL_LIST::Event( )** function:

S_CREATE and **S_SIZE** cause the virtual list to recompute the number of virtual elements that can be presented to the screen. These elements are then retrieved from disk. The message is passed to the **UIW_WINDOW::Event( )** member function for processing. This causes the list elements to be displayed to the screen.

L_DOWN and **L_NEXT** are not overridden unless the next element in the virtual list resides on disk. If the element is not present on the screen, the current element information is scrolled up in the window and the next element is retrieved from disk.

L_UP and **L_PREVIOUS** are not overridden unless the previous element in the virtual list resides on disk. If the element is not present on the screen, the current element information is scrolled down in the window and the previous element is retrieved from disk.

L_PGUP, **L_PGDN**, **L_TOP** and **L_BOTTOM** cause all of the current elements to be replaced by new elements from the disk.

The Load function

The VIRTUAL_LIST load functions allow us to read information from various parts of the database. The **VIRTUAL_LIST::LoadRecord( )** function is the only function that actually performs read operations from disk. The parameter, *recordNumber*, is used to determine which record is to be retrieved.

Enhancements

The information presented in this chapter should help you understand the operation of the UIW_WINDOW class and the implementation of a virtual list that uses many of the features of its base class but optimizes the presentation of large amounts of data. There are many variations and enhancements that could be made to the virtual list and element classes described above. Let's look at some variations you could implement to make the virtual list more powerful and flexible. (**NOTE:** The actual implementation of these ideas is left to the reader.)

1—Make the base class abstract by declaring pure virtual functions for the **LoadRecord( )** function. Doing this would allow you to read non-ascii text into the record and to display the record information in various formats.

2—Allow a buffer of records before and after the list elements presented to the screen, so that you don't need to read record information every time the bottom or top of the window is reached.

3—Scroll bars could be added to the vertical list to aid in scrolling and to show position within the list.

If you are able to envision the extensions and variations presented above, you are well on your way to understanding the operation of windows and window objects within Zinc Application Framework.

CHAPTER 16 – CUSTOMIZED DISPLAYS

This tutorial discusses the design features you should be aware of when deriving your own display classes. We will use the UI_BGI_DISPLAY library class as our example. As a result, some DOS graphics features will be presented. When you are finished with this tutorial you should understand:

- the details required to implement the Borland BGI display
- the basic design rules that control the operation of display classes used within Zinc Application Framework
- the type of information needed to initialize the base UI_DISPLAY class.

The source code associated with this program is located in **\ZINC\TUTOR\DISPLAY**. It contains the following files:

TEST.CPP—This file contains a graphics test program.

BORLAND.MAK—This is the makefile associated with the display program. You can compile **TEST.EXE**, by typing `make -fborland.mak test.exe` at the command line prompt.

In addition, the **D_BGIDSP.CPP** file, located in **\ZINC\SOURCE**, is used.

D_BGIDSP.CPP—This file contains the BGI class constructor, destructor and associated display member functions.

NOTE: The UI_BGI_DISPLAY class requires **GRAPHICS.LIB** and the associated BGI files (e.g., **EGAVGA.BGI**, **CGA.BGI** or **HERC.BGI**). These files are all provided with the Borland compiler. However, Zinc also provides the **.BGI** files in **\ZINC\BIN**. If you do not have the Borland compiler, it is still recommended that you read this tutorial so that you understand the theory and implementation details of Zinc Application Framework display classes.

Conceptual design

The main purpose of setting up a display class object is to control all presentation made to the screen. An additional benefit of a display class is that it allows the abstraction of screen painting. For example, if we want to draw a rectangular box, all we need to do with Zinc Application Framework is to call **display->Rectangle( )**. Since the variable

display is a pointer to the abstract UI_DISPLAY class, the actual details of drawing a rectangle are left to the device dependent **Rectangle( )** function. Since all display classes are derived from the base UI_DISPLAY class, it is not necessary to know which device dependent display class (UI_BGI_DISPLAY, UI_FG_DISPLAY, UI_GRAPHICS_-DISPLAY, UI_MOTIF_DISPLAY, UI_MSC_DISPLAY, UI_MSWINDOWS_DISPLAY, UI_OS2_DISPLAY, UI_TEXT_DISPLAY, etc.) was actually created.

There are three key aspects to the implementation of display classes. The first is the definition of the UI_DISPLAY base class. This class defines the general operation of all displays but leaves the implementation to derived display classes. For example, the function responsible for drawing a rectangle is declared virtual by the base UI_DISPLAY class:

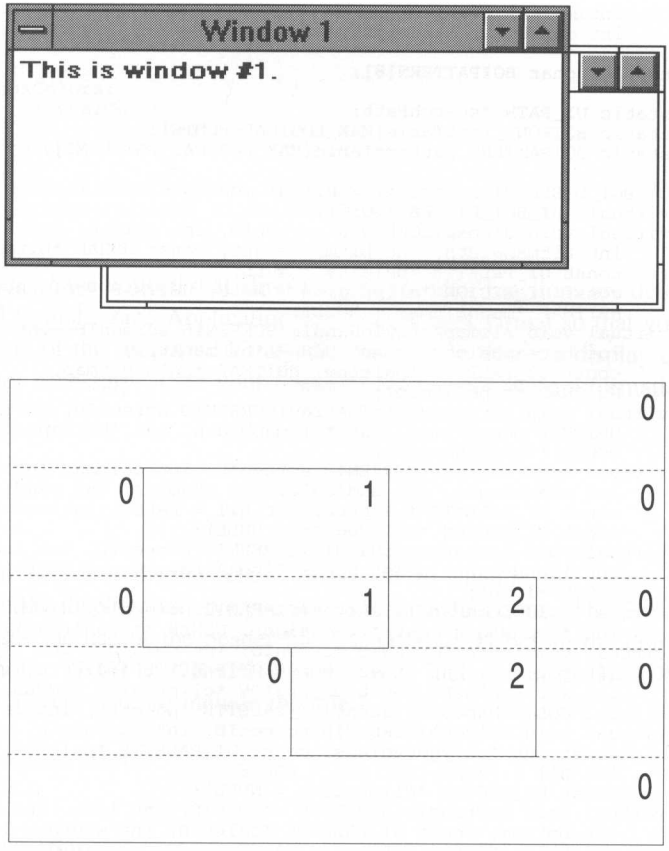
```
class EXPORT UI_DISPLAY
{
public:
    virtual void Rectangle(SCREENID screenID, int left, int top,
        int right, int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
```

The UI_DISPLAY function above is merely a stub. Thus, each derived display class is required to have an implementation for the **Rectangle( )** function (if it is to be supported).

The second aspect to the implementation of display classes is the dynamic coordinate system that can change at run-time depending on whether the program is running in various text or graphics modes. The coordinate system is left-top zero based (i.e., 0,0 is the coordinate of the left-top corner of the screen) where the right-bottom coordinates are determined by the type of display and the mode in which it is running. Some example display mode/coordinates are shown below.

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

Last, each display class maintains screen information by assigning unique identifications to rectangular regions of the screen. These rectangular regions are used for clipping and updating the display. For example, if the following two windows were attached to the screen, the display would contain several rectangular regions with different identifications:



NOTE: On environments where drawing routines are managed by the operating system (e.g., Windows, OS/2, Motif), the clipping is handled by the operating system.

Class implementation

The declaration for the BGI display class is defined in **UI_DSP.HPP**. Its declaration is almost identical to all of the other types of derived displays supported by Zinc Application Framework, with the exception of the constructor and destructor names:

```
class EXPORT UI_BGI_DISPLAY : public UI_DISPLAY, public UI_REGION_LIST
{
public:
    struct BGIFONT
    {
        int font;
        int charSize;
        int multX, divX;
```

```
    int multY, divY;
    int maxWidth, maxHeight;
};
typedef char BGIPATTERN[8];

static UI_PATH *searchPath;
static BGIFONT fontTable[MAX_LOGICAL_FONTS];
static BGIPATTERN patternTable[MAX_LOGICAL_PATTERNS];

UI_BGI_DISPLAY(int driver = 0, int mode = 0);
virtual ~UI_BGI_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);
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);
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);
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:
    int maxColors;
    char _virtualCount;
    UI_REGION _virtualRegion;
    char _stopDevice;

    void SetFont(LOGICAL_FONT logicalFont);
    void SetPattern(const UI_PALETTE *palette, int xor);
};
```

NOTE: You may have noticed that almost all the member functions are defined with the reserved word virtual. Zinc Application Framework uses virtual so that you can derive classes from any of the supported library classes. If you derive a display class, the use of virtual is not necessary. (See your compiler manual for more information about the use of virtual.)

#include files

Whenever you derive a display class, you need to include the proper header files associated with the display that give you access to their functions. The Borland graphics package requires the use of the header file **GRAPHICS.H** before the definition of the member functions. (**NOTE: GRAPHICS.H** is only included in the **BGIDSP.CPP** file, Zinc programs do not need to include this file.)

```
#include <graphics.h>
```

Display Construction

The abstract base UI_DISPLAY class constructor (called by the UI_BGI_DISPLAY constructor) requires one argument:

```
UI_DISPLAY(int isText);
```

The argument, *isText*, tells whether a text or graphics display has been created. Since we are implementing a graphics display, this value should be FALSE. This value is used by the base display to set the *UI_DISPLAY::isText* variable.

Additionally, the UI_DISPLAY class provides default settings for the following members:

- *installed* is a flag that tells whether the display has been installed. By default, this member is set to FALSE by the base UI_DISPLAY class. The derived display constructor should set this variable to be TRUE if the graphics display installs correctly.

- *isText* indicates whether the display is running in text or graphics mode. If *isText* is TRUE, the application is running in text mode. Otherwise, the application is running in graphics mode. This variable is set by passing either TRUE or FALSE to the base class.

- *isMono* is a flag that tells whether the display is operating in monochrome mode.

- *cellWidth* and *cellHeight* are width and height values of a cell coordinate. If the application is running in text mode, *cellWidth* and *cellHeight* are 1. Otherwise, the value of *cellWidth* and *cellHeight* is determined by the graphics mode and default font size. For example, the UI_BGI_DISPLAY class constructor sets *cellWidth* to 7 and *cellHeight* to 23.

- *columns* and *lines* tell how many physical columns or lines are on the display. The following table shows the values for the BGI implementation for *columns* and *lines*:

Display mode	columns	lines
Text 80 column x 25 line	80	25
Text 40 column x 25 line	40	25
Text 80 column x 43 line	80	43
Text 80 column x 50 line	80	50
CGA 320 column x 200 line	320	200
MCGA 320 column x 200 line	320	200
EGA 640 column x 350 line	640	350
VGA 640 column x 480 line	640	480

- *preSpace* denotes the size (in pixels) of the white space between the top border of a string field and the tallest character. By default, *preSpace* is set to 2.

- *postSpace* denotes the size (in pixels) of the white space between the bottom border of a string field and the lowest character. By default, *postSpace* is set to 2.

- *miniNumeratorX* and *miniDenominatorX* are values used to determine the width of a mini-cell. *miniNumeratorX* is set to 1 and *miniDenominatorX* is set to 10. (These values default to 1/10th of a cellwidth.) Mini-cells provide for more precise positioning of objects and are available in graphics modes only.

- *miniNumeratorY* and *miniDenominatorY* are values used to determine the height of a mini-cell. *miniNumeratorY* is set to 1 and *miniDenominatorY* is set to 10. (These values default to 1/10th of a cellheight.) Mini-cells provide for more precise positioning of objects and are available in graphics modes only.

- *backgroundPalette* is a pointer to the background color palette. This static pointer is initialized to point to the UI_PALETTE structure, *__backgroundPalette*, contained in **G_DSP.CPP**.

- *xorPalette* is a pointer to the XOR color palette. This static pointer is initialized to point to the UI_PALETTE structure, *__xorPalette*, contained in **G_DSP.CPP**.

- *colorMap* is a pointer to the normal color palette. This static pointer is initialized to point to the UI_PALETTE structure, *__colorMap*, contained in **G_DSP.CPP**.

After the base class initialization is complete, we must initialize any display-specific information. A listing of the UI_BGI_DISPLAY constructor is shown below. (**NOTE:** The step identifiers to the right are not part of the actual code.)

```
UI_BGI_DISPLAY::UI_BGI_DISPLAY(int driver, int mode) :
    UI_DISPLAY(FALSE)
{
    .
    .
    .

    // Register the system, dialog and small fonts that were linked in.
    BGIFONT BGIFont = {0, 0, 1, 1, 1, 1, 0, 0 };                    (Step 1)
    BGIFont.font = registerfarbgifont(SmallFont);
    if (BGIFont.font >= 0)
    {
        BGIFont.charSize = 0;
        BGIFont.maxWidth = 10;
        BGIFont.maxHeight = 11;
        UI_BGI_DISPLAY::fontTable[FNT_SMALL_FONT] = BGIFont;
    }
    BGIFont.font = registerfarbgifont(DialogFont);
    if (BGIFont.font >= 0)
    {
        BGIFont.charSize = 0;
        BGIFont.maxWidth = 11;
        BGIFont.maxHeight = 11;
        UI_BGI_DISPLAY::fontTable[FNT_DIALOG_FONT] = BGIFont;
    }
    BGIFont.font = registerfarbgifont(SystemFont);
    if (BGIFont.font >= 0)
    {
        BGIFont.charSize = 0;
        BGIFont.maxWidth = 11;
        BGIFont.maxHeight = 13;
        UI_BGI_DISPLAY::fontTable[FNT_SYSTEM_FONT] = BGIFont;
    }

    // Find the type of display and initialize the driver.          (Step 2)
    if (driver == DETECT)
        detectgraph(&driver, &mode);
    int tDriver, tMode;

    // Use temporary path if not installed in main().
    int pathInstalled = searchPath ? TRUE : FALSE;                  (Step 3)
    if (!pathInstalled)
        searchPath = new UI_PATH;
    const char *pathName = searchPath->FirstPathName();
```

```
    do
    {
        tDriver = driver;
        tMode = mode;
        initgraph(&tDriver, &tMode, pathName);
        pathName = searchPath->NextPathName();
    } while (tDriver == -3 && pathName);
    if (tDriver < 0)
        return;
    driver = tDriver;
    mode = tMode;

    // Delete path if it was installed temporarily.
    if (!pathInstalled)
    {
        delete searchPath;
        searchPath = NULL;
    }

    columns = getmaxx() + 1;                                    (Step 4)
    lines = getmaxy() + 1;
    maxColors = getmaxcolor() + 1;

    // Fill the screen according to the specified palette.      (Step 5)
    SetFont(FNT_DIALOG_FONT);
    cellWidth = (fontTable[FNT_DIALOG_FONT].font == DEFAULT_FONT) ?
        TextWidth("M", ID_SCREEN, FNT_DIALOG_FONT) :    // Bitmap font.
        TextWidth("M", ID_SCREEN, FNT_DIALOG_FONT) - 2; // Stroked font.
    cellHeight = TextHeight(NULL, ID_SCREEN, FNT_DIALOG_FONT) +
        preSpace + postSpace + 4 + 4;  // 4 above and 4 below the text.
    SetPattern(backgroundPalette, FALSE);
    setviewport(0, 0, columns - 1, lines - 1, TRUE);
    bar(0, 0, columns - 1, lines - 1);

    // Define the screen display region.
    Add(NULL, new UI_REGION_ELEMENT(ID_SCREEN, 0, 0, columns - 1,
        lines - 1));
    installed = TRUE;
}
```

The main steps in this initialization are:

1—The first step is to register any fonts that will be linked into the program (e.g., we defined the system, dialog and small fonts). These fonts are contained in the **.CHR** files in **\ZINC\SOURCE**. The fonts can be modified with the Borland font editor and must be compiled with the Borland utility **BGI2OBJ.EXE**. Once in **.OBJ** files, the fonts can be linked into the user application. (**NOTE:** These fonts are automatically linked into **DOS_BGI.LIB**.)

2—The second step determines what type of display can be created. In the Borland graphics library this is done by calling **detectgraph( )**. The *driver* and *mode* arguments of the constructor allow the programmer to override this default detection.

3—The third step required to set up the Borland graphics package is to find the associated graphics driver. The current working directory is the first place to be searched, the second is the originating directory of the application, and finally the UI_PATH class object is used to search the directories specified by the environment

variable "PATH." If the driver cannot be found, initialization ends, with the *installed* flag remaining FALSE. Otherwise, the graphics display is initialized and the process continues to the third step.

4—This step sets up *columns*, *lines* and *maxColors* variables. A description of these variables was discussed previously in this chapter.

5—The final step requires us to set up the default font, initialize *cellWidth* and *cellHeight* fill the background screen, and define the new display region (i.e., entire screen). Since the display was installed, *installed* is set to TRUE.

Display Destructor

The class destructor for UI_BGI_DISPLAY is straight-forward. If the display was installed, it must be un-installed by calling **closegraph( )**, which restores the screen.

```
UI_BGI_DISPLAY::~UI_BGI_DISPLAY(void)
{
    // Restore the display.
    if (installed)
        closegraph();
}
```

Paint Member Functions

All painting functions work under a set of similar principles. To illustrate these principles we will examine the **UI_BGI_DISPLAY::Rectangle( )** function. (**NOTE:** The step identifications to the right are not part of the actual code.)

```
void UI_BGI_DISPLAY::Rectangle(SCREENID screenID, int left, int top,
    int right, int bottom, const UI_PALETTE *palette, int width, int fill,
    int xor, const UI_REGION *clipRegion)
{
    // Assign the rectangle to the region structure.                (step 1)
    UI_REGION region, tRegion;
    if(!RegionInitialize(region, clipRegion, left, top, right, bottom))
        return;

    // Draw the rectangle on the display.
    int changedScreen = FALSE;
    for (UI_REGION_ELEMENT *dRegion = First(); dRegion;              (step 2)
        dRegion = dRegion->Next())
        if (screenID == ID_DIRECT ||
            (screenID == dRegion->screenID &&
            dRegion->region.Overlap(region, tRegion)))
        {
            if (screenID == ID_DIRECT)
                tRegion = region;
```

```
            if (!changedScreen)
            {
                changedScreen = VirtualGet(screenID, region.left,
                    region.top, region.right, region.bottom);
                SetPattern(palette, xor);
            }
            setviewport(tRegion.left, tRegion.top, tRegion.right,    (step 3)
                tRegion.bottom, TRUE);
            if (fill && xor)                    // Patch for Borland bar() xor bug.
            {
                for (int i = 0; i < tRegion.right - tRegion.left; i++)
                    line(i, top - tRegion.top, i, bottom - tRegion.top);
            }
            else if (fill)
                bar(left - tRegion.left, top - tRegion.top,
                    right - tRegion.left, bottom - tRegion.top);
            for (int i = 0; i < width; i++)
                rectangle(left - (tRegion.left - i), top -
                    (tRegion.top - i), right - (tRegion.left + i),
                    bottom - (tRegion.top + i));
            if (screenID == ID_DIRECT)
                break;
        }
    // Update the screen.                                        (step 4)
    if (changedScreen)
        VirtualPut(screenID);
}
```

1—The first step for any painting function is to set up the desired region that is to be painted on the screen. In the case of the **Rectangle( )** function, the programmer can specify up to two regions. The first region is given by four coordinates: *left*, *top*, *right* and *bottom*. This is the region where the programmer wants to draw the rectangle or fill region. The second region is specified by *clipRegion*. This region is used to describe a constraining screen region where the drawing should be clipped. The clip region is useful within Zinc Application Framework because unique screen identifications (described below) are only set up at the window level. Thus a window may contain several different sub-fields (e.g., buttons, title-bar, border) but all the objects share the same identification. The way to ensure that one sub-object does not draw over another sub-object is by specifying a *clipRegion* that is the *true* coordinates of the object that wants to paint to the screen. The object's true screen coordinates are contained in the public *UI_WINDOW_OBJECT::true*.

2—In the conceptual discussion of display classes, we saw how the display keeps track of reserved areas on the screen. The second step of each paint routine is used to determine which areas of the screen have the same identification as that passed down by the *screenID* argument. This is done by walking through the list of region elements and checking their identifications with that specified by *screenID*. If the identifications match, and if there is overlap between the screen region and the region specified by the programmer, the third step is executed. The best way to do this "clipping" would be to set up all the clip regions at once and then paint the image. Unfortunately, the BGI graphics library does not have this multiple-clip region

capability. We must, therefore, walk through the list of regions and display the image each time an overlapping region is found.

NOTE: For operating systems that associate a handle with a window object (e.g., Windows, OS/2, Motif), *screenID* is set equal to the window handle.

3—This step performs the actual operation of drawing information to the screen. The type of low-level display calls made in this step depend on the type of function that is called (e.g., **Rectangle( )**, **Ellipse( )**, **Polygon( )**) and whether the *fill* parameter is set to TRUE or FALSE.

4—In order to make the screen drawing faster, the **VirtualGet( )** and **VirtualPut( )** functions have been added. (See "Chapter 2—UI_BGI_DISPLAY" in the *Programmer's Reference* for more details.)

Information Member Functions

There are two information functions associated with the display. **TextHeight( )** is used to get the maximum height of a string using a specific font. If the font parameter, *logicalFont*, has an entry in the font table, its associated value is returned. Otherwise, the Borland **textheight( )** function is called. **TextWidth( )** is used to get the width of the text displayed in the current font. Its operation is similar to that of **TextHeight( )**.

```
int UI_BGI_DISPLAY::TextHeight(const char *string, SCREENID,
    LOGICAL_FONT logicalFont)
{
    logicalFont &= 0x0FFF;
    SetFont(logicalFont);
    if (fontTable[logicalFont].maxHeight)
        return (fontTable[logicalFont].maxHeight);
    else if (string && *string)
        return (textheight((char *)string));
    else
        return (textheight("Mq"));
}

int UI_BGI_DISPLAY::TextWidth(const char *string, SCREENID,
    LOGICAL_FONT logicalFont)
{
    if (!string || !(*string))
        return (0);
    SetFont(logicalFont & 0x0FFF);
    int length = textwidth((char *)string);
    .
    .
    .

    return (length);
}
```

Graphic display information functions must return the width and height of a string in pixel values. In addition, the text width or height should be returned, not the cell height and cell width (defined by the *cellWidth* and *cellHeight* values).

Color mapping

Most graphics libraries have special ways of implementing colors. The UI_BGI_DISPLAY has a protected member function called **MapColor( )** that maps Zinc UI_PALETTE structure information to colors understood by the Borland graphics library. The code responsible for this conversion is shown below:

```
COLOR UI_BGI_DISPLAY::MapColor(const UI_PALETTE *palette, int foreground)
{
    // Match the color request based on the type of display.
    if (maxColors == 2)
        return (foreground ? palette->bwForeground : palette->bwBackground);
    else if (maxColors < 16)
        return (foreground ? palette->grayScaleForeground :
            palette->grayScaleBackground);
    return (foreground ? palette->colorForeground :
        palette->colorBackground);
}
```

If you derive a display class from a different library package, you will need to write a map function for your display.

Conclusion

You should now have a basic understanding of the display operation within Zinc Application Framework. If you want to support additional displays, use this tutorial as a template for your implementation.

Remember, each graphics library has its own way of doing things. Even though you are at a very low-level in the library, you still need to understand the operation of the whole system. The main things to remember are to be very consistent in your implementation, make sure that you set up the clip regions properly, and be sure that you understand where you really want to paint an image.

SECTION V
PORTABILITY ISSUES

CHAPTER 17 – MULTI-LANGUAGE PROGRAMS

One of Zinc Application Framework's strongest points is its ability to be used across many different platforms. In the previous chapters we have seen examples of programs being run on different operating systems, being compiled using different compilers, and being run using different graphics modes and displays. In this chapter, we will take the concept of portability one step further in order to achieve language portability.

After studying this tutorial you should understand:

- how simple planning can make programs language-independent
- additional uses for the Zinc data file and Zinc Designer
- international uses for Zinc Application Framework

The source code associated with this program is located in **\ZINC\TUTOR\LANGUAGE**. It contains the following files:

LANG.CPP—This file contains the main program loop (i.e., **UI_APPLICATION::-Main( )**) as well as the following functions:

INTL_WINDOW::INTL_WINDOW( )
INTL_WINDOW::New( )

LANG_WIN.DAT—This file contains the window objects (created with Zinc Designer) used in this example.

LANG_WIN.CPP—This file contains the object table and user table for the objects stored in **LANG_WIN.DAT**. This file was generated by Zinc Designer.

LANG_WIN.HPP—This file contains the numberIDs and help context indices for the objects stored in **LANG_WIN.DAT**. This file was generated by Zinc Designer.

.DEF**, ***.RC**—These files are the environment specific definition and resource files required when compiling for Windows or OS/2. (**NOTE:** The **W*. files are for Windows and the **O*.*** files are for OS/2.)

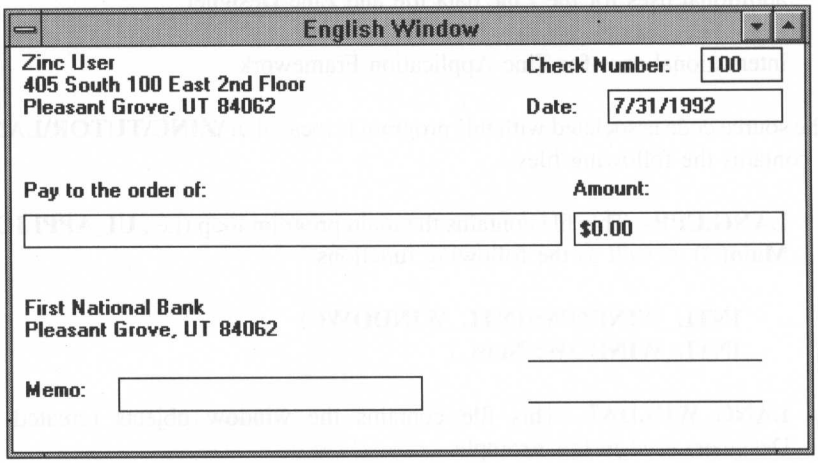
***.MAK**—These files are the compiler-dependent makefiles associated with the Lang program. (See "Chapter 1—Initializing the Library" for information on compiling for each Zinc-supported platform.)

NOTE: This tutorial program makes use of features of the UI_INTERNATIONAL class that are currently unavailable in the Unix environment. Thus, the Lang program cannot be compiled for Motif.

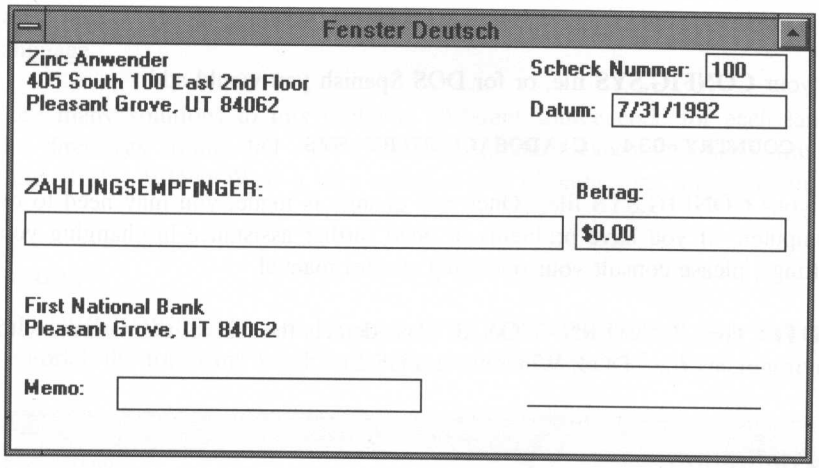
Program execution

The operation of this program can be examined by compiling and running the application **LANG.EXE**. This program will run in either English, German or Spanish depending on the current country setting in your system configuration file (e.g., **CONFIG.SYS** for the DOS environment). One of the following windows should appear:

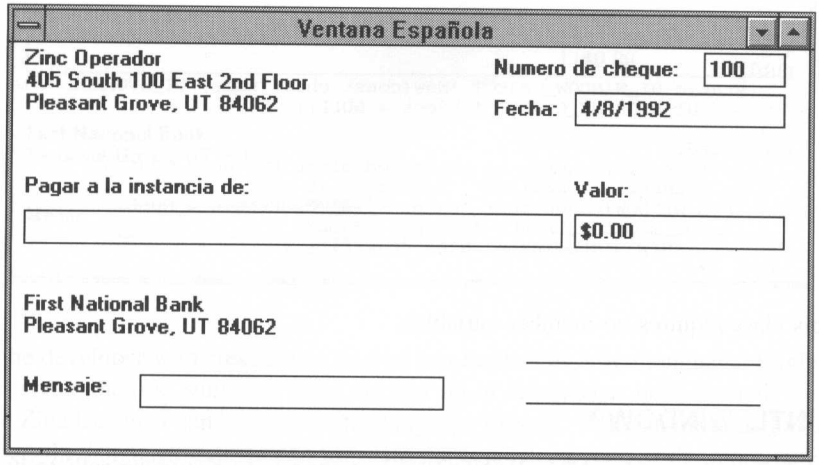
In English:



In German:



In Spanish:



In order to have this program run in one of the supported modes, it may be necessary to change your system configuration file (e.g., **CONFIG.SYS** for the DOS environment). For example,

for DOS English you would add:

```
COUNTRY=001,,C:\DOS\COUNTRY.SYS
```

to your **CONFIG.SYS** file, for DOS German you would add:

```
COUNTRY=049,,C:\DOS\COUNTRY.SYS
```

to your **CONFIG.SYS** file, or for DOS Spanish you would add:

```
COUNTRY=034,,C:\DOS\COUNTRY.SYS
```

to your **CONFIG.SYS** file. Once this change is made, you may need to re-start your computer. If you have problems or need further assistance in changing your system's settings, please consult your operating system manual.

NOTE: The UI_INTERNATIONAL class detects the native international settings on all environments (i.e., DOS, Windows and OS/2).

Class definition

The international window class is implemented with a class called INTL_WINDOW (found in **LANG.CPP**). The INTL_WINDOW definition is given below:

```
class EXPORT INTL_WINDOW : public UIW_WINDOW
{
public:
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL);

protected:
    // This constructor is protected since the name must be
    // ''internationalized'' before it is called.
    INTL_WINDOW(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL)
        : UIW_WINDOW(name, file, object) { }
};
```

This class requires no member variables.

Why INTL_WINDOW?

Upon first glance, the INTL_WINDOW class appears to just make calls to its base class UIW_WINDOW. While this is true, it does provide a very useful purpose.

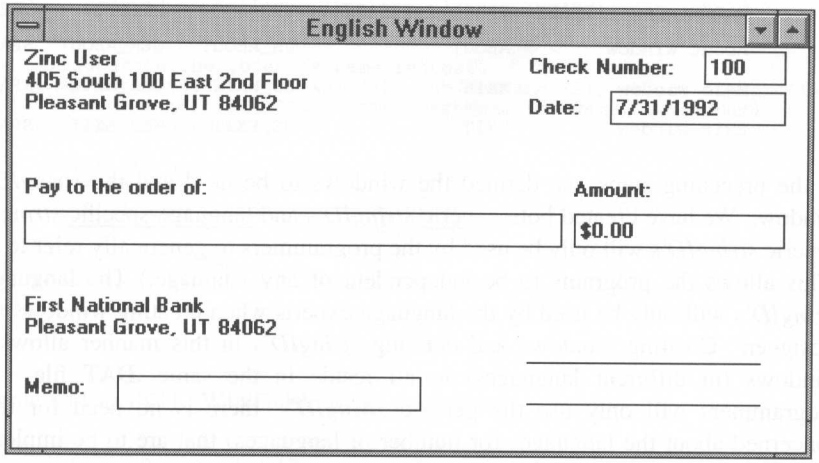
Let's suppose that you are assigned to create a program that will be shipped to several countries. The program is fairly complex, but you feel that you and your team can complete it within the allotted time period. However, you are told that it should be available in three or four different written languages. At the time of your assignment, management has not decided on all of the specifications. (Sorry for the unlikely

scenario). You are told that adding another language is "just a matter of translation" and should not affect your development schedule. Since the members of your team have little or no experience with the required languages, you are assigned a language specialist for each language.

There are many solutions to this problem: different source code for each language, compiler directives around I/O calls (requiring different executable files), translation tables, etc. INTL_WINDOW is a very compact way of addressing this problem.

Design issues

The INTL_WINDOW class is used to load UIW_WINDOW objects from a **.DAT** file. In this tutorial, the following window was created using Zinc Designer:



Since the developer who created this tutorial has English as a first language, English was used to create the first window. Once the first set of screens has been created, the **.DAT** file and Zinc Designer can be given to the language experts. After an introduction to Zinc Designer, the language experts can translate the English screens to the other languages. (Currently these are limited to those languages whose letters are part of the ANSI character set.)

Since each window in the **.DAT** file is accessed by its *stringID*, it seems logical that this variable can be used to help distinguish between the languages as well as the windows. Before any development is done or any windows are created, some planning needs to be done. Consider the following steps:

1—Make a list of all windows and their *stringID*'s that will be included in the final product. (This list is dynamic, but should be accurate.) For example:

Window description	generic stringID
About window	ABOUT
Main window	MAIN
Exit window	EXIT

2—Create a short language identifier for each language. This identifier will serve as a prefix to the *stringID* for each of the windows. In this tutorial, the language identifiers are: **ENG_** for English, **GER_** for German and **SPA_** for Spanish.

3—Now assign the *stringID*'s for each window to be created:

Window description	generic *stringID*	English	German	Spanish
About window	ABOUT	US_ABOUT	GER_ABOUT	SPA_ABOUT
Main window	MAIN	US_MAIN	GER_MAIN	SPA_MAIN
Exit window	EXIT	US_EXIT	GER_EXIT	SPA_EXIT

In the preceding steps, we defined the windows to be used and the *stringID* for each window. We have created both generic *stringID*'s and language-specific *stringID*'s. The generic *stringID*'s will only be used by the programmers to generically refer to a window. (This allows the programs to be independent of any language.) The language-specific *stringID*'s will only be used by the language experts when creating windows using Zinc Designer. Creating windows and defining *stringID*'s in this manner allows the same windows (in different languages) to all reside in the same **.DAT** file. Since the programmers will only use the generic *stringID*'s, there is no need for them to be concerned about the languages (or number of languages) that are to be implemented.

Using INTL_WINDOW

INTL_WINDOW is designed to load a window from a **.DAT** file. As a result, only two member functions are needed: **INTL_WINDOW::New( )** and **INTL_WINDOW::-INTL_WINDOW** (found in **LANG.CPP**). The constructor is used only to call the base class' constructor, **UIW_WINDOW::UIW_WINDOW( )**, where all the window-specific routines and data are kept. The **New( )** function, which does all the country translation, is listed below:

```
UI_WINDOW_OBJECT *INTL_WINDOW::New(const char *name, UI_STORAGE *file,
    UI_STORAGE_OBJECT *object)
{
```

```
    // Initialize the UI_INTERNATIONAL information.
    if (!UI_INTERNATIONAL::initialized)
        UI_INTERNATIONAL::Initialize();

    // Get the current country code.
    char langID[5];
    switch (UI_INTERNATIONAL::countryCode)
    {
    case US:
        strcpy(langID, "US_");
        break;

    case SPAIN:
        strcpy(langID, "SPA_");
        break;

    case GERMANY:
        strcpy(langID, "GER_");
        break;

    default:
        // United States English is used by default.
        strcpy(langID, "US_");
        break;
    }

    // Separate the object's name request.
    char pathName[128], fileName[32], objectName[32], objectPathName[128];
    UI_STORAGE::StripFullPath(name, pathName, fileName, objectName,
        objectPathName);

    // Put the new name request back together.
    char langObjectName[330];
    sprintf(langObjectName, "%s%s~%s%s%s", pathName, fileName,
        objectPathName, langID, objectName);

    // Load the window.
    UI_WINDOW_OBJECT *obj = new INTL_WINDOW(langObjectName, file, object);

    return obj;
}
```

A sample call to INTL_WINDOW is shown below:

```
// Get the generic ''window'' from the .DAT file.
UI_WINDOW_OBJECT *window = INTL_WINDOW::New("LANG_WIN~WINDOW");
*windowManager
    + window;
```

When **INTL_WINDOW::New( )** is called, it performs the following steps:

1—Initializes the UI_INTERNATIONAL class if it has not already been initialized. The UI_INTERNATIONAL class contains the system's current country settings. (See the *Programmer's Reference* for more information regarding UI_INTERNATIONAL.)

2—*UI_INTERNATIONAL::countryCode* is checked to see if it matches any of the specified country codes. (The default is United States English.) The country codes used in this example and by UI_INTERNATIONAL are the same as the country codes used by DOS. (See the **COUNTRY** command in the DOS manual for more details.)

3—The *name* parameter is dissected in order to get the *stringID* of the window being called. This must always be the generic *stringID* (discussed above).

4—*name* is re-built as *langObjectName* so that the name of the object now includes the language-specific prefix (described above).

5—The window is loaded.

6—The loaded window is returned. If the window is not found in the **.DAT** file, a basic UIW_WINDOW structure is returned.

Conclusion

The design presented in this chapter shows that support for international languages can be added to a program without making the task more complicated for the programmer. You should now have a good understanding of how simple planning can be useful when making programs language-independent. You should also understand some additional uses for the Zinc data file and Zinc Designer, and how Zinc Application Framework can be used to create international programs.

CHAPTER 18 – INTERNATIONAL CURRENCY

In the previous chapter, we demonstrated Zinc Application Framework's ability to create international language support for an application. Continuing with the international theme, we will examine how to implement an international currency class. In the previous chapter, internationalization was achieved through the use of the UI_INTERNATIONAL settings. In this chapter we will discuss how to implement the internationalization of currency independent of the system settings (i.e., as represented by UI_INTERNATIONAL). Currency exchanges for the following countries are presented: Germany, Japan, Spain, United Kingdom and the United States.

After studying this tutorial you should understand:

- how currency values can be translated from one currency to another
- how multiple currency values can be displayed simultaneously
- how messages (i.e., events) can be used to update window objects
- how Zinc Application Framework can be used for international projects.

The source code associated with this program is located in **\ZINC\TUTOR\CURRENCY**. It contains the following files:

MONEY.CPP—This file contains the main program loop (i.e., **UI_APPLICATION::Main( )**) as well as the following functions:

UIW_INTL_CURRENCY::UIW_INTL_CURRENCY( )
UIW_INTL_CURRENCY::New( )

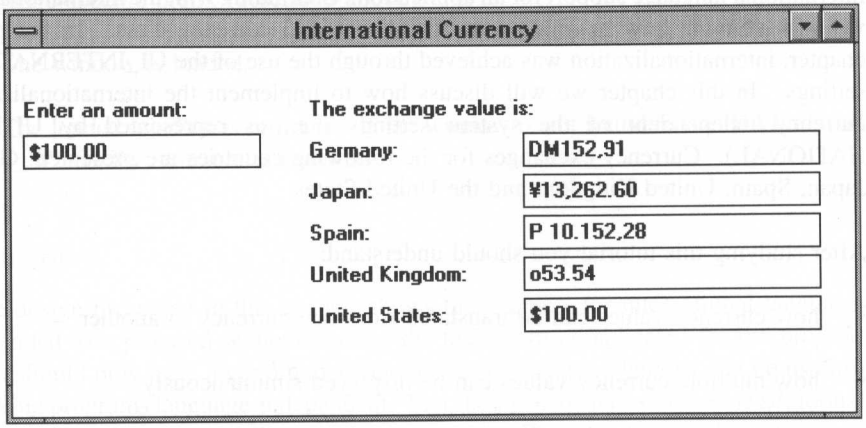
MONEY.HPP—This file contains the class definition for UIW_INTL_CURRENCY.

.DEF**, ***.RC**—These files are the environment specific definition and resource files required when compiling for Windows or OS/2. (**NOTE:** The **W*. files are for Windows and the **O*.*** files are for OS/2.)

***.MAK**—These files are the compiler-dependent makefiles associated with the Money program. (See "Chapter 1—Initializing the Library" for information on compiling for each Zinc-supported platform.)

Program execution

The operation of this program can be examined by compiling and running the application **MONEY.EXE**. The following window should appear on the screen:



In the field entitled "Enter an amount:", currency values can be typed. Pressing <ENTER> will take the value entered and translate it to the various currencies listed under the title "The exchange value is:" and then update the currency objects.

This program begins execution with US dollars as the default currency for the entry field. However, if you desire to enter a value in German Marks, press <PgDn>. Pressing either <PgUp> or <PgDn> will cycle through all of the available currencies (listed above). When you are done experimenting with the international currency program, exit either by selecting the "Close" option from the system button's menu, or by typing <Alt+F4>.

Class definition

The international currency object is implemented with a class called UIW_INTL_-CURRENCY (found in **MONEY.HPP**). The UIW_INTL_CURRENCY definition is given below:

```
class EXPORT UIW_INTL_CURRENCY : public UIW_STRING
{
public:
    static NMF_FLAGS rangeFlags;
    NMF_FLAGS nmFlags;

    UIW_INTL_CURRENCY(int left, int top, int width, UI_BIGNUM *value,
        NMF_FLAGS nmFlags = NMF_CURRENCY | NMF_DECIMAL(2),
        WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
        USER_FUNCTION userFunction = NULL, int _countrySetting = ENTRY_US);
```

```
    virtual ~UIW_INTL_CURRENCY(void) {};

    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);
    UI_BIGNUM *DataGet(void);
    void DataSet(UI_BIGNUM *_number);
    void SetCountryCode(int _countryTableEntry);
    int GetCountryCode();

protected:
    // Default current currency setting is US.
    int countryTableEntry;
    UI_BIGNUM *number;

    void ConvertToSystemSettings(char *string);
    void ConvertToLocalSettings(char *string);
};
```

This class uses the following member variables:

- *rangeFlags* is an NMF_FLAGS that gives all of the valid numeric ranges. (**NOTE:** Range checking was not implemented for this tutorial.)
- *nmFlags* gives the information on how to display and interpret the numeric information.
- *countryTableEntry* is the object's current country number (described later).
- *number* is a pointer to a UI_BIGNUM that is used to manage the low-level bignum information. If the WOF_NO_ALLOCATE_DATA flag is set, the UI_BIGNUM is allocated by the user. Otherwise, it is allocated by UIW_INTL_CURRENCY class.

UIW_INTL_CURRENCY( )

The following prototype shows the syntax of the UIW_INTL_CURRENCY class:

```
UIW_INTL_CURRENCY(int left, int top, int width, UI_BIGNUM *value,
    NMF_FLAGS nmFlags = NMF_CURRENCY | NMF_DECIMAL(2),
    WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
    USER_FUNCTION userFunction = NULL, int _countrySetting = US_ENTRY);
```

The international currency constructor returns a pointer to a UIW_INTERNATIONAL object and is created with the following parameters:

- $left_{in}$ and top_{in} is the starting position of the currency object within its parent window.
- $width_{in}$ is the width of the currency field. (The height of the currency field is determined automatically by the UIW_INTL_CURRENCY object.)

- *value*$_{in/out}$ is a pointer to a UI_BIGNUM object.

- *nmFlags*$_{in}$ gives information on how to display and interpret the numeric information. The following flags (declared in **UI_WIN.HPP**) control the general presentation of a UIW_INTL_CURRENCY class object:

  NMF_DECIMAL(2)—Displays the number with a decimal point at a fixed location. (**NOTE:** For this currency class, a two-digit decimal format has been set by the constructor.)

  ```
  $10,000.00
  £5,354.17
  DM15.290,51
  ```

  NMF_CURRENCY—Displays the number with the country-specific currency symbol. (**NOTE:** This flag has also been set by the constructor.

  ```
  $10,000.00
  £5,354.17
  DM15.290,51
  ```

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

  ```
  ($10,000.00)
  (£5,354.17)
  ```

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the currency object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_INTL_CURRENCY class object:

  WOF_AUTO_CLEAR—Automatically clears the numeric buffer if the end-user tabs to the currency field (from another window field) and presses a key (without first having pressed any movement or editing keys).

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

  WOF_INVALID—Sets the initial status of the currency field to be "invalid." Invalid numbers fit in the absolute range determined by the object type but do not fulfill all the requirements specified by the program. For example, a currency may initially be set to $200.00, but the final number, edited by the end-user, must be in the range "10.00..100.00." The initial number in this example fits the absolute requirements of a UI_INTL_CURRENCY class object but does not fit into the specified range. (**NOTE:** Range checking is not implemented in this example. This flag is available for the possibility of user enhancements.)

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

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

WOF_NO_ALLOCATE_DATA—Prevents the currency object from allocating a numeric value to store the numeric information. If this flag is set, the programmer must allocate the UI_BIGNUM (passed as the *value* parameter) that is used by the currency object.

WOF_NO_FLAGS—Does not associate any special window flags with the currency object. Setting this flag left-justifies the numeric information. This flag should not be used in conjunction with any other WOF flags.

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

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

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

- *userFunction*$_{in}$ is a programmer defined function that is called whenever:

  1—the user moves onto the field (i.e., S_CURRENT message),

  2—the <ENTER> key is pressed (i.e., L_SELECT message) or

  3—the user moves to a different field in the window or to a different window (i.e., S_NON_CURRENT).

The following arguments are associated with *userFunction* when a new currency is entered:

returnValue should be 0 if the bignum is valid. Otherwise, the programmer should call the error system with an appropriate error message and return -1.

object$_{in}$—A pointer to the UIW_INTL_CURRENCY object or the class object derived from the UIW_INTL_CURRENCY object base class. This argument must be typecast by the programmer.

event$_{in}$—A run-time message passed to the UIW_INTL_CURRENCY object.

ccode$_{in}$—The logical or system code that caused the user function to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

L_SELECT—The <ENTER> key was pressed.

S_CURRENT—The currency object is about to be edited. This code is sent before any editing operations are permitted.

S_NON_CURRENT—A different field or window has been selected. This code is sent after editing operations have been performed.

Support structures

_exchangeRate

The static array, *_exchangeRate*, contains the exchange rate factors used in this program:

```
#define MAX_COUNTRIES 5

double _exchangeRate[MAX_COUNTRIES] =
{
    1.000000, 0.654000, 0.009850, 1.867700, 0.007540
};
```

Each entry in *_exchangeRate* contains the factor by which one US dollar would be multiplied in order to convert US dollars to another currency. Country identifiers are used to specify a particular entry in the table. The following identifiers are supported:

```
// Table entry country codes.
#define ENTRY_US        0
#define ENTRY_GERMAN    1
#define ENTRY_SPANISH   2
#define ENTRY_BRITISH   3
#define ENTRY_JAPANESE  4
```

COUNTRY_INFO

Most operating systems only maintain country-specific information for the current country setting. In DOS, for example, if the current country is the United States, only country-specific information for the United States would be available from the operating system. In this tutorial, we must keep a table of the additional countries that we wish to concurrently support. A structure, countryInfo, has been created to contain country-specific currency information. The declaration for countryInfo is shown below:

```
typedef struct
{
    char *currencySymbol;
    char thousandsSeparator;
    char decimalSeparator;
} countryInfo;
```

In order to keep track of the country-specific currency information for all of the supported countries, an array, *_currencyInfo*, (of length MAX_COUNTRIES) is used. *_currencyInfo* is shown below:

```
// Country-specific currency formatting information.
countryInfo _currencyInfo[MAX_COUNTRIES] =
{
    { "$", ',', '.' },              // United States
    { "DM", '.', ',' },             // Germany
    { "₧", '.', ',' },              // Spain
    { "£", ',', '.' },              // United Kingdom
    { "¥", ',', '.' },              // Japan
};
```

When currency values are displayed, the currency formatting characters are looked-up in the *_currencyInfo* table to ensure that the proper formatting characters are used. (**NOTE:** Additional information regarding country-specific currency formats is available in your operating system manual.)

Currency translation

UIW_INTL_CURRENCY maintains its value (i.e., *number*) according to the country specifier *countryTableEntry*. For example, if *countryTableEntry* is set to ENTRY_US, *number* is assumed to be in US dollars. If *countryTableEntry* is ENTRY_BRITISH, *number* is assumed to be in British pounds.

If UIW_BIGNUM, with a user function, were used instead of creating the UIW_INTL_-CURRENCY class, the currency values could still be translated between currencies. However, since UIW_BIGNUM uses the country information obtained from the class UI_-INTERNATIONAL (where it is obtained from the operating system) to format currency values, it would only use the system's current currency settings. As a result, if the

operating system reports that the current country setting is the United States, all currency values formatted by UIW_BIGNUM will be in US dollars. On the other hand, UIW_-INTL_CURRENCY maintains a list of user-defined country information, so that formatting can be done independent of the operating system country settings. To overcome this dependency, UIW_INTL_CURRENCY implements two member functions, **ConvertToLocalSettings( )** and **ConvertToSystemSettings( )**, to format currency strings.

When a currency value is translated from one country to another, it is first converted from the local UIW_INTL_CURRENCY settings (both value and formatting) to the current system settings (as specified by UI_INTERNATIONAL). In this state, UIW_INTL_-CURRENCY can take advantage of the UI_BIGNUM functionality that is already part of the library. Once a currency value has been modified, it is converted to the current UIW_INTL_CURRENCY settings in order to be displayed. The following figures show this interaction:

O.S. has US country setting:

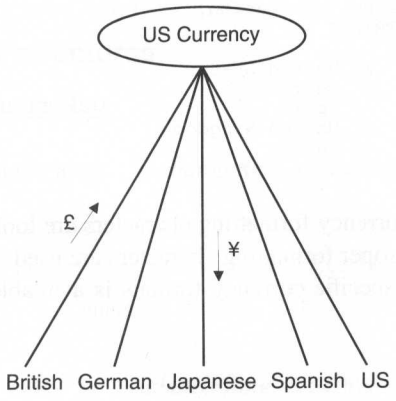


Fig. 1

O.S. has German country setting:

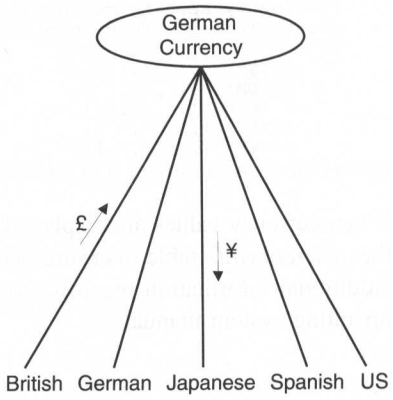


Fig. 2

ConvertToSystemSettings( )

In order to be processed correctly by UI_BIGNUM, a currency value must be in the currency format for the current country settings (as specified by UI_INTERNATIONAL). The following function, **ConvertToSystemSettings( )**, accomplishes this and is shown below:

```
void UIW_INTL_CURRENCY::ConvertToSystemSettings(char *string)
{
    char newString[NUMBER_WHOLE+NUMBER_DECIMAL+4];
    strcpy(newString, UI_INTERNATIONAL::currencySymbol);

    char *ch = string;
    int i = strlen(newString);
    while(*ch != '\0')
    {
        if (isdigit(*ch) || *ch == '-')
            newString[i++] = *ch;
        else if (*ch == _currencyInfo[countryTableEntry].decimalSeparator)
            for (int j = 0; UI_INTERNATIONAL::decimalSeparator[j] != '\0';
                j++)
                newString[i++] = UI_INTERNATIONAL::decimalSeparator[j];
        else if (*ch == _currencyInfo[countryTableEntry].thousandsSeparator)
            for (int j = 0; UI_INTERNATIONAL::thousandsSeparator[j] != '\0';
                j++)
                newString[i++] = UI_INTERNATIONAL::thousandsSeparator[j];
        ch++;
    }
    newString[i++] = '\0';

    strcpy(string, newString);
}
```

The string argument is in the local currency format and will be modified to be in the current system country format. It accomplishes this by replacing the current currency symbol, decimal separator and thousands separator with those defined by UI_-INTERNATIONAL. The input string must be long enough to hold the resulting string.

ConvertToLocalSettings( )

In order to be displayed correctly by UIW_INTL_CURRENCY, a currency value must be in the currency format currently defined by UIW_INTL_CURRENCY. The following function, **ConvertToLocalSettings( )**, accomplishes this and is shown below:

```
void UIW_INTL_CURRENCY::ConvertToLocalSettings(char *string)
{
    char newString[NUMBER_WHOLE+NUMBER_DECIMAL+4];
    strcpy(newString, _currencyInfo[countryTableEntry].currencySymbol);

    char *ch = string;
    int i = strlen(newString);
    while(*ch != '\0')
    {
        if (isdigit(*ch) || *ch == '-')
            newString[i++] = *ch;
        else if (*ch == UI_INTERNATIONAL::decimalSeparator[0])
            newString[i++] =
                _currencyInfo[countryTableEntry].decimalSeparator;
        else if (*ch == UI_INTERNATIONAL::thousandsSeparator[0])
            newString[i++] =
                _currencyInfo[countryTableEntry].thousandsSeparator;
        ch++;
    }
    newString[i++] = '\0';

    strcpy(string, newString);
}
```

This function converts the string argument from the current system currency settings to the current UIW_INTL_CURRENCY settings. This is accomplished by replacing the system currency symbol, decimal separator and thousands separator with those defined by UIW_INTL_CURRENCY. The input string must be long enough to hold the resulting string.

User interaction

DataGet( )

In order to retrieve the value of a UIW_INTL_CURRENCY object, **DataGet( )** has been defined. **DataGet( )** returns a pointer to a UI_BIGNUM containing the value of the UIW_INTL_CURRENCY object. Since UI_BIGNUM does not specify currency formatting information, you must be careful how it is used. For example, **UI_-BIGNUM::Export(string, NMF_CURRENCY)** will return a string that is formatted according to the system's current country settings. In other words, if the UIW_INTL_-CURRENCY is currently set for German currency, the screen representation would be of the form **DM2.153,37**, whereas the string representation (after invoking **DataGet( )** and **UI_BIGNUM::Export(string, NMF_CURRENCY)**) would be of the form **$2,153.37** if the current system's country setting were for the United States. If Germany were set as the current country specification for both the system (i.e., UI_INTERNATIONAL) and UIW_INTL_CURRENCY, the screen representation and the string representation would be of the same format (i.e., **DM2.153,37**).

The **DataGet( )** function is shown below:

```
UI_BIGNUM *UIW_INTL_CURRENCY::DataGet(void)
{
    // Get the string from the base class.
    UIW_STRING::DataGet();

    // Convert to current country info.
    char newString[NUMBER_WHOLE+NUMBER_DECIMAL+4];
    strcpy(newString, text);
    ConvertToSystemSettings(newString);

    // Put the new value into the UI_BIGNUM object and return it.
    number->Import(newString);
    return (number);
}
```

DataSet( )

To update an existing UIW_INTL_CURRENCY value, **DataSet( )** has been implemented. The UI_BIGNUM value is rounded to 2 decimal places (since currency values used in this tutorial only have 2 decimal places) and then it is exported to a character string. The

character string is converted to the local UIW_INTL_CURRENCY format by calling **ConvertToLocalSettings( )** and is then displayed on the string (i.e., by calling **UIW_STRING::DataSet( )**).

```
void UIW_INTL_CURRENCY::DataSet(UI_BIGNUM *_number)
{
    // Reset the number.
    if (number == _number || FlagSet(woFlags, WOF_NO_ALLOCATE_DATA))
        number = _number;
    else if (_number)
        number = new UI_BIGNUM(*_number);
    else
        number = new UI_BIGNUM(0L);

    // Round the currency value to 2 decimal places.
    *number = round(*number, 2);

    // Get the text associated with the new number.
    number->Export(text, nmFlags | NMF_COMMAS);

    // Format the number according to the local country settings.
    char newString[NUMBER_WHOLE+NUMBER_DECIMAL+4];
    strcpy(newString, text);
    ConvertToLocalSettings(newString);
    strcpy(text, newString);

    // Display the currency text.
    UIW_STRING::DataSet(text);
}
```

Key Mapping

Aside from the currency conversion aspects of this tutorial, the UIW_INTL_CURRENCY class implements re-mapping of two keys: <PgUp> and <PgDn>. These keys are used to cycle the current UIW_INTL_CURRENCY field through all of the specified currency settings. For example, if the current value displayed is $100.00 (i.e., US dollars), pressing <PgDn> would cause the currency value to change to DM152,91 (i.e., German marks).

An event map entry consists of: the object identification, the logical event, the event type and the raw code of the event to be mapped. The following key mappings were implemented as part of UIW_INTL_CURRENCY:

```
#if defined (ZIL_MSWINDOWS)
static UI_EVENT_MAP _eventTable[] =
{
    .
    .
    .

    { ID_INTL_CURRENCY,     L_PGUP,         WM_KEYDOWN,     GRAY_PGUP },
    { ID_INTL_CURRENCY,     L_PGUP,         WM_KEYDOWN,     WHITE_PGUP },
    { ID_INTL_CURRENCY,     L_PGDN,         WM_KEYDOWN,     GRAY_PGDN },
    { ID_INTL_CURRENCY,     L_PGDN,         WM_KEYDOWN,     WHITE_PGDN },

    // End of array.
    { ID_END, 0, 0, 0 }
};
```

```
#elif defined (ZIL_OS2)
static UI_EVENT_MAP _eventTable[] =
{
    .
    .
    .

    { ID_INTL_CURRENCY,     L_PGUP,         WM_CHAR,        GRAY_PGUP },
    { ID_INTL_CURRENCY,     L_PGUP,         WM_CHAR,        WHITE_PGUP },
    { ID_INTL_CURRENCY,     L_PGDN,         WM_CHAR,        GRAY_PGDN },
    { ID_INTL_CURRENCY,     L_PGDN,         WM_CHAR,        WHITE_PGDN },

    // End of array.
    { ID_END, 0, 0, 0 }
};
#elif defined (ZIL_MSDOS)
static UI_EVENT_MAP _eventTable[] =
{
    .
    .
    .

    { ID_INTL_CURRENCY,     L_PGUP,         E_KEY,      GRAY_PGUP },
    { ID_INTL_CURRENCY,     L_PGUP,         E_KEY,      WHITE_PGUP },
    { ID_INTL_CURRENCY,     L_PGDN,         E_KEY,      GRAY_PGDN },
    { ID_INTL_CURRENCY,     L_PGDN,         E_KEY,      WHITE_PGDN },

    // End of array.
    { ID_END, 0, 0, 0 }
};
#elif defined (ZIL_MOTIF)
static UI_EVENT_MAP _eventTable[] =
{
    .
    .
    .

    { ID_INTL_CURRENCY,     L_PGUP,         KeyPress,       XK_Prior },
    { ID_INTL_CURRENCY,     L_PGDN,         KeyPress,       XK_Next },

    // End of array.
    { ID_END, 0, 0, 0 }
};
#endif
```

Once this event map is defined, it must be assigned to the UIW_INTL_CURRENCY object. In UIW_INTL_CURRENCY, this is done in the constructor:

```
UIW_INTL_CURRENCY::UIW_INTL_CURRENCY(int left, int top, int width,
    UI_BIGNUM *_number, NMF_FLAGS _nmFlags, WOF_FLAGS woFlags,
    USER_FUNCTION userFunction, int _countrySetting) : UIW_STRING(left,
    top, width, NULL, _maxLength, STF_NO_FLAGS, woFlags, userFunction),
    countryTableEntry(_countrySetting), number(NULL), nmFlags(_nmFlags)
{
    UIW_INTL_CURRENCY::Information(INITIALIZE_CLASS, NULL);
    UIW_INTL_CURRENCY::DataSet(_number);
    eventMapTable = _eventTable;
}
```

Using a modified event map table eliminates the need for looking for special key events in the object's **Event( )** routine. By using this table, the **Event( )** routine simply needs to pass its event argument to **LogicalEvent( )**. **LogicalEvent( )** searches the event map table for an entry corresponding to the type of event that occurred. For example, if an

event of type E_KEY (in DOS) or WM_KEYDOWN (in Windows) containing a raw code of GRAY_PGUP were received, the logical event type, L_PGUP, would be returned. The **Event( )** function would check the logical event and perform the desired function. This example uses the library defined logical events. Consider an alternate implementation that checks for certain keystrokes and then returns a user-defined event type:

```
#if defined (ZIL_MSDOS)
static UI_EVENT_MAP _eventTable[] =
{
    .
    .
    .

    { ID_INTL_CURRENCY, L_TRANSLATE_TO_BRITISH,     E_KEY,      Ctrl_B },
    { ID_INTL_CURRENCY, L_TRANSLATE_TO_GERMAN,      E_KEY,      Ctrl_G },
    { ID_INTL_CURRENCY, L_TRANSLATE_TO_JAPANESE,    E_KEY,      Ctrl_J },
    { ID_INTL_CURRENCY, L_TRANSLATE_TO_SPANISH,     E_KEY,      Ctrl_S },
    { ID_INTL_CURRENCY, L_TRANSLATE_TO_US,          E_KEY,      Ctrl_U },

    // End of array.
    { ID_END, 0, 0, 0 }
};
.
.
.
```

Using this event map, the <Ctrl+ > keys will switch the UIW_INTL_CURRENCY value from the current country settings to the country settings defined by the logical event generated. The next section describes how the **Event( )** routine can be used to process user-defined events.

Event( )

When the programmer defines new event types (remembering that these must be above 10000) they will be interpreted as S_UNKNOWN (i.e., the are ignored) unless the programmer also creates a receiver for that event. This is the purpose of the virtual **Event( )** routine. When the **UIW_INTL_CURRENCY::Event( )** receives an event, it calls **LogicalEvent( )** to get a logical interpretation of the event, then the logical events can be used to perform desired actions. For an example, examine the following portion of **UIW_INTL_CURRENCY::Event( )**:

```
EVENT_TYPE UIW_INTL_CURRENCY::Event(const UI_EVENT &event)
{
    EVENT_TYPE ccode = LogicalEvent(event, ID_INTL_CURRENCY);
    switch (ccode)
    {
    .
    .
    .

    default:
        switch (event.type)
        {
```

```
            .
            .
            .
            case L_TRANSLATE_TO_US:
                SetCountryCode(ENTRY_US);
                break;

            case L_TRANSLATE_TO_GERMAN:
                SetCountryCode(ENTRY_GERMAN);
                break;

            case L_TRANSLATE_TO_SPANISH:
                SetCountryCode(ENTRY_SPANISH);
                break;

            case L_TRANSLATE_TO_BRITISH:
                SetCountryCode(ENTRY_BRITISH);
                break;

            case L_TRANSLATE_TO_JAPANESE:
                SetCountryCode(ENTRY_JAPANESE);
                break;

            default:
                ccode = UIW_STRING::Event(event);
                break;
            }
        }
        return ccode;
    }
```

In the above example, the user-defined events are checked to perform currency translation. By using the event map table (described above), only the logical events need to be checked since **LogicalEvent( )** handles the keystroke interpretation.

Enhancements

There are several enhancements that can be made to UIW_INTL_CURRENCY to provide a different implementation or additional features. Some of these ideas are described below. (**NOTE:** The actual implementation of these ideas is left as an exercise for the reader.)

1—Instead of "hard coding" the exchange rates, the exchange rate table could be disk-based so that the table could be more easily modified.

2—If the exchange rate table is disk-based an additional process could be created to update the table. With this implementation, a monitor could be created to check the table for updates and then notify each of the UIW_INTL_CURRENCY fields to update the currency value displayed.

3—When a user enters a new currency value, range checking could be implemented so that only those values within the specified range would be accepted.

SECTION VI
PERSISTENT OBJECTS

CHAPTER 19 – GRAPHIC OBJECTS

The next two tutorials are centered around the topic of persistence. These tutorials are written so that you can understand the underlying design and implementation of persistent objects (used for the storage and retrieval of window objects) within Zinc Application Framework.

Webster's New Universal Unabridged Dictionary has the following definitions of persistence:

1. the act of persisting; stubborn or enduring continuance, as in a chosen course or purpose.

2. a persistent or lasting quality; resoluteness, tenacity.

3. continuous existence; endurance, as of a headache.

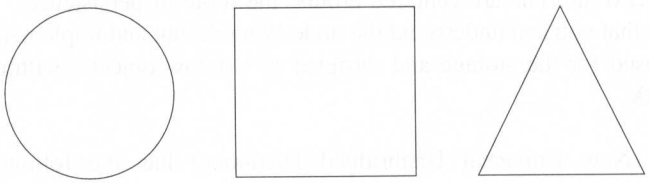
4. the continuance of an effect after the cause which first gave rise to it is removed; as persistence of vision causes visual impressions to continue upon the retina for some time.

OBJECT ORIENTATION Concepts, Languages, Databases, User Interfaces (Khoshafian & Abnous. John Wiley & Sons, Inc., 1990, pages 274-275) describes the use of persistence within computer languages:

> "The data manipulated by an object-oriented database can be either *transient* or *persistent*. Transient data is only valid inside a program or transaction; it is lost once the program or transaction terminates. . . . Persistent data is stored outside of a transaction context, and so survives transaction updates. There are several levels of persistence. Usually the term persistent data is used to indicate the *databases* that are shared, accessed and updated across transactions. . . . The least persistent objects are those that are created and destroyed in procedures data (*local data*). Next are objects that persist within the workspace of a transaction, but that are invalidated when the transaction terminates (aborts or commits). . . . The *only* type of objects that persist across transactions (and sessions for that matter) are permanent objects that typically are shared by multiple users."

Traditional C programming allows for the storage of structures and data within a file. In C++, however, class objects not only contain structural information, but also contain unique information that constitutes a class; such as member functions, single and multiple inheritance, pointers to member functions, etc.

We will use three graphic objects to introduce the concept of persistence. These objects are a circle, a rectangle and a triangle:



C and C++

The three graphic objects we chose require the following basic information:

Circle—A central screen point and radius value (*column*, *line* and *radius*)

Rectangle—Four rectangle points (*left*, *top*, *right* and *bottom*)

Triangle—Three triangular points (*left-top*, *left-bottom* and *right-bottom*)

Before we examine the use of persistent objects in C++, let's examine the code used to display our graphic objects and justify the use of C++ in our implementation. (The code is contained in **PERSIST1.C** and was compiled with the Borland compiler.)

```
#ifdef __BORLANDC__
#include <conio.h>
#include <stdlib.h>
#include <graphics.h>

main()
{
    int triangle[] = { 400, 100, 350, 200, 450, 200, 400, 100 };

    /* Initialize the screen. */
    int mode;
    int driver = DETECT;
    initgraph(&driver, &mode, 0L);
    if (graphresult() != grOk)
        exit(1);

    /* Draw the graphic objects. */
    circle(100, 150, 50);
    rectangle(200, 100, 300, 200);
    drawpoly(4, triangle);

    /* Get user input then restore the screen. */
    getch();
    closegraph();

    return (0);
}
#endif
```

The code shown above compiles with both C and C++ compilers. The conceptual flow of this program is quite easy to follow:

1—The program initializes the screen (**initgraph( )**).

2—The three objects (**circle( )**, **rectangle( )**, **drawpoly( )**) are drawn to different areas of the screen.

3—The program waits for user response from the keyboard (**getch( )**).

4—The program restores the screen (**closegraph( )**).

Although this code is very simple, its main drawbacks are that the presentation of each graphic object is compiler specific, and that there is virtually no concept of a circle, rectangle or triangle. Some of these problems can be fixed using well structured C code. Let's look at another way we could set up a program to display the graphic objects. (This code resides in **PERSIST2.C**.)

```
#define NULL     0L
#include <conio.h>
#include <graphics.h>

struct CIRCLE
{
    short column, line, radius;
};

struct RECTANGLE
{
    short left, top, right, bottom;
};

struct TRIANGLE
{
    short triangle[8];
};

void DrawCircle(struct CIRCLE *sCircle)
{
    circle(sCircle->column, sCircle->line, sCircle->radius);
}

void DrawRectangle(struct RECTANGLE *sRectangle)
{
    rectangle(sRectangle->left, sRectangle->top, sRectangle->right,
        sRectangle->bottom);
}

void DrawTriangle(struct TRIANGLE *sTriangle)
{
    drawpoly(4, sTriangle.triangle);
}

void InitializeDisplay(void)
{
    int mode;
    int driver = DETECT;
    initgraph(&driver, &mode, NULL);
```

```
    if (graphresult() != grOk)
        exit(1);
}

void RestoreDisplay(void)
{
    closegraph();
}

main()
{
    /* Initialize the screen and graphic objects. */
    struct CIRCLE circle = { 100, 150, 50 };
    struct RECTANGLE rectangle = { 200, 100, 300, 200 };
    struct TRIANGLE triangle = { 400, 100, 350, 200, 450, 200, 400, 100 };

    InitializeDisplay();

    /* Draw the objects. */
    DrawCircle(&circle);
    DrawRectangle(&rectangle);
    DrawTriangle(&triangle);

    /* Wait for user response then restore the screen. */
    getch();

    RestoreDisplay();
    return (0);
}
```

This implementation shows the following features:

Structures—Structures are used to represent the graphic objects. Each basic structure contains data information that is needed to display the object.

Display initialization—This consists of routines that hide the details of screen initialization and restoration. The functions provided above are **InitializeDisplay( )** and **RestoreDisplay( )**. The **InitializeDisplay( )** function for Zortech's Flash Graphics is as follows:

```
void InitializeDisplay(void)
{
    if (!fg_init())
        exit(1);
}
```

Object display function—Each graphic object has an associated **Draw( )** function (i.e., **DrawCircle( )**, **DrawRectangle( )** and **DrawTriangle( )**) that displays the object to the screen. The file **DRAW.C** makes library dependent function calls supporting a number of graphics libraries.

Features of C allow us to revise the code associated with the actual implementation of paint functions but provide little benefit with problems of abstraction, encapsulation and data hiding. An alternative C design (contained in **PERSIST3.C**), which helps with the problem of abstraction, is shown below:

```
struct CIRCLE
{
    short column, line, radius;
};

struct RECTANGLE
{
    short left, top, right, bottom;
};

struct TRIANGLE
{
    short triangle[8];
};

struct GRAPHIC_OBJECT
{
    int type;
    union
    {
        struct TRIANGLE triangle;
        struct RECTANGLE rectangle;
        struct CIRCLE circle;
    } graphic;
};

void DrawObject(struct GRAPHIC_OBJECT *object)
{
    if (object->type == ID_CIRCLE)
        DrawCircle(object->graphic.circle.column,
            object->graphic.circle.line, object->graphic.circle.radius);
    else if (object->type == ID_RECTANGLE)
        DrawRectangle(object->graphic.rectangle.left,
            object->graphic.rectangle.top, object->graphic.rectangle.right,
            object->graphic.rectangle.bottom);
    else if (object->type == ID_TRIANGLE)
        DrawTriangle(object->graphic.triangle.triangle);
}
```

The super structure and function defined above allows us to provide a level of abstraction on the graphic objects, but presents several new problems. First, the design is quite inflexible. For instance, if we were to define a new line object, the GRAPHIC_OBJECT structure would need to be modified and the **DrawObject( )** function would need to be modified. As more and more objects were defined, the GRAPHIC_OBJECT structure would become increasingly complex. Second, the link program, which produces executable files, would never be able to remove any graphic object's code from the executable, even if we never used the object!

The C++ solution to the problems presented above involves:

- Defining an abstract graphic object class with an abstract display routine and declaring compiler specific instances of the class.

The code implementation of these concepts is shown below. (The actual code is contained in the file **PERSIST.HPP**. These examples contain storage and retrieval functions which will be discussed later on in this chapter.)

```
#define NULL     0L
#include <conio.h>
#include <graphics.h>

class GRAPHIC_OBJECT          // Abstract graphic class.
{
public:
    virtual void Draw(void) = 0;
    static GRAPHIC_OBJECT *New(FILE *file);
    virtual void Store(FILE *file)
        { fwrite(&type, sizeof(type), 1, file); }

protected:
    short type;

    GRAPHIC_OBJECT(short _type) : type(_type) { }
    GRAPHIC_OBJECT(FILE *file, LOAD_FLAGS flags = L_NO_FLAGS)
        { if (!(flags & L_SKIP_TYPE)) fread(&type, sizeof(type), 1, file); }

private:
    struct JUMP_ELEMENT
    {
        short type;
        GRAPHIC_OBJECT *(*newFunction)(FILE *file, LOAD_FLAGS flags);
    };

    static JUMP_ELEMENT _jumpTable[];
};

class CIRCLE : public GRAPHIC_OBJECT
{
public:
    CIRCLE(short _column, short _line, short _radius) :
        GRAPHIC_OBJECT(ID_CIRCLE), column(_column), line(_line),
        radius(_radius) {   }
    CIRCLE(FILE *file, LOAD_FLAGS flags = L_NO_FLAGS) :
        GRAPHIC_OBJECT(file, flags)
        { fread(&column, sizeof(column), 1, file);
          fread(&line, sizeof(line), 1, file);
          fread(&radius, sizeof(radius), 1, file); }

    virtual void Draw(void)
        { DrawCircle(column, line, radius); }
    static GRAPHIC_OBJECT *New(FILE *file, LOAD_FLAGS flags = L_NO_FLAGS)
        { return (new CIRCLE(file, flags)); }
    void Store(FILE *file)
        { GRAPHIC_OBJECT::Store(file);
          fwrite(&column, sizeof(column), 1, file);
          fwrite(&line, sizeof(line), 1, file);
          fwrite(&radius, sizeof(radius), 1, file); }

private:
    short column, line, radius;
};
class RECTANGLE : public GRAPHIC_OBJECT
{
public:
    RECTANGLE(short _left, short _top, short _right, short _bottom) :
        GRAPHIC_OBJECT(ID_RECTANGLE), left(_left), top(_top),
        right(_right), bottom(_bottom) { }
    RECTANGLE(FILE *file, LOAD_FLAGS flags = L_NO_FLAGS) :
        GRAPHIC_OBJECT(file, flags)
        { fread(&left, sizeof(left), 1, file);
          fread(&top, sizeof(top), 1, file);
          fread(&right, sizeof(right), 1, file);
          fread(&bottom, sizeof(bottom), 1, file); }
```

```
    virtual void Draw(void)
        { DrawRectangle(left, top, right, bottom); }
    static GRAPHIC_OBJECT *New(FILE *file, LOAD_FLAGS flags = L_NO_FLAGS)
        { return (new RECTANGLE(file, flags)); }
    void Store(FILE *file)
        { GRAPHIC_OBJECT::Store(file);
          fwrite(&left, sizeof(left), 1, file);
          fwrite(&top, sizeof(top), 1, file);
          fwrite(&right, sizeof(right), 1, file);
          fwrite(&bottom, sizeof(bottom), 1, file); }

private:
    short left, top, right, bottom;
};

class TRIANGLE : public GRAPHIC_OBJECT
{
public:
    TRIANGLE(short column1, short line1, short column2, short line2,
        short column3, short line3) :
        GRAPHIC_OBJECT(ID_TRIANGLE)
        { triangle[0] = triangle[6] = column1,
          triangle[1] = triangle[7] = line1,
          triangle[2] = column2, triangle[3] = line2,
          triangle[4] = column3, triangle[5] = line3; }
    TRIANGLE(FILE *file, LOAD_FLAGS flags = L_NO_FLAGS) :
        GRAPHIC_OBJECT(file, flags)
        { fread(triangle, sizeof(triangle), 1, file); }

    virtual void Draw(void)
        { DrawTriangle(triangle); }
    static GRAPHIC_OBJECT *New(FILE *file, LOAD_FLAGS flags = L_NO_FLAGS)
        { return (new TRIANGLE(file, flags)); }
    void Store(FILE *file)
        { GRAPHIC_OBJECT::Store(file);
          fwrite(triangle, sizeof(triangle), 1, file); }

private:
    short triangle[8];
};

main()
{
    // Initialize the screen.
    InitializeDisplay();

    // Initialize the graphics objects.
    GRAPHIC_OBJECT *object[3];
    object[0] = new CIRCLE(100, 150, 50);
    object[1] = new RECTANGLE(200, 100, 300, 200);
    object[2] = new TRIANGLE(400, 100, 350, 200, 450, 200);

    // Draw the objects.
    for (int i = 0; i < 3; i++)
    {
        object[i]->Draw();
        delete object[i];
    }

    // Get user input then restore the screen.
    getch();
    RestoreDisplay();
    return (0);
}
```

The C++ solutions are manifest through the following features:

Classes—The use of class definitions allows us to encapsulate the definition and description of each graphic object. The C definition required that we define a structure with each type of object but had no way of grouping the structure and function information together. Each object's structure and functions are disjointed, except for the naming conventions we used to conceptually tie the object and function together (e.g., CIRCLE, **DrawCircle( )**).

Class scope—The use of "public," "protected" and "private" members allows us to hide the implementation details of data and display. For example, in C the structure CIRCLE contained three variables: *column*, *line* and *radius*. These variables could be seen throughout the application. In C++, however, this data is hidden. The circle is created with three arguments, but its implementation is hidden, so far as external functions are concerned.

Abstraction—The abstraction of the graphics class is accomplished through inheritance and the use of virtual and pure virtual functions. In addition to the function abstraction, class abstraction is provided by the graphic object base classes.

Encapsulation—One method of encapsulation can be seen by the late definition of the window objects. In C, we had to define the structures that would contain the graphic objects at the front of the routine; with C++ we can wait until the object is needed. Another method is provided by the class object definitions where both data and member functions are provided for the CIRCLE, RECTANGLE and TRIANGLE classes.

The main drawback of the C++ code shown above is the level of complexity placed on the definition of objects. What originally was a short program has blossomed to over 100 lines of code. This discrepancy is hard to justify when you deal with simple designs. The real benefit of what we are doing shows up when more objects are declared, or when more displays are defined.

Basic storage and retrieval

We are now ready to examine the code required to store and retrieve the graphic information. At this point, we will limit our discussion to the C++ implementation of storage.

In C++, the storage and retrieval of graphic information is quite easy to implement and to modify. The following code shows how the TRIANGLE class implements a storage and a retrieval scheme using a **Store( )** member function and overloaded class constructor. (The file **PERSIST5.CPP** contains the code required to store all of the graphics objects.)

```
class GRAPHIC_OBJECT
{
public:
    virtual void Draw(void) = 0;
    static GRAPHIC_OBJECT *New(FILE *file);
    virtual void Store(FILE *file)
        { fwrite(&type, sizeof(type), 1, file); }

protected:
    short type;

    GRAPHIC_OBJECT(short _type) : type(_type) { }
    GRAPHIC_OBJECT(FILE *file, LOAD_FLAGS flags = L_NO_FLAGS)
        { if (!(flags & L_SKIP_TYPE)) fread(&type, sizeof(type), 1, file); }

private:
    struct JUMP_ELEMENT
    {
        short type;
        GRAPHIC_OBJECT *(*newFunction)(FILE *file, LOAD_FLAGS flags);
    };

    static JUMP_ELEMENT _jumpTable[];
};

class TRIANGLE : public GRAPHIC_OBJECT
{
public:
    TRIANGLE(short column1, short line1, short column2, short line2,
        short column3, short line3) : GRAPHIC_OBJECT(ID_TRIANGLE)
        { triangle[0] = triangle[6] = column1,
            triangle[1] = triangle[7] = line1,
            triangle[2] = column2, triangle[3] = line2,
            triangle[4] = column3, triangle[5] = line3; }
    TRIANGLE(FILE *file, LOAD_FLAGS flags = L_NO_FLAGS) :
        GRAPHIC_OBJECT(file, flags)
        { fread(triangle, sizeof(triangle), 1, file); }

    virtual void Draw(void)
        { DrawTriangle(triangle); }
    static GRAPHIC_OBJECT *New(FILE *file, LOAD_FLAGS flags = L_NO_FLAGS)
        { return (new TRIANGLE(file, flags)); }
    void Store(FILE *file)
        { GRAPHIC_OBJECT::Store(file);
          fwrite(triangle, sizeof(triangle), 1, file); }

private:
    short triangle[8];
};


main()
{
    // Initialize the graphics objects.
    GRAPHIC_OBJECT *object[3];
    object[0] = new CIRCLE(100, 150, 50);
    object[1] = new RECTANGLE(200, 100, 300, 200);
    object[2] = new TRIANGLE(400, 100, 350, 200, 450, 200);

    // Store the objects.
    FILE *file = fopen("persist.dat", "wb");
    printf("Generating GRAPHICS.DAT ");
    for (int i = 0; i < 3; i++)
    {
        printf("*");
        object[i]->Store(file);
        delete object[i];
    }
```

```
    printf(" Done!\n");
    fclose(file);

    return (0);
}
```

With this implementation, each graphics object has an associated **Store( )** function and overloaded file constructor. The **Store( )** function is declared virtual by the base GRAPHIC_OBJECT class so that the derived class' store functions will be called when we store each of our objects. The code above shows how all three graphic objects can be stored to disk. The code required to read the same three objects from disk is shown below (contained in **PERSIST6.CPP**):

```
main()
{
    // Set up the graphics screen display.
    InitializeDisplay();

    // Load the graphics objects.
    FILE *file = fopen("persist.dat", "rb");
    GRAPHIC_OBJECT *object[3];
    object[0] = new CIRCLE(file);
    object[1] = new RECTANGLE(file);
    object[2] = new TRIANGLE(file);
    fclose(file);

    // Draw the objects.
    for (int i = 0; i < 3; i++)
    {
        object[i]->Draw();
        delete object[i];
    }

    // Get user input then restore the screen.
    getch();
    RestoreDisplay();
    return (0);
}
```

Abstract storage and retrieval

The only drawback with the first implementation of storage was its requirement for us to call specific class constructors (e.g., **CIRCLE::CIRCLE(file)**). Complete abstraction requires us to make three subtle but significant modifications to our design. You may recall that, up to this point, we had to know the type of object we were reading and writing. The only way to remove this restriction is to push the work on the base class GRAPHIC_OBJECT. The way we do this is to first re-define the base **Store( )** function to store the type of graphic object before the object stores its information. (The DOS version of this code is contained in **PERSIST7.CPP**, the Windows version is contained in **WPERSIST.CPP**, the OS/2 version is contained in **OPERSIST.CPP** and the Motif version is contained in **MPERSIST.CPP**)

```
class GRAPHIC_OBJECT
{
```

```
public:
    virtual void Store(FILE *file)
        { fwrite(&type, sizeof(type), 1, file); }
    .
    .
    .
};
```

Next, we need to change the derived object classes so that they call **GRAPHIC_-OBJECT::Store( )** before they store any information. An example of how this change is implemented is shown by the RECTANGLE class:

```
class RECTANGLE : public GRAPHIC_OBJECT
{
public:
    void Store(FILE *file)
        { GRAPHIC_OBJECT::Store(file);
          fwrite(&left, sizeof(left), 1, file);
          fwrite(&top, sizeof(top), 1, file);
          fwrite(&right, sizeof(right), 1, file);
          fwrite(&bottom, sizeof(bottom), 1, file); }
    .
    .
    .
}
```

The second major change we must make concerns object retrieval. This change requires us to write static **New( )** functions for all graphic objects, including the base GRAPHIC_-OBJECT class. The code below shows how the RECTANGLE and GRAPHIC_OBJECT classes are modified.

```
class GRAPHIC_OBJECT
{
public:
    virtual void Draw(void) = 0;
    virtual void Store(FILE *file)
        { fwrite(&type, sizeof(type), 1, file); }
    static GRAPHIC_OBJECT *New(FILE *file);
    .
    .
    .
}

class RECTANGLE : public GRAPHIC_OBJECT
{
public:
    RECTANGLE(FILE *file, LOAD_FLAGS flags = L_NO_FLAGS) :
        GRAPHIC_OBJECT(file, flags)
        { fread(&left, sizeof(left), 1, file);
          fread(&top, sizeof(top), 1, file);
          fread(&right, sizeof(right), 1, file);
          fread(&bottom, sizeof(bottom), 1, file); }
    static GRAPHIC_OBJECT *New(FILE *file)
        { return (new RECTANGLE(file)); }
};
```

The **New( )** functions associated with derived graphic objects are used to provide a jumping point to the object's class constructor. (In C++, we cannot get the address of a constructor directly.)

The base **GRAPHIC_OBJECT::New( )** function uses a privately defined jump table that contains four entries: one for CIRCLE, one for RECTANGLE, one for TRIANGLE and one that is used as an end-of-array indicator.

```
class GRAPHIC_OBJECT
{
    .
    .
    .

private:
    struct JUMP_ELEMENT
    {
        short type;
        GRAPHIC_OBJECT *(*newFunction)(FILE *file);
    };

    static JUMP_ELEMENT _jumpTable[];
};

GRAPHIC_OBJECT::JUMP_ELEMENT GRAPHIC_OBJECT::_jumpTable[] =
{
    { ID_CIRCLE, CIRCLE::New },
    { ID_RECTANGLE, RECTANGLE::New },
    { ID_TRIANGLE, TRIANGLE::New },
    { 0, NULL }
};
```

The derived base class **New( )** function is used as the abstract constructor and is not placed in the table. Let's look at how our code changes when we use **GRAPHIC_-OBJECT::New( )** instead of each graphic object's constructor.

```
main()
{
    // Set up the graphics screen display.
    InitializeDisplay();

    FILE *file = fopen("persist.dat", "rb");
    int fileObjects = 1;
    do
    {
        GRAPHIC_OBJECT *object = GRAPHIC_OBJECT::New(file);
        if (object)
        {
            object->Draw();
            delete object;
        }
        else
            fileObjects = 0;
    } while (fileObjects);
    fclose(file);

    // Get user input then restore the screen.
    getch();
    RestoreDisplay();
    return (0);
}
```

You can see that there is no specific reference to any particular graphics object, only the base GRAPHIC_OBJECT class. When **GRAPHIC_OBJECT::New( )** is called, it reads the type information from disk. Then it searches its jump table to find the identification found when the type was read. Once that identification is found, it calls the associated **New** function for the type.

This implementation gives us the total abstraction we wanted at a relatively small inconvenience. Let's review the steps required to implement the simple persistence for graphic objects:

1—We defined an abstract GRAPHIC_OBJECT class containing a pure virtual **Draw( )** function that is used by derived classes to paint information to the screen.

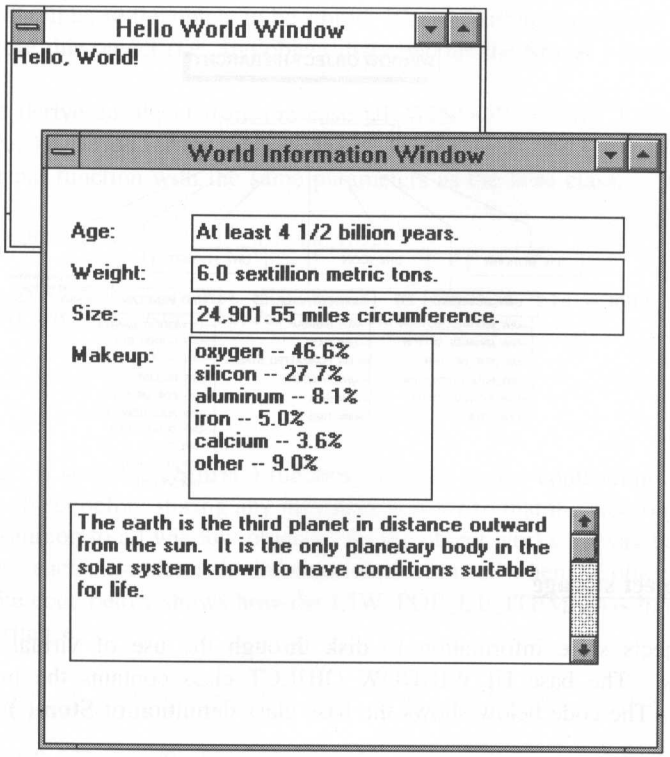
2—We defined a virtual **Store( )** function that is used to store the graphic object information. The base **GRAPHIC_OBJECT::Store( )** function just stores the object type, whereas the derived classes each store their private information.

3—We defined a static **New( )** function for each graphic object class. The derived objects' **New( )** functions are used by the base **GRAPHIC_OBJECT::New( )** function to provide jump points to the class constructors. The base **New( )** function is used by our program to provide abstract retrieval of graphic objects.

This concludes the introduction of persistent objects. The next tutorial shows you how Zinc actually implements this strategy to store and retrieve window objects that you can use in your applications.

CHAPTER 20 – ZINC WINDOW OBJECTS

The previous tutorial should give you a good introduction of how simple persistent objects are implemented. Zinc Application Framework retrieves window objects created by the interactive design tool. For example, the "Hello, World!" tutorial loaded the following two windows from disk:



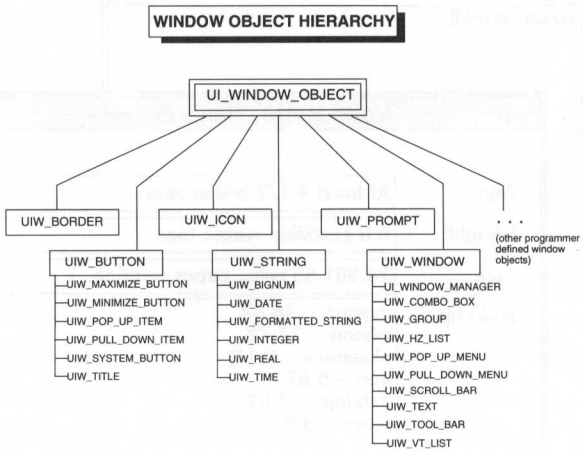
The retrieval of these two objects from disk required only two lines of constructor code. This code is shown below:

```
// Add two windows to the window manager.
*windowManager
    + new UIW_WINDOW("hello.dat~HELLO_WORLD_WINDOW")
    + new UIW_WINDOW("hello.dat~WORLD_INFORMATION_WINDOW");
```

Implementation details

The basic design of persistent objects in Zinc Application Framework is centered around three fundamental points: class object storage, class object retrieval and low-level file support.

A discussion of these points requires that you understand the window object hierarchy supported by Zinc Application Framework:



Class object storage

Zinc objects store information to disk through the use of virtual **Store( )** member functions. The base UI_WINDOW_OBJECT class contains the initial definition of **Store( )**. The code below shows the base class definition of **Store( )**:

```
class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    virtual void Store(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL);
```

The arguments passed to this function are used in the following manner:

- *name* contains the object name, or a name that contains the drive, directory, file and object path name. The name of the object is distinguished from a drive path by using the '~' (tilde) character. A full path name is required if no file is specified. Some example path names are shown below:

```
d:\zil\data\myfile~WINDOW
window.dat~HELLO
WORLD_INFORMATION_WINDOW
```

- *file* is a pointer to the file that contains the object information. The default argument NULL allows you to read an object from disk without first opening a file. In this case, **UI_WINDOW_OBJECT::Store( )** object opens the file and the top-level object closes the file.
- *object* is a pointer to the object to be stored. The default argument is NULL to allow you to store this object (i.e., the object that contains the **Store( )** function) to disk.

Whenever we derive an object from the base UI_WINDOW_OBJECT class, we define a virtual **Store( )** function for the derived object. For example, the UIW_BUTTON class contains a virtual function with the same parameters as the base class.

```
class UIW_BUTTON : public UI_WINDOW_OBJECT
{
public:
    virtual void Store(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT storeFlags = NULL);
    .
    .
    .
}
```

When an object is stored, its **Store( )** function is called by the controlling class. It calls the base class object before storing any information itself so that the base object can store information common to all window objects. As the object works its way back down the inheritance tree, each class stores the information it will need when the object is read back from disk. The code below shows how the UIW_POP_UP_ITEM class implements two levels of inheritance:

```
void UIW_POP_UP_ITEM::Store(const char *name, UI_STORAGE *directory,
    UI_STORAGE_OBJECT *file)
{
    // Store the pop-up item information.
    UIW_BUTTON::Store(name, directory, file);
    file->Store(mniFlags);
    menu.Store(NULL, directory, file);
}

void UIW_BUTTON::Store(const char *name, UI_STORAGE *directory,
    UI_STORAGE_OBJECT *file)
{
    .
    .
    .
}
void UI_WINDOW_OBJECT::Store(const char *name, UI_STORAGE *file,
    UI_STORAGE_OBJECT *object)
{
    // Write the base object information to disk.
```

```
    .
    .
    .
}
```

Class object retrieval

Window objects are loaded from disk in a manner similar to that used when storing the object. Instead of a **Store( )** function, however, the control is provided by an overloaded constructor that takes the object name, file pointer and special load flags. Here is an example of how the UIW_POP_UP_ITEM class object defines this retrieve capability:

```
class UIW_POP_UP_ITEM : public UIW_BUTTON
{
public:
    UIW_POP_UP_ITEM(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    .
    .
    .

}
```

The retrieval code works in a similar manner to that used by the store operation. For example, here is the code that loads the UIW_POP_UP_ITEM class:

```
UIW_POP_UP_ITEM::UIW_POP_UP_ITEM(const char *name, UI_STORAGE *directory,
    UI_STORAGE_OBJECT *file) : 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_NO_FLAGS)
{
    // Initialize the pop-up item information.
    UIW_POP_UP_ITEM::Load(name, directory, file);
    UI_WINDOW_OBJECT::Information(INITIALIZE_CLASS, NULL);
    UIW_BUTTON::Information(INITIALIZE_CLASS, NULL);
    UIW_POP_UP_ITEM::Information(INITIALIZE_CLASS, NULL);
    UIW_BUTTON::DataSet(text);
}
```

In this example, the pop-up item first calls the button constructor, then the **Load( )** functions are called to load the data for each inherited class.

Low-level file support

The final component is low-level file support. In Zinc, the low-level storage and retrieval operations are performed by a class called UI_STORAGE_OBJECT. This class has several member functions that are designed specifically for persistent object implementation. A partial listing of the class is given below:

```
class EXPORT UI_STORAGE_OBJECT
{
    friend class EXPORT UI_STORAGE;
public:
    int objectError;
    OBJECTID objectID;
    char stringID[32];

    UI_STORAGE_OBJECT(void);
    UI_STORAGE_OBJECT(UI_STORAGE &file, const char *name,
        OBJECTID nobjectID, UIS_FLAGS pflags = UIS_READWRITE);
    ~UI_STORAGE_OBJECT(void);
    int Load(char *value);
    int Load(UCHAR *value);
    int Load(short *value);
    int Load(USHORT *value);
    int Load(long *value);
    int Load(ULONG *value);
    int Load(void *buff, int size, int len);
    int Load(char *string, int len);
    int Load(char **string);
    void Touch(void);
    UI_STATS_INFO *Stats(void);
    UI_STORAGE *Storage(void)
    int Store(char value);
    int Store(UCHAR value);
    int Store(short value);
    int Store(USHORT value);
    int Store(long value;
    int Store(ULONG value);
    int Store(void *buff, int size, int len);
    int Store(const char *string);
};
```

The main components of this class are:

- *Load( )* is an overloaded function that allows objects to read portable information from disk.

- *Store( )* is an overloaded function that allows objects to write portable information to disk.

Conclusion

There are two catches to the implementation scheme described in this chapter. First, we need to maintain an object table that gives us a handle on the file constructors. This table is automatically created by Zinc Application Framework when you store information to disk. You just need to compile and link the file in your application.

Second, the use of virtual **Store( )** functions and the overloaded file constructor is not handled properly by current versions of compilers. They are not able to link out virtual functions that are never used. The way we get around this problem is to use *#ifdef* statements around the persistent object functions. The library source code contains the *#ifdef* directive *ZIL_PERSISTENCE*. If you re-compile the source code, with

ZIL_PERSISTENCE not defined, a version of Zinc without persistent object capabilities will be created. This may be useful for applications that are known to never use persistent objects, since it will keep the executable size smaller.

You should now be familiar with the implementation details associated with persistent objects. Their use can greatly improve the size and implementation of windows in your application.

SECTION VII
ZINC DESIGNER

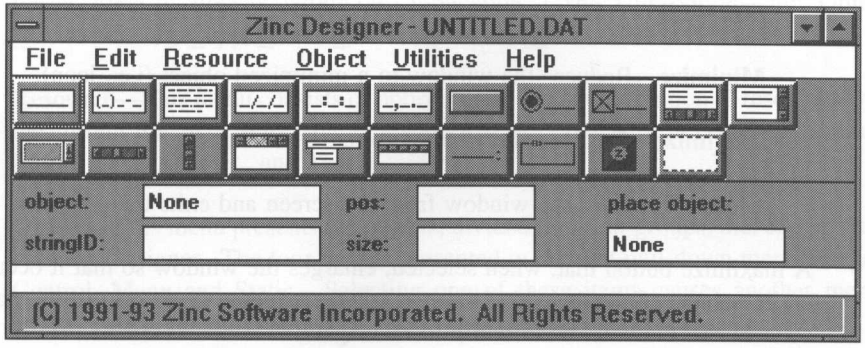
CHAPTER 21 - GETTING STARTED

Zinc Designer is an interactive tool created in order to save you, the programmer, time and effort in developing Zinc Application Framework applications. This chapter discusses the overall layout of the interactive design tool, as well as the basic procedures used in creating resources with it. (See "Chapter 3—Using Zinc Designer" for more information about using and creating resources.)

THE DESIGNER SCREEN

Overview

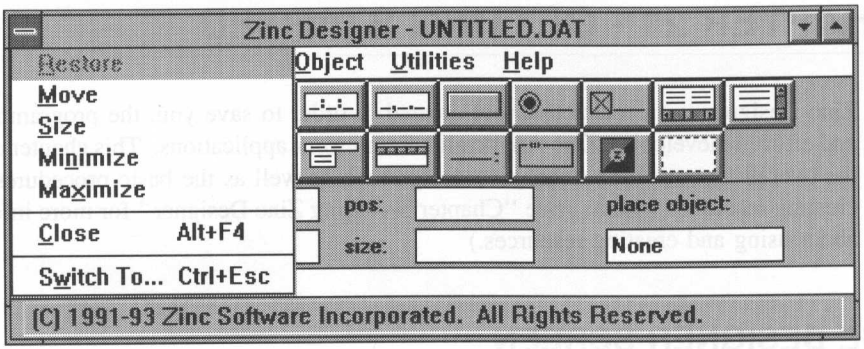
When you enter Zinc Designer, a main control window similar to the following appears:



This control window includes eight main elements:

- A title bar that identifies this window as the main control window of Zinc Designer. When a specific application is being created, the title also includes the name of the current file.

- A system button, which, when selected, displays the following pop-up menu:



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

Move—Allows the window to be moved.

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

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

Maximize—Enlarges the window to its maximum size.

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

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

- A minimize button that, when selected, reduces the window to its smallest representable form.

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

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

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

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

The menu bar

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

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

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

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

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

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

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

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

The object bar

The object bar presents some of the available window objects within Zinc Designer. It is designed to allow you to easily select these items with a mouse and then attach them

directly to your current resource. When one of the objects is selected, its name appears in the "place object" field on the status bar, where it remains until it is attached to a window, or until another object is selected from the object bar. The object is attached to a resource by positioning the cursor on the desired location and clicking the mouse button.

By default the objects on the object bar are displayed by their bitmap representations, but they can also be displayed as text only or as text and bitmaps. (See the Preferences section in "Chapter 22—File Options" for information on how to alter the object bar defaults.)

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

(See Chapters 26 through 29 for more information on creating and modifying window objects.)

The status bar

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

object—Indicates what the current object is.

stringID—Displays the string identification of the current object.

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

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

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

HOW TO START

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

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

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

3—Attach the desired objects to the window:

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

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

4—Edit the objects:

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

b) Change the default information by positioning the cursor on a field, pressing the left mouse button, and entering the new information. Flags are toggled by clicking on the associated check box. (Refer to Chapters 26 through 29 for specific information on the capabilities of each object.) When all of the necessary information is entered, select the "OK" button.

5—Save the current resource by selecting Resource | Store As... If you want to name the resource something other than the default "RESOURCE_1" enter a name for it at the "stringID" prompt. Select the "OK" button to close the window and store the resource.

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

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

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

a) Select Utilities | Help Editor.

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

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

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

e) Save the context by selecting Context | Save.

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

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

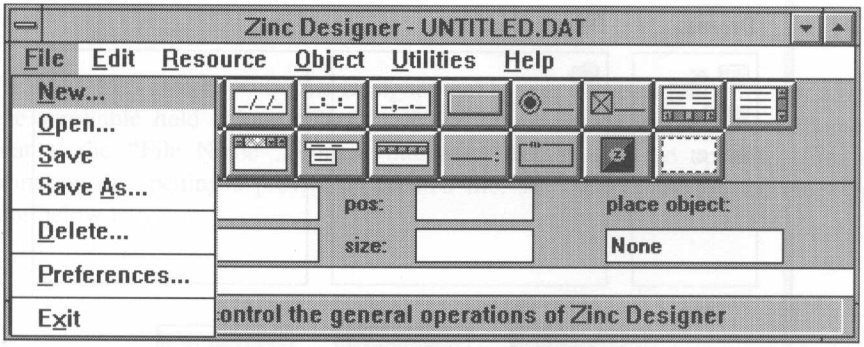
h) Call the editor for each object and select the help context to be associated with it from the "helpContext" combo box list. Select the editor's "OK" button.

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

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

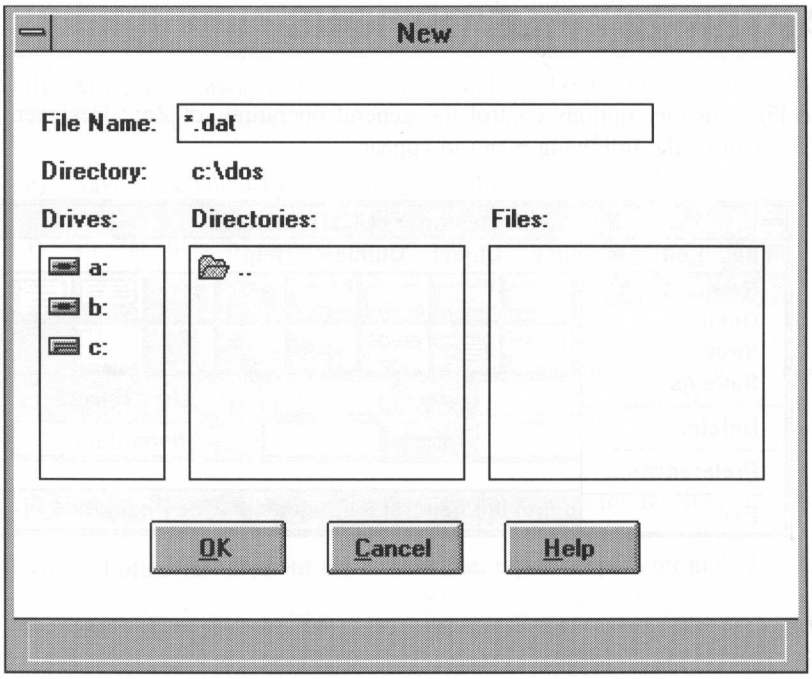
CHAPTER 22 - FILE OPTIONS

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



NEW

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



File name

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

Directory

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

Drives

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

Directories

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

Files

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

OK

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

Cancel

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

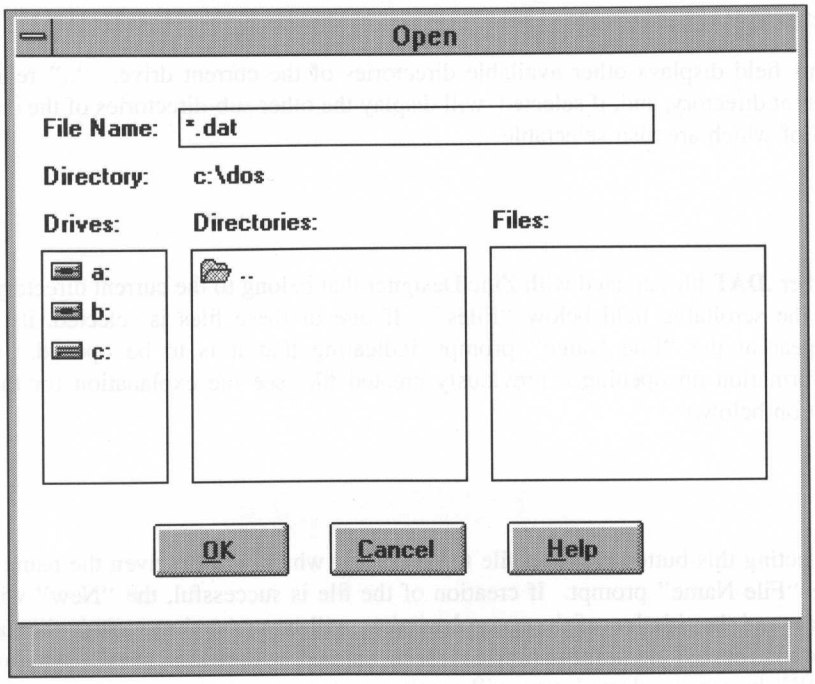
Help

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

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

OPEN

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



File name

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

Directory

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

Drives

This field displays the available drives on your system. If you want to make a different drive the current drive, select the desired drive from the drive list.

Directories

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

Files

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

OK

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

Cancel

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

Help

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

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

SAVE

Selecting the "Save" option causes the current file to be saved in its present condition. If the file has never been named, the "Save As" window will appear and allow you to name the file by entering a name at the "File Name" prompt. When you select the "OK" button, the "Save As" window will close and the file will be saved under that

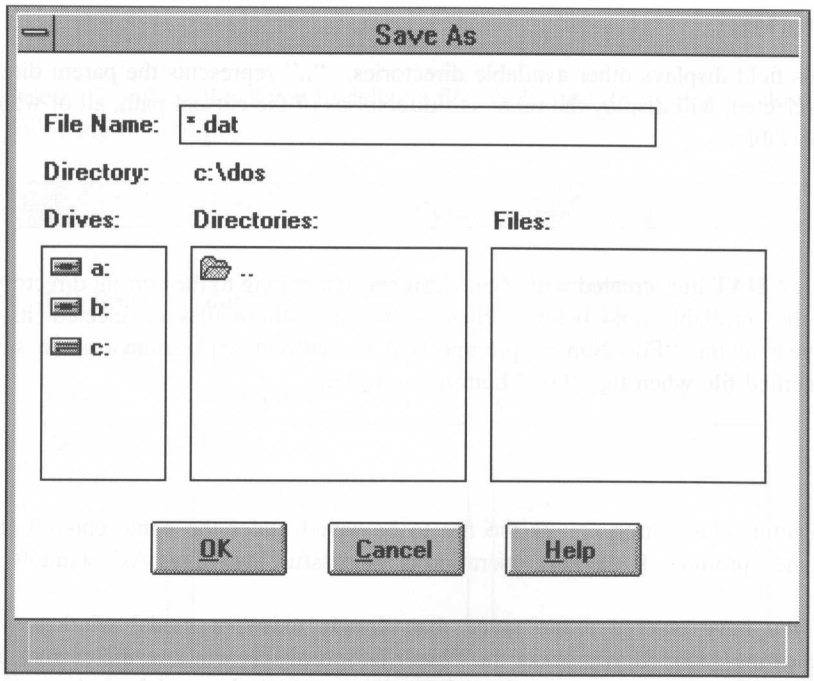
name. (See the Save As section for further details on how to save a file for the first time.)

Upon every save operation, Zinc Designer automatically creates the following four files:

- a "**.DAT**" file, which contains the binary information associated with the objects saved in the application
- a "**.CPP**" file, which contains the definition for *_objectTable*, an array that provides the function read access points for objects saved to disk, as well as the definition for *_userTable*, an array of function access points for user and compare functions.
- an "**.HPP**" file, which contains the numeric identifications (entered as stringID's) unique to each field
- one or more "**.BK#**" (i.e., backup) files, specified in File | Preferences. (**NOTE:** Only one backup file is created per Designer session and only if a previous **.DAT** file existed.)

SAVE AS

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



File name

Enter a name for the file at the "File Name" prompt, or select it from the "Files" field, and the name of the file will automatically appear at the prompt. If you do not include it yourself when entering the name at the prompt, a **.DAT** extension will be automatically attached when the file is actually created. A new file will be created under that name with the current modifications, if any.

Directory

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

Drives

This field displays the available drives on your system. If you want to make a different drive the current drive, select the desired drive from the drive list.

Directories

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

Files

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

OK

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

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

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

Cancel

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

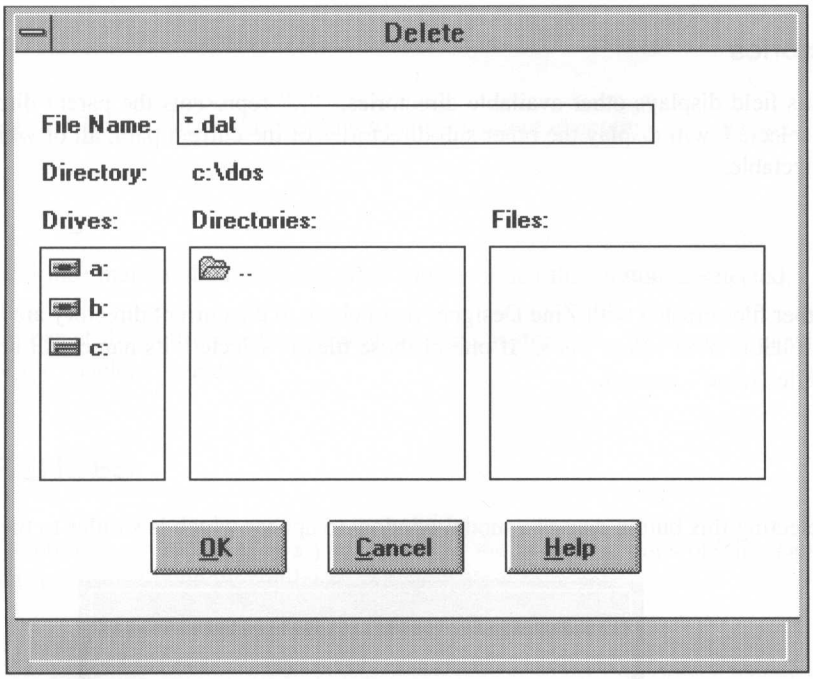
Help

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

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

DELETE

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



File name

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

Directory

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

Drives

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

Directories

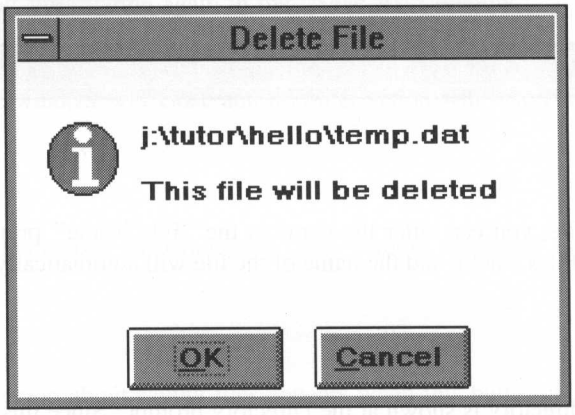
This field displays other available directories. ".." represents the parent directory, and, if selected, will display the other sub-directories of the current path, all of which are also selectable.

Files

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

OK

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



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

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

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

Cancel

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

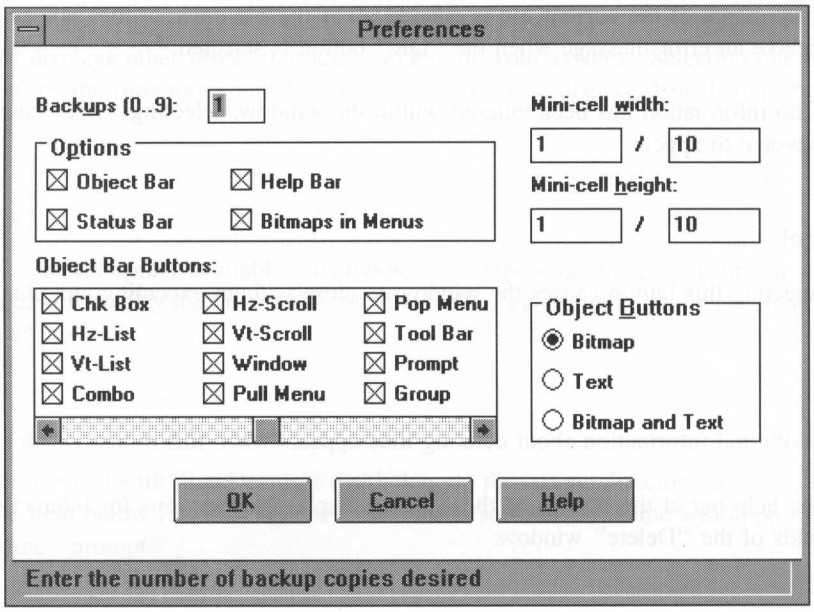
Help

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

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

PREFERENCES

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



Backups

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

Options

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

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

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

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

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

Mini-Cell

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

Object Bar Buttons

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

Object Buttons

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

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

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

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

OK

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

Cancel

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

Help

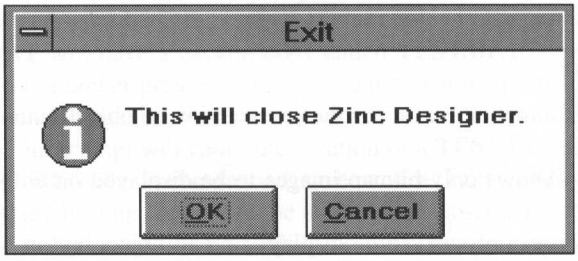
Additional information about default settings appears when this button is selected.

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

EXIT

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

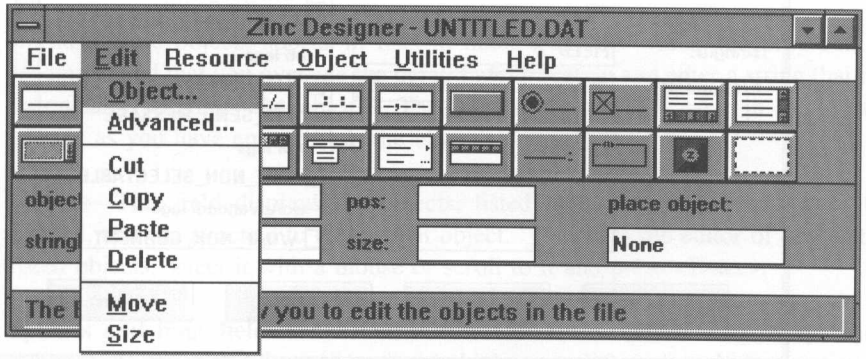
If you have not made any changes within Zinc Designer, selecting "Exit" causes a modal window to appear which is similar to the following:



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

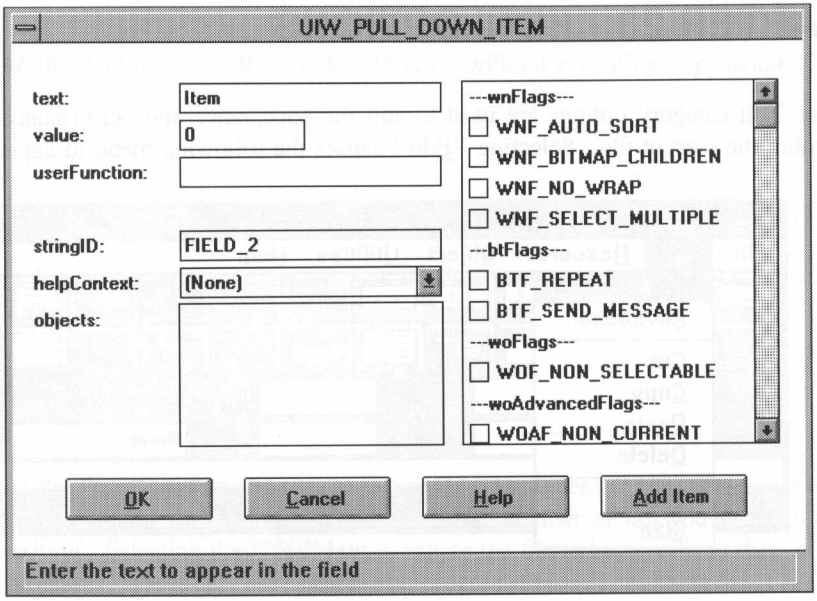
CHAPTER 23 - EDIT OPTIONS

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



OBJECT

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



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

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

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

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

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

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

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

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

(See Chapters 26 through 29 for more specific information regarding individual field objects.)

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

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

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

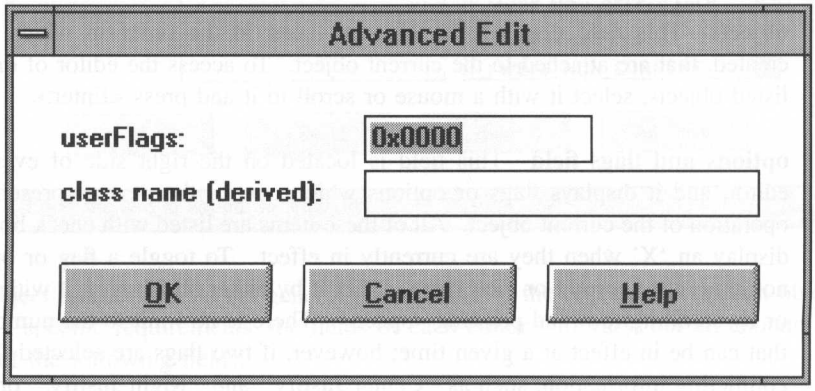
Help—Additional information about the current object appears when this button is selected.

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

ADVANCED

Selecting the "Advanced" option allows the advanced properties of an object to be edited. (**NOTE:** These properties should only be changed by experienced users!)

The Advanced Edit window is shown below:



The advanced properties that may be edited include:

userFlags—used to set the *userFlag* member variable associated with each object. Since Zinc Designer does not import the user-defined include files, the numeric representation of the user flags must be entered.

class name (derived)—is the name of the derived class to inherit the properties of the object being edited. If this field is left blank, no derived object will be created. If a name is entered into this field (e.g., EXAMPLE_CLASS), the object table in the **.CPP** file (generated by the designer) will contain an entry for the derived object's **New( )** function (e.g., EXAMPLE_CLASS::New( )). As part of the code for the derived class, the programmer must create a static **New( )** function that is able to call the constructor for the derived class. For an example of creating a **New( )** function, see "Chapter 14—Help Bar" of this manual. The **.HPP** file (generated by the designer) will contain a definition of the derived class' identification (i.e., an OBJECTID for the derived class).

CUT

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

COPY

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

PASTE

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

DELETE

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

MOVE

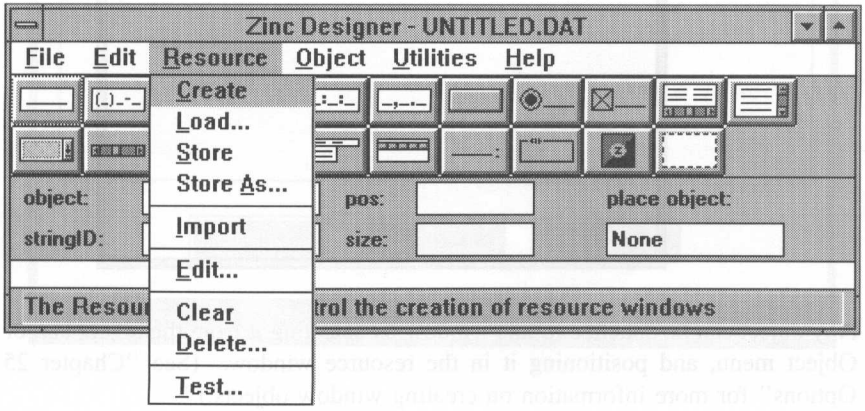
Selecting "Move" allows you to move the current object either by dragging the mouse or by using the arrow keys.

SIZE

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

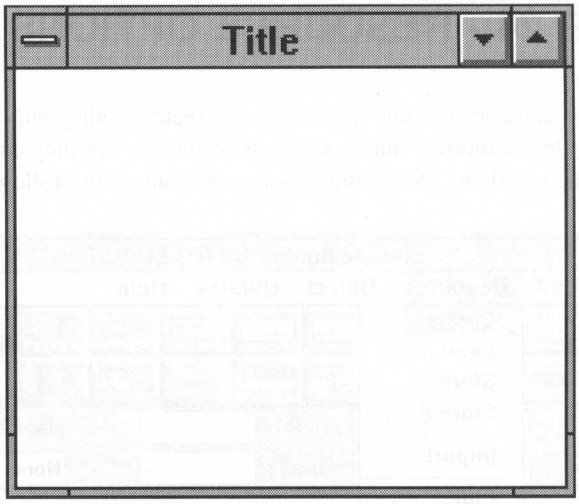
CHAPTER 24 - RESOURCE OPTIONS

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



CREATE

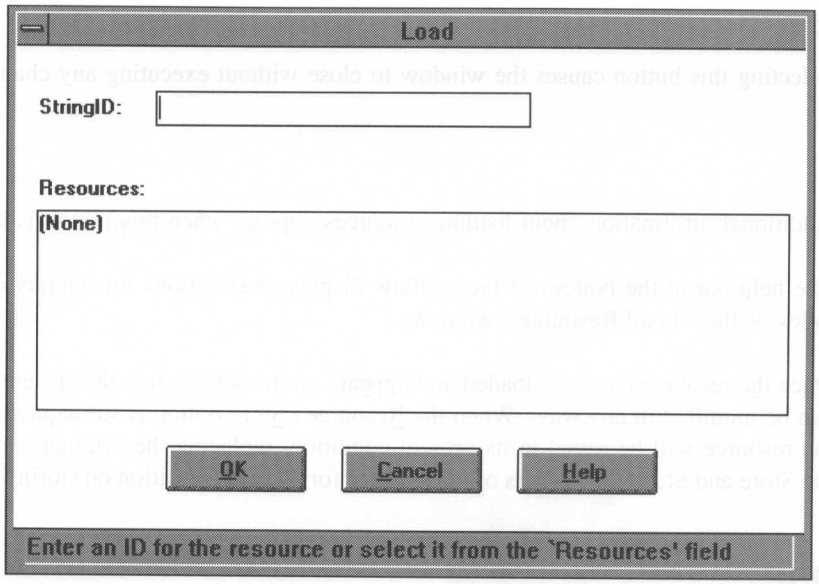
Selecting "Create" automatically places the following window on the screen, complete with a title bar, a system button, and minimize and maximize buttons.



Any object can be attached to this window by selecting it from the object bar, or from the Object menu, and positioning it in the resource window. (See "Chapter 25—Object Options" for more information on creating window objects.)

LOAD

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



stringID

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

Resources

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

OK

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

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

Cancel

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

Help

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

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

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

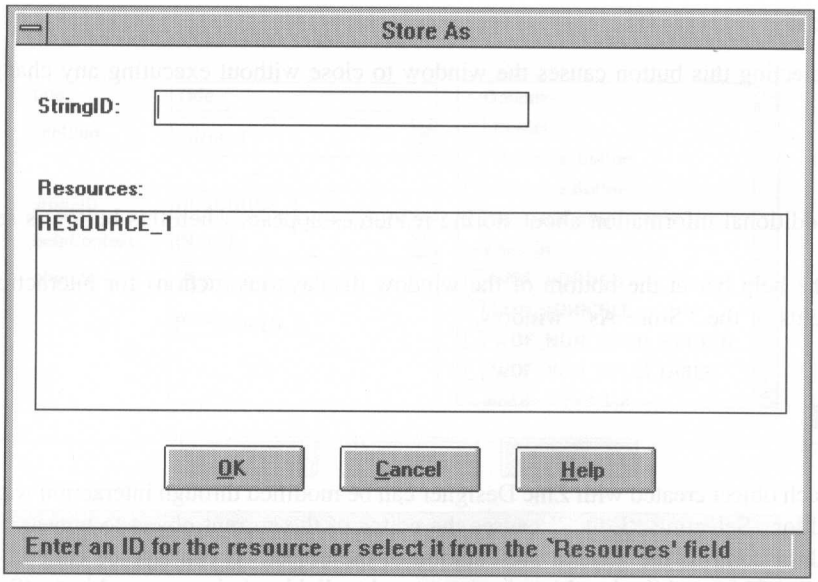
STORE

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

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

STORE AS

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



stringID

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

Resources

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

OK

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

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

Cancel

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

Help

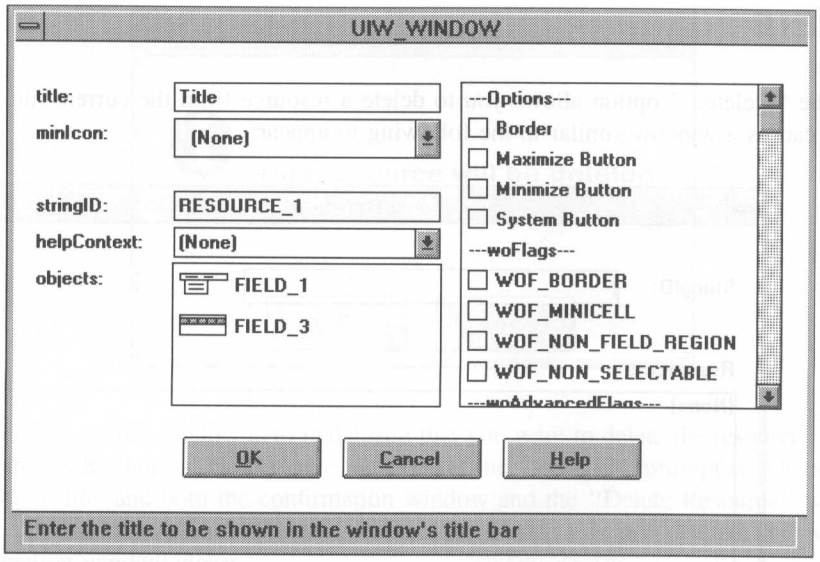
Additional information about storing resources appears when this button is selected.

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

EDIT

Each object created with Zinc Designer can be modified through interaction with its object editor. Selecting "Edit..." causes the editor of the current object to appear. The object editor controls the general presentation of the object. It can also be called by selecting Edit | Object while the object is current or by clicking twice on an object. (See Chapters 26 through 29 for further information on object editors.)

The resource editor (i.e., UIW_WINDOW editor) is invoked by selecting the Resource | Edit menu option (when the resource is current) or by double clicking the left mouse button on the body of the window. The UIW_WINDOW editor will appear similar to the following:



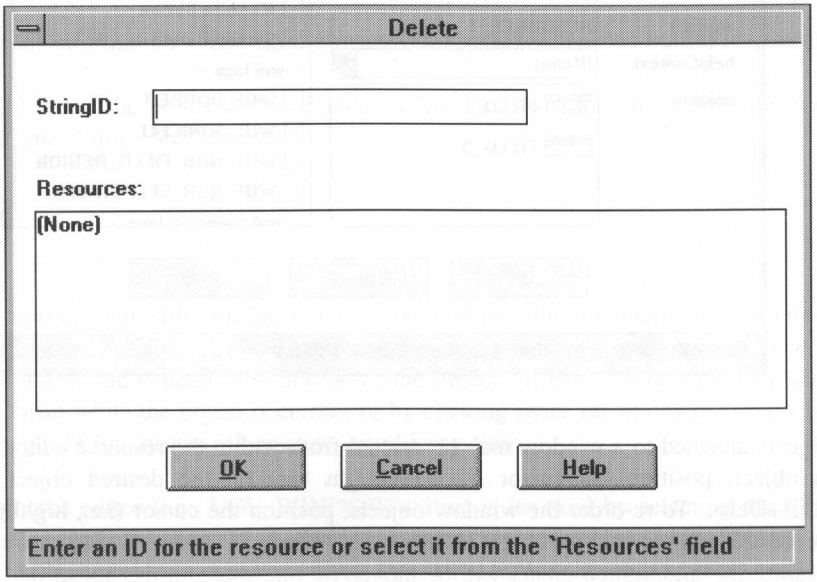
Objects attached to a window may be deleted from within the resource editor. To delete an object, position the cursor (i.e., highlight bar) on the desired object and press <Ctrl+Del>. To re-order the window objects, position the cursor (i.e., highlight bar) on the desired object and press the <Ctrl+↑> or <Ctrl+↓> keys. Each time <Ctrl+↑> is pressed, the highlighted object will be moved up one space in the object list. Similarly, each time <Ctrl+↓> is pressed, the highlighted object will be moved down one space in the object list. (**NOTE:** The tab sequence of the objects in the window is the order of the objects in the object list.)

CLEAR

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

DELETE

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



stringID

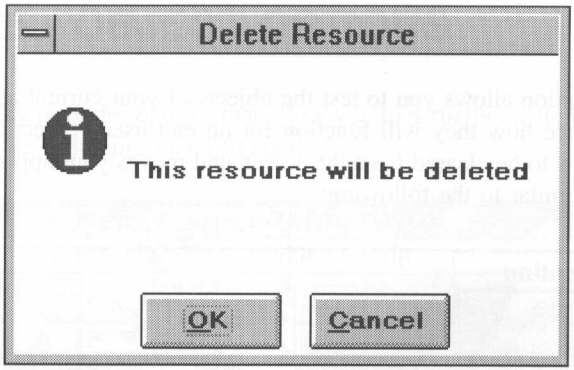
Enter the string identification of the resource to be deleted, or select the desired resource from the "Resources" field and the stringID will automatically be displayed at the prompt.

Resources

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

OK

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



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

If the resource entered does not exist, you will receive an error message when the "OK" button is selected.

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

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

Cancel

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

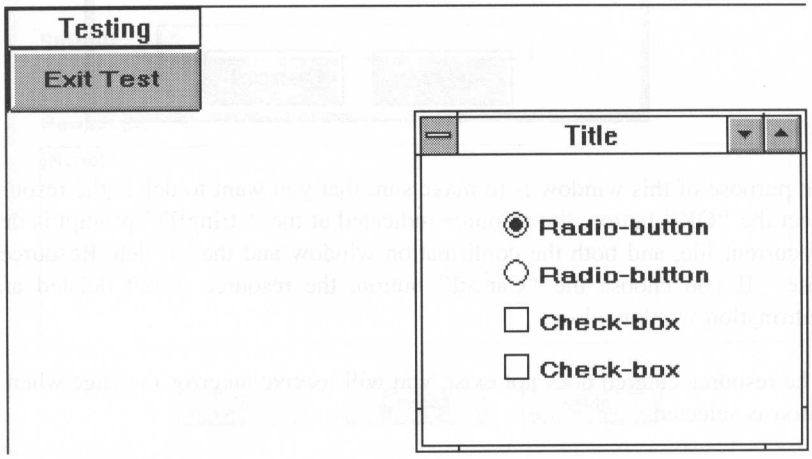
Help

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

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

TEST

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

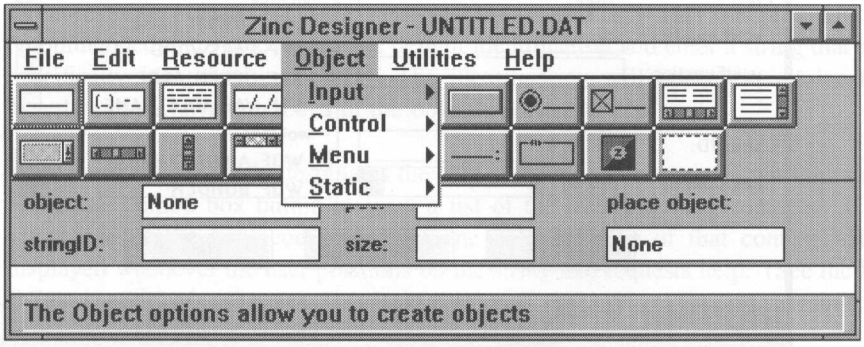


In test mode the objects of your application will look and act as they will for an end user. For example, check boxes and radio buttons will actually toggle and scroll bars will actually scroll information. No objects can be created or modified while in test mode.

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

CHAPTER 25 - OBJECT OPTIONS

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



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

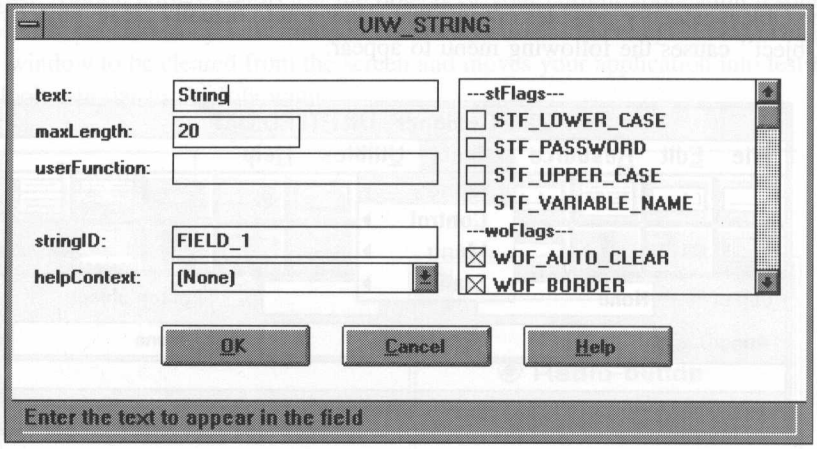
To create an object, select it from the associated menu. Position the mouse cursor where you want the object to appear on the resource window and either press the left mouse button or press <Enter>.

NOTE: All objects must be attached to a resource parent window; they cannot be attached directly to the screen. (See "Chapter 24—Resource Options" for more information on creating resource windows.)

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

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

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



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

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

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

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

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

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

helpContext—This field designates the help context to be associated with the string. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the string and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

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

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

Other fields that are more specific to individual objects are discussed in Chapters 26 through 29.

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

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

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

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

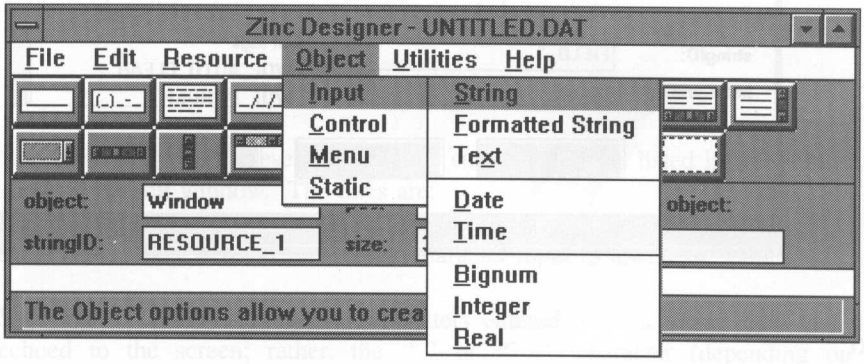
A help bar is also included in each object editor that displays help on how to interact with the edit window's fields.

To test how an object will actually appear and function for the end user, try it in test mode, which is accessed by selecting Resource | Test while the parent resource is active. (See the Test section of "Chapter 24—Resource Options" for more information on testing objects.)

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

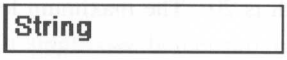
CHAPTER 26 - INPUT OBJECTS

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

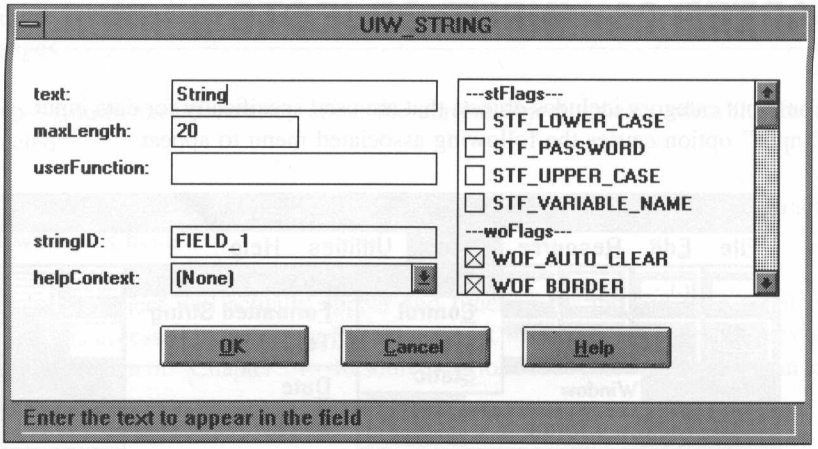


STRING

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



The string object may be placed on a window by clicking (i.e., on the window) with the left mouse button. To modify the string object, call its editor by double clicking the mouse on the object. The following window will appear:



text

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

maxLength

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

userFunction

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

stringID

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

helpContext

This field designates the help context to be associated with the string. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the string and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

flags

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

STF_LOWER_CASE—Converts all character input to lowercase values.

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

STF_UPPER_CASE—Converts all character input to uppercase values.

STF_VARIABLE_NAME—Converts the space character to an underscore value.

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

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

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

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

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

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

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

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

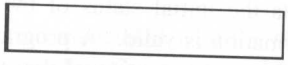
WOF_UNANSWERED—Sets the initial status of the string field to be "unanswered." An unanswered string field is displayed as blank space on the screen.

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

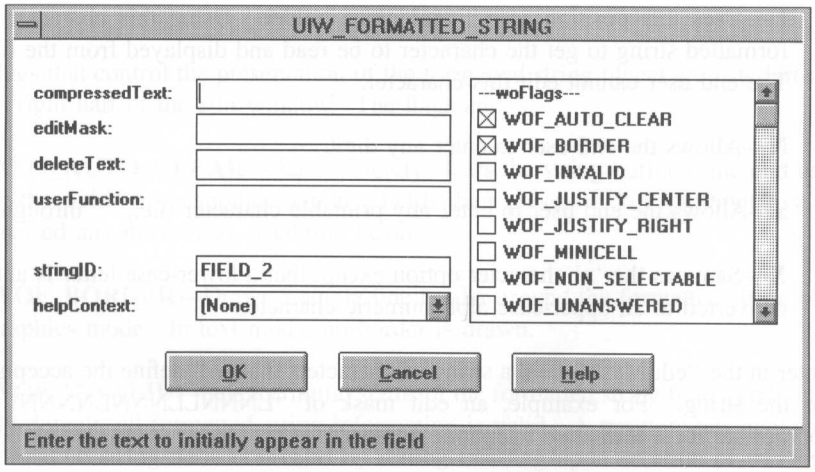
WOAF_NON_CURRENT—The string cannot be made current. If this flag is set, users will not be able to select the string from the keyboard nor with the mouse.

FORMATTED STRING

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



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



compressedText

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

editMask

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

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

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

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

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

L—Uses this position as a literal place holder. Using this character causes the formatted string to get the character to be read and displayed from the literal mask. The end user cannot edit this character.

N—Allows the end user to enter any digit.

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

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

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

deleteText

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

userFunction

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

stringID

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

helpContext

This field designates the help context to be associated with the formatted string. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the formatted string and requests help. (See the Help Editor section of "Chapter 30—Utilities Options" for information on creating help contexts.)

flags

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

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

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

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

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

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

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

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

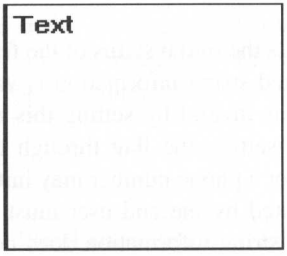
WOF_NON_SELECTABLE—Prevents the formatted string object from being selected. If this flag is set, the end user will not be able to edit or position on the formatted string information.

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

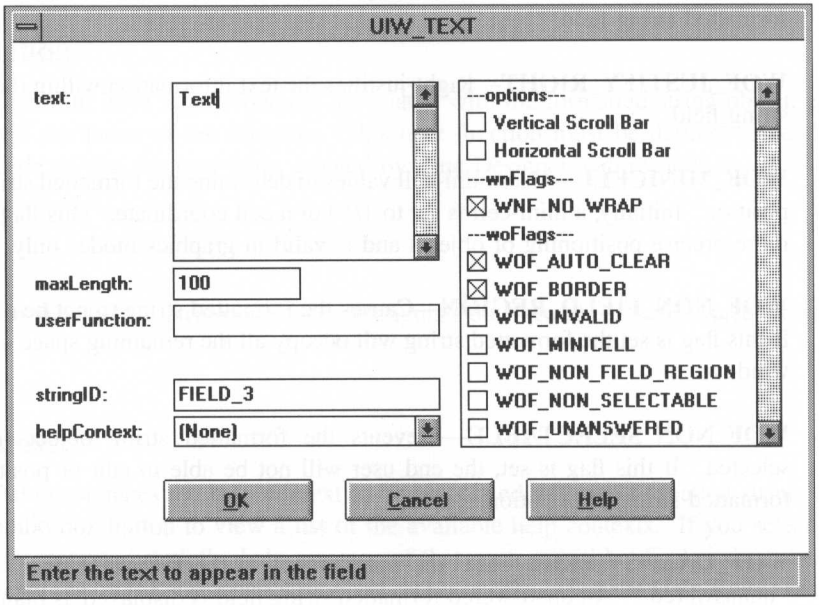
WOAF_NON_CURRENT—The formatted string object cannot be made current. If this flag is set, users will not be able to select the formatted string from the keyboard or with the mouse.

TEXT

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



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



text

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

maxLength

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

userFunction

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

stringID

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

helpContext

This field designates the help context to be associated with the text field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the text field and requests help. (See the Help Editor section of "Chapter 30—Utilities Options" for information on creating help contexts.)

options and flags

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

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

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

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

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

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

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

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

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

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

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

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

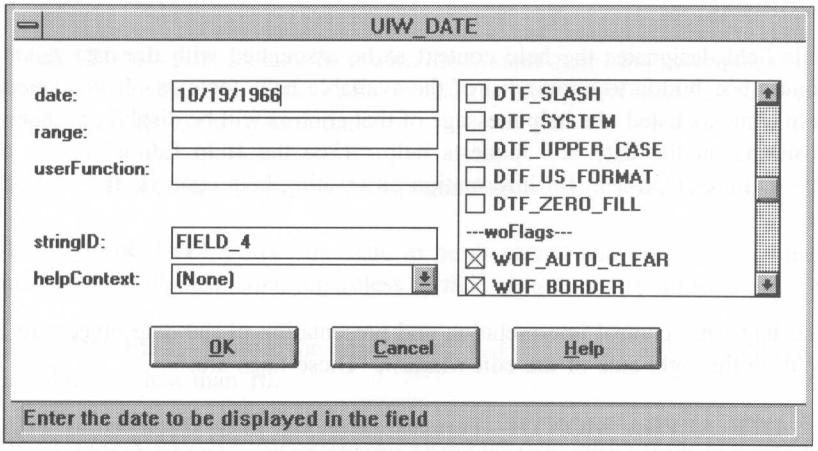
WOF_VIEW_ONLY—Prevents the text object from being edited. If this flag is set, the end user will not be able to edit the text information but will be able to browse through the text field.

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

DATE

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

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



date

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

range

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

userFunction

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

stringID

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

helpContext

This field designates the help context to be associated with the date field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the date and requests help. (See the Help Editor section of "Chapter 30—Utilities Options" for information on creating help contexts.)

flags

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

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

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

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

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

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

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

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

DTF_SHORT_MONTH—Uses a shortened alphanumeric month in the date.

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

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

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

DTF_UPPER_CASE—Converts the alphanumeric date to upper case.

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

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

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

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

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

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

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

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

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

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

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

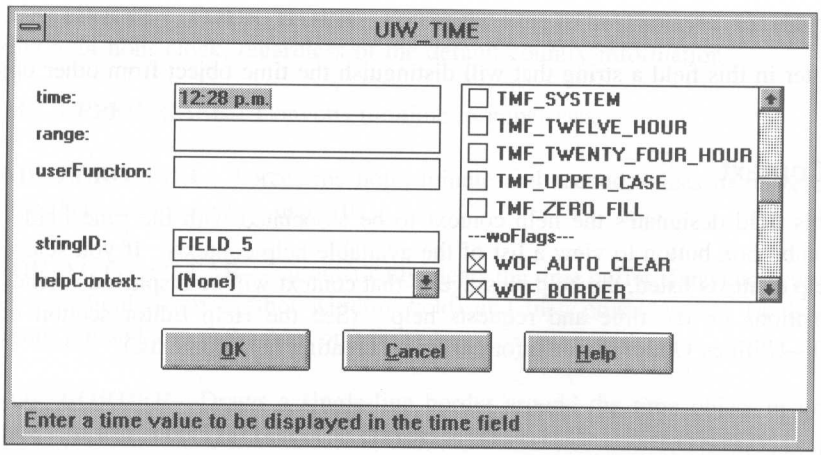
WOF_VIEW_ONLY—Prevents the date object from being edited. If this flag is set, the end user will not be able to edit a date object's information but will be able to browse through the information.

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

TIME

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

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



time

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

range

If you want to specify a certain range of acceptable time values, enter in this field the valid time ranges. For example, if you want to accept only those times whose values fall in post-meridian time, enter the range of "12:00pm..11:59:59pm." If no range is entered, any time value will be accepted.

userFunction

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

stringID

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

helpContext

This field designates the help context to be associated with the time field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the time and requests help. (See the Help Editor section of "Chapter 30—Utilities Options" for information on creating help contexts.)

flags

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

TMF_COLON_SEPARATOR—Separates each time value with a colon.

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

TMF_LOWER_CASE—Converts the time to lower-case.

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

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

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

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

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

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

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

TMF_UPPER_CASE—Converts the time to upper case.

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

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

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

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

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

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

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

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

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

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

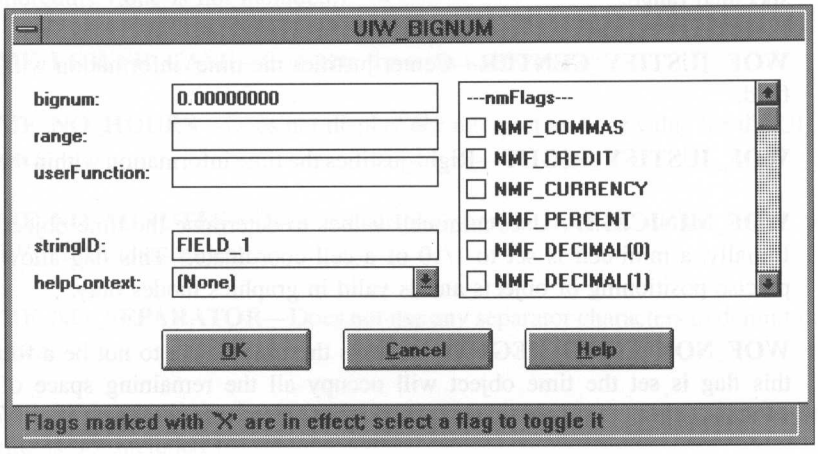
WOF_VIEW_ONLY—Prevents the time field from being modified. This flag will still allow the time field to become current.

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

BIGNUM

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

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



bignum

Enter in this field the number that you want to appear in the bignum field. The number will be displayed with the number of decimal places designated by the flags you have set.

A bignum object can have up to thirty digits to the left of the decimal place and up to eight digits to the right of the decimal place.

range

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

userFunction

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

stringID

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

helpContext

This field designates the help context to be associated with the bignum field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the bignum field and requests help. (See the Help Editor section of "Chapter 30—Utilities Options" for information on creating help contexts.)

flags

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

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

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

NMF_COMMAS—Displays the number with commas.

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

NMF_PERCENT—Displays the number with a percentage symbol.

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

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

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

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

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

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

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

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

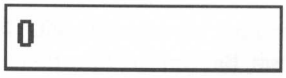
WOF_UNANSWERED—Sets the initial status of the bignum field to be "unanswered." An unanswered bignum field is displayed as blank space on the screen.

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

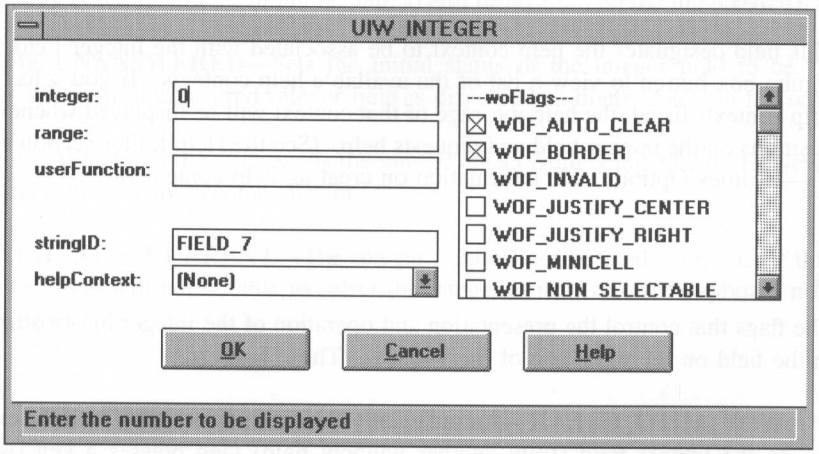
WOAF_NON_CURRENT—The bignum object cannot be made current. If this flag is set, the bignum object cannot be selected from the keyboard nor with the mouse.

INTEGER

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



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



integer

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

range

If you want to specify a certain range of acceptable integer values, enter in this field the valid integer ranges. For example, if you want to accept only numbers between 100 and 10,000, enter the range of "100..10000." If no range is entered, any integer value will be accepted.

userFunction

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

stringID

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

helpContext

This field designates the help context to be associated with the integer field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the integer field and requests help. (See the Help Editor section of "Chapter 30—Utilities Options" for information on creating help contexts.)

flags

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

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

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

WOF_INVALID—Sets the initial status of the integer field to be "invalid." Invalid numbers fit in the absolute range determined by the object type but do not fulfill all the requirements specified by the program. For example, an integer may initially be

set to **200**, but the final number, edited by the end user, must be in the range "10..100." The initial number in this example fits the absolute requirements of an integer class object but does not fit into the specified range.

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

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

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

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

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

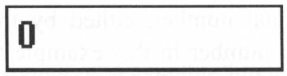
WOF_UNANSWERED—Sets the initial status of the integer field to be "unanswered." An unanswered integer field is displayed as blank space on the screen.

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

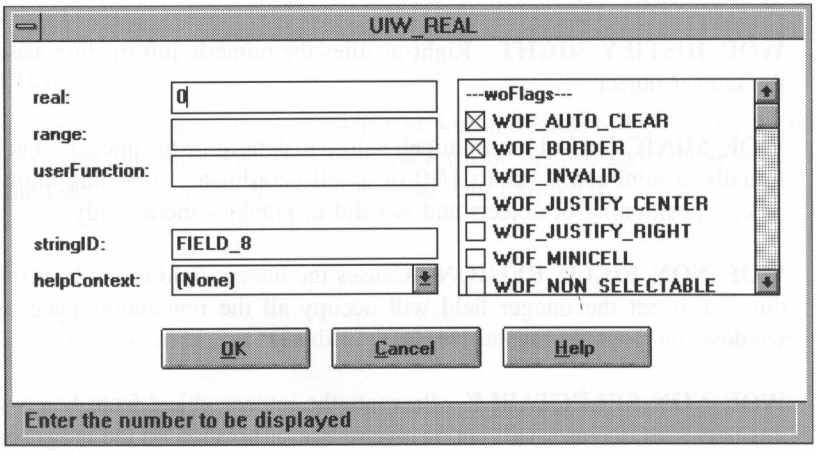
WOAF_NON_CURRENT—The integer field cannot be made current. If this flag is set, users will not be able to select the integer object from the keyboard nor with the mouse.

REAL

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



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



real

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

range

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

userFunction

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

stringID

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

helpContext

This field designates the help context to be associated with the real number field. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the real number field and requests help. (See the Help Editor section of "Chapter 30—Utilities Options" for information on creating help contexts.)

flags

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

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

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

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

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

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

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

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

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

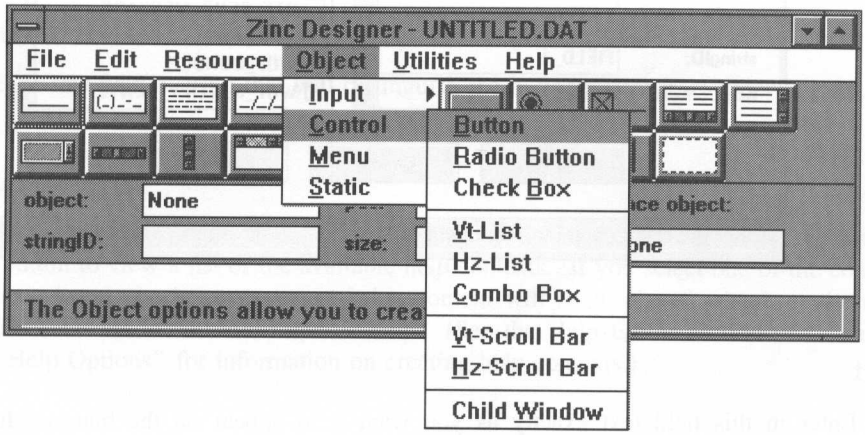
WOF_UNANSWERED—Sets the initial status of the real number field to be "unanswered." An unanswered real number field is displayed as blank space on the screen.

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

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

CHAPTER 27 - CONTROL OBJECTS

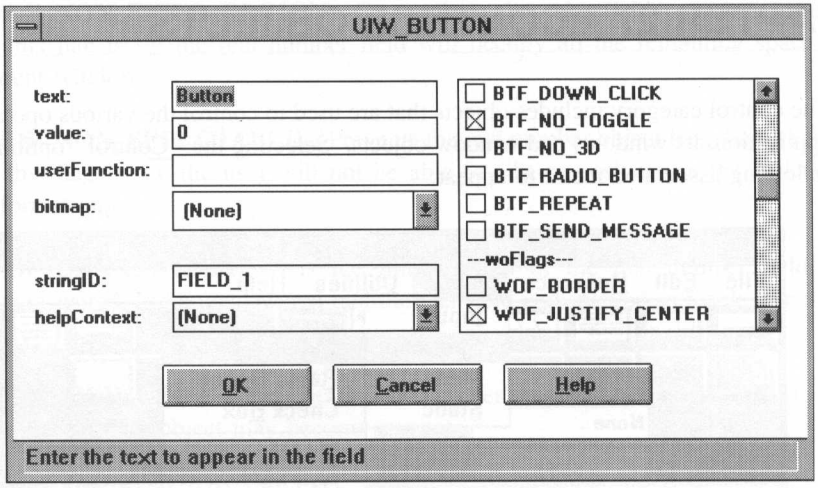
The control category includes objects that are used to control the various operations of an application, its windows and window objects. Selecting the "Control" option causes the following associated menu to appear:



BUTTON

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

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



text

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

value

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

userFunction

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

bitmap

This field designates the bitmap image to be associated with the button. Select the combo box button to view a list of the available bitmaps. If you select one of the bitmaps listed, it will be displayed on the button. (See the Image Editor section of "Chapter 30—Utilities Options" for information on creating bitmap images.)

stringID

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

helpContext

This field designates the help context to be associated with the button. Select the combo box button to view a list of the available help contexts. If you select one of the context contexts listed, the help message of that context will be displayed whenever the user positions on the button and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

flags

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

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

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

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

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

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

BTF_NO_3D—Causes the button to be displayed without a 3D appearance.

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

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

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

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

WOF_JUSTIFY_CENTER—Center-justifies the text within the button.

WOF_JUSTIFY_RIGHT—Right-justifies the text within the button.

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

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

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

WOAF_NON_CURRENT—Prevents the button object from being made current. However, clicking the button with the mouse or pressing the button's hotkey will call the object's user function.

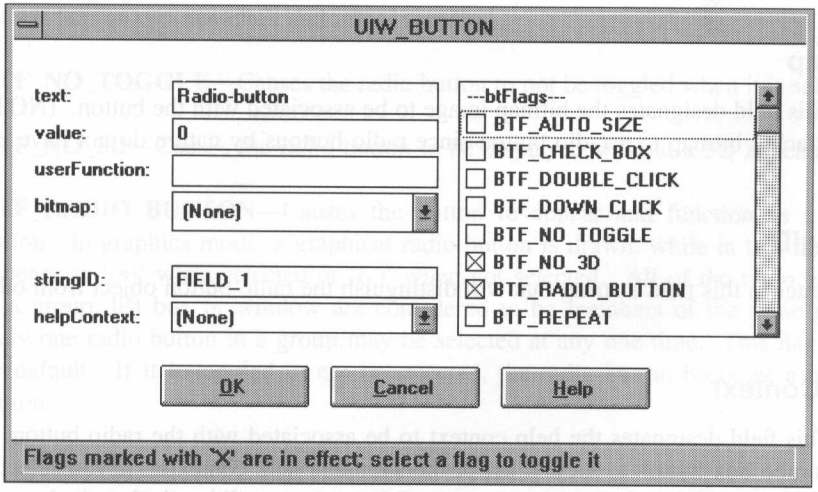
RADIO BUTTON

A radio button is a type of button that displays not only text, but also an indicator that toggles. All of the radio buttons in a group, list box or window are considered to be members of the same group. Only one radio button in a group may be selected at any one time. (**NOTE:** The radio button's parent must not have the WNF_SELECT_-MULTIPLE flag set or multiple radio buttons would be able to be displayed in the 'on' state.) Selecting "Radio Button" causes the following object to appear (in graphics mode):

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

NOTE: To have multiple radio button groups on the same window use the group object. (See the Group section of "Chapter 29—Static Options" for information on creating groups.)

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



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

text

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

value

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

userFunction

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

bitmap

This field designates the bitmap image to be associated with the button. (**NOTE:** Do not attach a bitmap to a radio button since radio buttons by nature do not have bitmaps.)

stringID

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

helpContext

This field designates the help context to be associated with the radio button. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user

positions on the radio button and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

flags

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

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

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

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

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

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

BTF_NO_3D—Causes the radio button to be displayed without a 3D appearance.

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

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

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

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

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

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

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

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

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

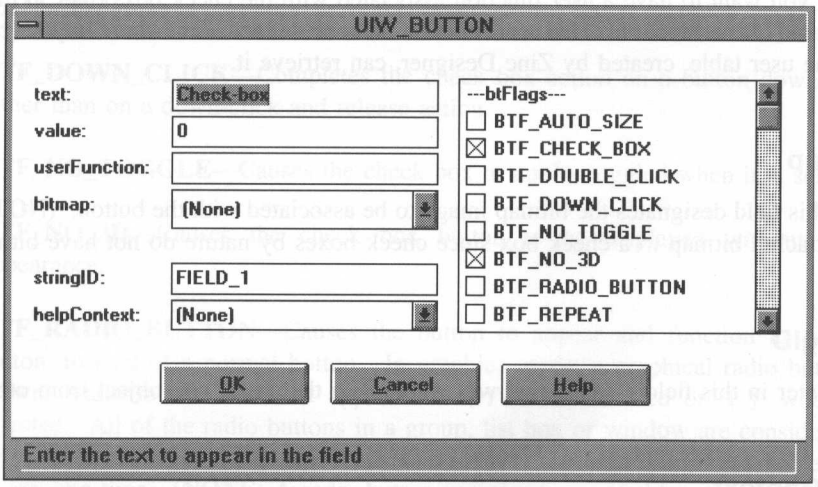
WOAF_NON_CURRENT—Prevents the radio button object from being made current. However, clicking the radio button with the mouse or pressing the radio button's hotkey will call the object's user function.

CHECK BOX

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

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

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



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

text

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

value

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

userFunction

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

bitmap

This field designates the bitmap image to be associated with the button. (**NOTE:** Do not attach a bitmap to a check box since check boxes by nature do not have bitmaps.)

stringID

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

helpContext

This field designates the help context to be associated with the check box button. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the check box and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

flags

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

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

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

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

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

BTF_NO_TOGGLE—Causes the check box to not be toggled when it is selected.

BTF_NO_3D—Causes the check box button to be displayed without a 3D appearance.

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

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

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

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

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

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

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

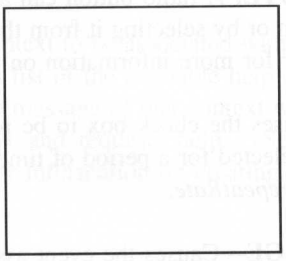
WOF_NON_FIELD_REGION—Causes the check box object to not be a form field. If this flag is set the check box object will occupy all of the remaining space of its parent window.

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

WOAF_NON_CURRENT—Prevents the check box object from being made current. However, clicking the check box with the mouse or pressing the check box's hotkey will call the object's user function.

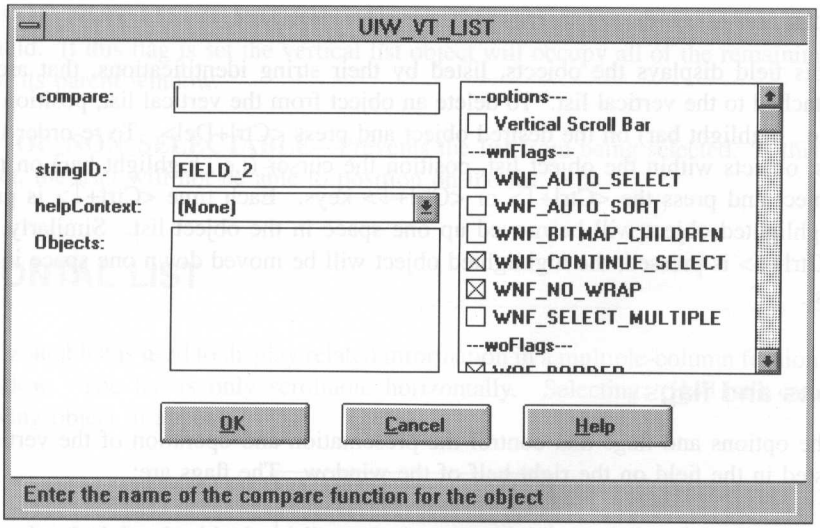
VERTICAL LIST

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



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

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



compare

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

stringID

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

helpContext

This field designates the help context to be associated with the vertical list. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the vertical list and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

objects

This field displays the objects, listed by their string identifications, that are currently attached to the vertical list. To delete an object from the vertical list, position the cursor (i.e., highlight bar) on the desired object and press <Ctrl+Del>. To re-order the vertical list objects within the object list, position the cursor (i.e., highlight bar) on the desired object and press the <Ctrl+↑> or <Ctrl+↓> keys. Each time <Ctrl+↑> is pressed, the highlighted object will be moved up one space in the object list. Similarly, each time <Ctrl+↓> is pressed, the highlighted object will be moved down one space in the object list.

options and flags

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

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

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

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

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

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

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

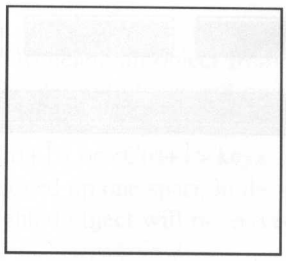
WOF_MINICELL—Uses mini-cell values to determine the object's position. Initially, a mini-cell is set to 1/10 of a cell coordinate. This flag allows for more precise positioning of objects and is valid in graphics modes only.

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

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

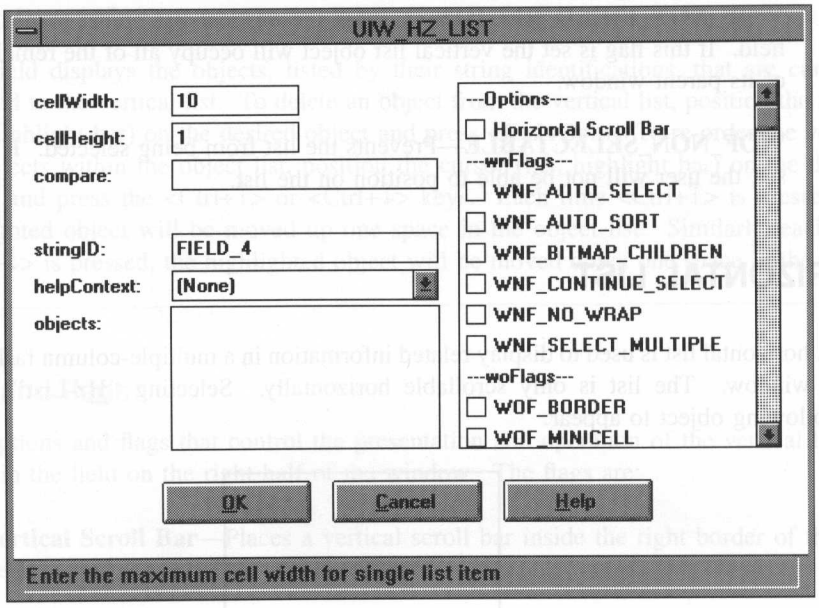
HORIZONTAL LIST

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



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

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



cellWidth

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

cellHeight

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

compare

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

stringID

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

helpContext

This field designates the help context to be associated with the horizontal list. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the horizontal list and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

objects

This field displays the objects, listed by their string identifications, that are currently attached to the horizontal list. To delete an object from the horizontal list, position the cursor (i.e., highlight bar) on the desired object and press <Ctrl+Del>. To re-order the horizontal list objects within the object list, position the cursor (i.e., highlight bar) on the desired object and press the <Ctrl+↑> or <Ctrl+↓> keys. Each time <Ctrl+↑> is pressed, the highlighted object will be moved up one space in the object list. Similarly, each time <Ctrl+↓> is pressed, the highlighted object will be moved down one space in the object list.

options and flags

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

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

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

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

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

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

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

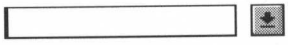
WOF_MINICELL—Uses mini-cell values to determine the object's position. Initially, a mini-cell is set to 1/10 of a cell coordinate. This flag allows for more precise positioning of objects and is valid in graphics modes only.

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

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

COMBO BOX

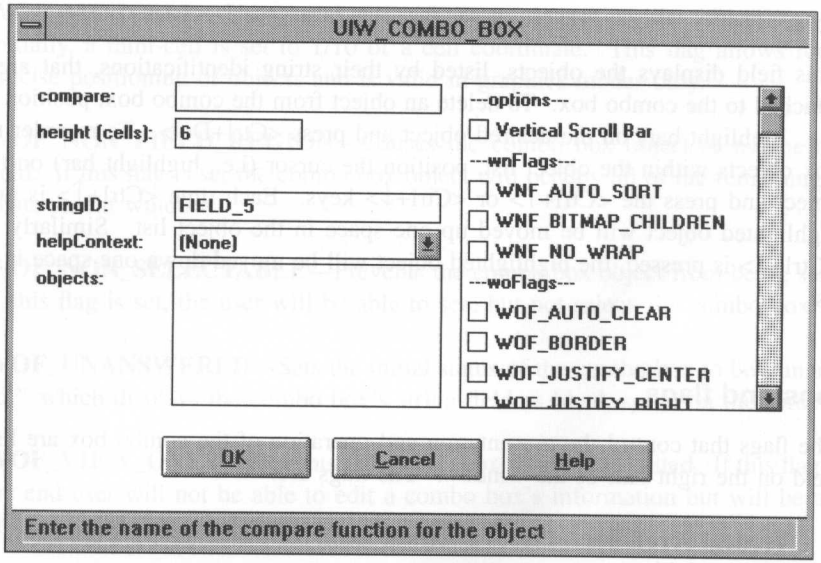
A combo box is a combination of a string field and a scrollable list box. It is used to display a list of selectable items. When one of the items is selected, it appears in the string field. Selecting "Combo Box" causes the following object to appear:



The scrollable list is displayed when the button to the right of the string field is selected. When one of the items of that list is selected, it is copied into the string field and the list box disappears. If the string field is editable, the user can enter text and the item in the list that most closely matches the characters typed will be highlighted. The user can then select the item to copy it back into the string field.

Objects are added to the combo box's list by selecting them from the menu or object bar and placing them on the combo box object. By default, they will be automatically aligned in a single column in the order in which they were created.

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



compare

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

stringID

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

helpContext

This field designates the help context to be associated with the combo box. Select the help context field's combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the combo box and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

objects

This field displays the objects, listed by their string identifications, that are currently attached to the combo box. To delete an object from the combo box, position the cursor (i.e., highlight bar) on the desired object and press <Ctrl+Del>. To re-order the combo box objects within the object list, position the cursor (i.e., highlight bar) on the desired object and press the <Ctrl+↑> or <Ctrl+↓> keys. Each time <Ctrl+↑> is pressed, the highlighted object will be moved up one space in the object list. Similarly, each time <Ctrl+↓> is pressed, the highlighted object will be moved down one space in the object list.

options and flags

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

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

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

WNF_BITMAP_CHILDREN—Should be set when items other than strings are added to the combo box.

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

WOF_AUTO_CLEAR—Automatically clears the edit buffer if the end-user tabs to the combo box (from another window field) and presses a non-movement key.

WOF_BORDER—Draws a border around the combo box object. In graphics mode, setting this option draws a single-pixel border around the object. In text mode, no border is drawn.

WOF_JUSTIFY_CENTER—Center-justifies the string associated with the combo box's string field.

WOF_JUSTIFY_RIGHT—Right-justifies the string associated with the combo box's string field.

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

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

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

WOF_UNANSWERED—Sets the initial status of the combo box to be "unanswered," which displays the combo box's string field as blank space on the screen.

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

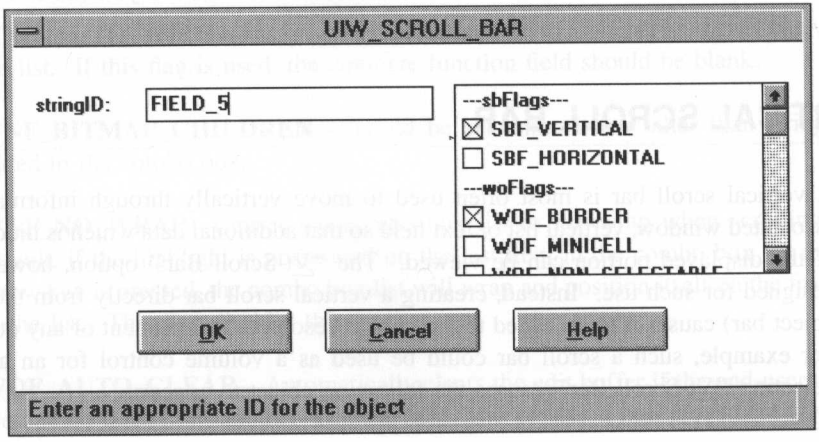
WOAF_NON_CURRENT—Prevents the combo box object from being made current.

VERTICAL SCROLL BAR

A vertical scroll bar is most often used to move vertically through information in an associated window, vertical list or text field so that additional data which is hidden outside of the displayed portion can be viewed. The "Vt-Scroll Bar" option, however, is not designed for such use. Instead, creating a vertical scroll bar directly from the menu (or object bar) causes it to be added to the current resource, independent of any other object. For example, such a scroll bar could be used as a volume control for an application. Selecting "Vt-Scroll Bar" causes the following object to appear:

NOTE: To associate a vertical scroll bar with a window, a text field or a vertical list, simply select the vertical scroll bar option or flag available in the editor for each of these objects.

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



NOTE: The editor for the vertical scroll bar is actually an editor for a generic scroll bar object but with the SBF_VERTICAL flag set. If this flag is toggled off, or if another SBF flag is set, the scroll bar will no longer be displayed as a vertical one.

flags

The flags that control the presentation and operation of the vertical scroll bar are listed in the right field of the window. The flags are:

SBF_HORIZONTAL—Defines the scroll bar object to be a horizontal scroll bar.

SBF_VERTICAL—Defines the scroll bar object to be a vertical scroll bar.

WOF_BORDER—Draws a single line border around the scroll bar object.

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

WOF_NON_FIELD_REGION—Causes the vertical scroll bar object to not be a form field. If this flag is set the vertical scroll bar object will occupy all of the right-most space of its parent window.

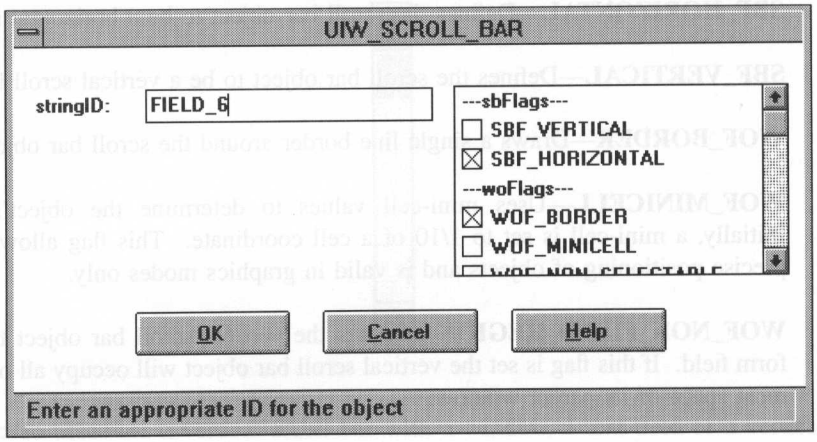
WOF_NON_SELECTABLE—Prevents the vertical scroll bar from being selected. If this flag is set, the user will not be able to position on the scroll bar.

HORIZONTAL SCROLL BAR

A horizontal scroll bar is most often used to move horizontally through information in an associated window, horizontal list or text field so that additional data which is hidden outside of the displayed portion can be viewed. The "Hz-Scroll Bar" option, however, is not designed for such use. Instead, creating a horizontal scroll bar directly from the menu (or object bar) causes it to be added to the current resource, independent of any other object. For example, such a scroll bar could be used as a volume control for an application. Selecting "Hz-Scroll Bar" causes the following object to appear:

NOTE: To associate a horizontal scroll bar with a window, a text field or a horizontal list, simply select the horizontal scroll bar option or flag available in the editor for each of these objects.

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



NOTE: The editor for the horizontal scroll bar is actually an editor for a generic scroll bar object but with the SBF_HORIZONTAL flag set. If this flag is toggled off, or if another SBF flag is set, the scroll bar will no longer be displayed as a horizontal one.

flags

The flags that control the presentation and operation of the horizontal scroll bar are listed in the right field of the window. The flags are:

SBF_HORIZONTAL—Defines the scroll bar object to be a horizontal scroll bar.

SBF_VERTICAL—Defines the scroll bar object to be a vertical scroll bar.

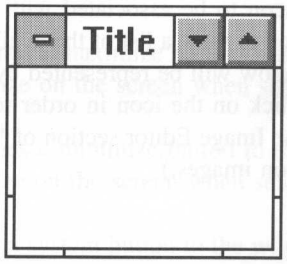
WOF_BORDER—Draws a single line border around the scroll bar object.

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

WOF_NON_FIELD_REGION—Causes the horizontal scroll bar object to not be a form field. If this flag is set the horizontal scroll bar object will occupy all of the bottom-most space of its parent window.

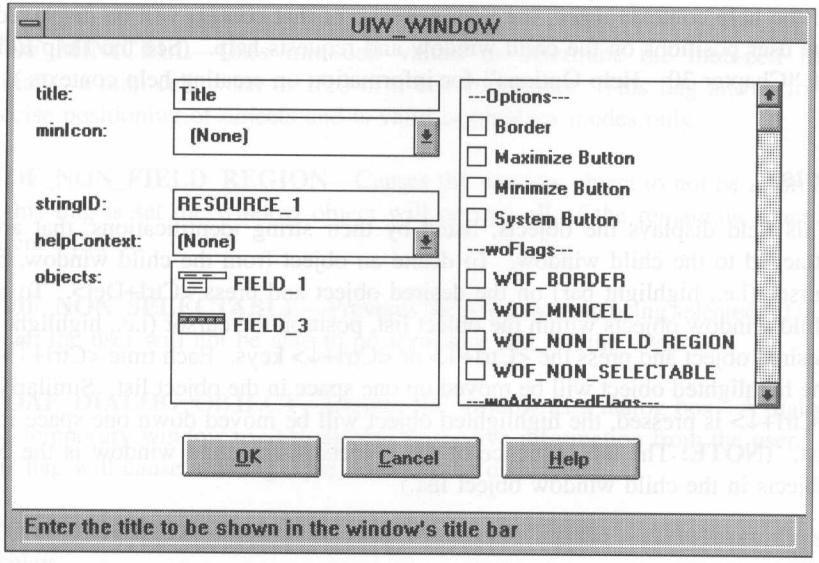
CHILD WINDOW

A window is used as a controlling structure for displaying and interacting with other objects. This object is known as a child window in order to distinguish it from the main resource window to which it must be attached. Selecting "Child Window" causes the following object to appear:



By default, the window is created with a title, a system button, a maximize button and a minimize button. Other objects can be added by simply selecting them from the menu or object bar and placing them on the window.

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



title

Enter in this field the text exactly as you want it to appear in the window's title. If you do not want a title for the window, delete the default string "Title."

minIcon

This field designates the icon to be associated with the window when it is minimized. Select the combo box button to view a list of the available icon images. If you select one of the icons listed, the window will be represented by it when in a minimized state. The end user will be able to click on the icon in order to restore the window to its original size on the screen. (See the Image Editor section of "Chapter 30—Utilities Options" for information on creating icon images.)

stringID

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

helpContext

This field designates the help context to be associated with the child window. Select the field's combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the child window and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

objects

This field displays the objects, listed by their string identifications, that are currently attached to the child window. To delete an object from the child window, position the cursor (i.e., highlight bar) on the desired object and press <Ctrl+Del>. To re-order the child window objects within the object list, position the cursor (i.e., highlight bar) on the desired object and press the <Ctrl+↑> or <Ctrl+↓> keys. Each time <Ctrl+↑> is pressed, the highlighted object will be moved up one space in the object list. Similarly, each time <Ctrl+↓> is pressed, the highlighted object will be moved down one space in the object list. (**NOTE:** The tab sequence of the objects in the child window is the order of the objects in the child window object list.)

options and flags

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

Border—Draws a three-dimensional border around the outer perimeter of the window. (**NOTE:** This is an actual UIW_BORDER object, unlike the WOF_-BORDER flag, which basically just outlines the window field.)

Maximize Button—Attaches a maximize button to the window that will enlarge the window to its maximum size on the screen when selected.

Minimize Button—Attaches a minimize button to the window that will reduce the window to its minimum size on the screen when selected.

System Button—Attaches a system button to the window. When selected, a system button displays the following selectable options: Restore, Move, Size, Minimize, Maximize and Close.

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

WOF_BORDER—Draws a single line border around the window. If the application program is running in text mode, no border is drawn.

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

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

WOF_NON_SELECTABLE—Prevents the window from being selected. If this flag is set, the user will not be able to position, nor edit, on the window.

WOAF_DIALOG_OBJECT—Creates the window as a dialog box. A dialog box is a temporary window used to display or receive information from the user. Using this flag will cause a dialog style border to be displayed.

WOAF_LOCKED—Prevents the user from removing the window from the screen display.

WOAF_MDI_OBJECT—Creates the window as an MDI window. If the MDI window is added to the Window Manager, it becomes an MDI parent (i.e., it can contain MDI child objects.) An MDI parent must have a pull-down menu. In general, other than the standard support objects (i.e., system button, border, title, etc.) and the pulldown menu, MDI parent windows should only contain MDI children.

If the MDI window is added directly to another MDI window, it will become an MDI child object. MDI child windows can be moved or sized but will remain entirely within the MDI parent window.

WOAF_MODAL—Prevents any other window from receiving event information from the Window Manager. A modal window receives all event information until it is removed from the screen display.

WOAF_NO_DESTROY—Prevents the Window Manager from calling the window's destructor. If this flag is set, the window can be removed from the screen display, but the programmer must call the destructor associated with the window to actually destroy it.

WOAF_NO_MOVE—Prevents the end user from changing the screen location of the window during an application.

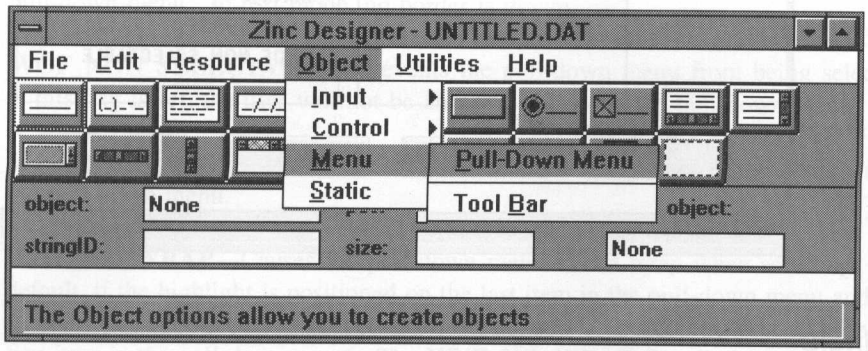
WOAF_NON_CURRENT—Prevents the window from being made current. However, clicking on the window with the mouse will call the window's user function.

WOAF_NO_SIZE—Prevents the end user from changing the size of the window during an application.

WOAF_TEMPORARY—Causes the window to only occupy the screen temporarily. Once another window is selected from the screen, the temporary window is removed from the Window Manager (i.e., erased from the display). Once removed, a temporary window will be destroyed if the **WOAF_NO_DESTROY** flag is not set.

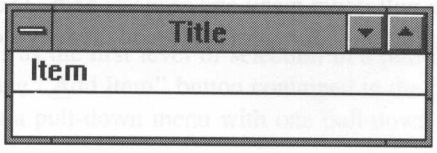
CHAPTER 28 - MENU OBJECTS

The menu category includes objects used specifically for creating menus that display selectable options. Selecting "Menu" causes the following associated menu to appear:



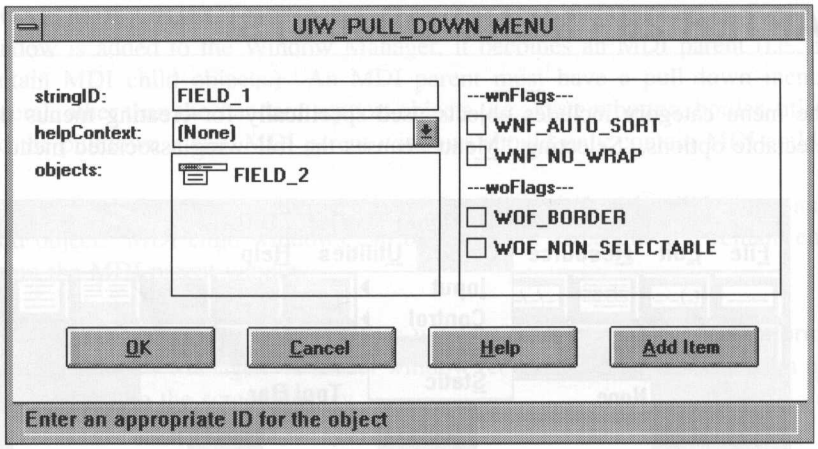
PULL-DOWN MENU

A pull-down menu acts as a structure for selectable menu items that appear in a single horizontal line. It automatically occupies the length of the top portion of the window to which it is attached. Selecting the "Pull-Down Menu" option and attaching it to a window causes the following object to appear:



A multi-level selectable menu is created by adding pull-down items and pop-up items to the pull-down menu. The pull-down menu object is automatically created with one pull-down item attached to it. (For information on adding more items, see "Add Item" of this section.)

To modify the pull-down menu, call its editor. The following window will appear:



stringID

Enter in this field a string that will distinguish the pull-down menu object from other objects.

helpContext

This field designates the help context to be associated with the pull-down menu. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the pull-down menu and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

objects

This field displays the pull-down items, listed by their string identifications, that are currently attached to the pull-down menu. Select one of these items, and its editor will appear. (See the Pull-down Item section below for more information on pull-down items.)

To delete a pull-down item from the object list, position the cursor (i.e., highlight bar) on the desired item and press <Ctrl+Del>. To reorder the pull-down items within the object list, position the cursor (i.e., highlight bar) on the desired item and press the <Ctrl+↑> or <Ctrl+↓> keys. Each time <Ctrl+↑> is pressed, the highlighted item will be moved up one space in the object list. Similarly, each time <Ctrl+↓> is pressed, the highlighted item will be moved down one space in the object list.

flags

The flags that control the presentation of the pull-down menu object are listed in the field on the right half of the window. The flags are:

WOF_BORDER—In graphics mode, this flag draws a single line border around the pull-down menu. In text mode, no border is drawn.

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

WNF_AUTO_SORT—Assigns a compare function to alphabetize the items within the pull-down menu.

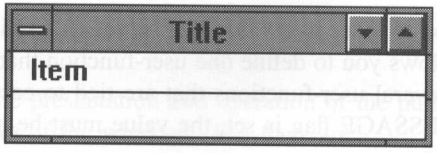
WNF_NO_WRAP—Causes the pull-down menu to not wrap when scrolling. By default, if the highlight is positioned on the last item in the pull-down menu and the right-arrow key is pressed, the pull-down menu will wrap and position itself on the first item in the pull-down menu. The **WNF_NO_WRAP** flag disables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

Add Item

Selecting this button causes a pull-down item to be added to the pull-down menu. (For more information on pull-down items, refer to the Pull-Down Item section below.)

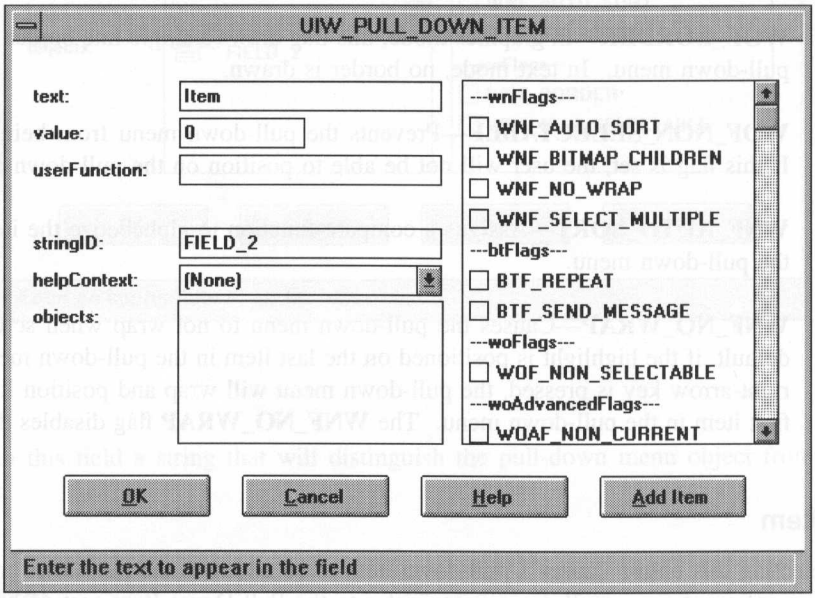
PULL-DOWN ITEM

A pull-down item serves as the first level of selection in a pull-down menu. It can only be created by selecting the "Add Item" button contained in the pull-down menu's editor. The figure below shows a pull-down menu with one pull-down item attached to it:



The multi-level effect of a pull-down menu is further achieved by adding pop-up items to the pull-down item. (For more information, see "Add Item" of this section.)

To modify the pull-down item, including adding pop-up items to it, the editor must be called. This can only be done by selecting the item from the "objects" field of the pull-down menu's editor. Upon doing so, the following window appears:



text

Enter in this field text exactly as you want it to appear on the pull-down item. It will be automatically centered vertically.

value

This field allows you to enter in a value that serves as a unique identification for a pull-down item. This allows you to define one user-function that looks at the pull-down item values, instead of several user-functions that are tied to each pull-down item object. If the BTF_SEND_MESSAGE flag is set, the value must be an event type.

userFunction

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

NOTE: Do not attach a user function to a pull-down item that has any sub pop-up items attached to it or that has the BTF_SEND_MESSAGE flag selected. In either case, the user function will not be called.

stringID

Enter in this field a string that will distinguish the pull-down item object from other objects.

helpContext

This field designates the help context to be associated with the pull-down item. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the pull-down item and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

objects

This field displays the pop-up items, listed by their string identifications, that are currently attached to the pull-down item. Select one of these items, and its editor will appear. (Refer to the Pop-Up Item section below for more information on pop-up items.)

To delete a pop-up item from the object list, position the cursor (i.e., highlight bar) on the desired item and press <Ctrl+Del>. To reorder the pop-up items within the object list, position the cursor (i.e., highlight bar) on the desired item and press the <Ctrl+↑> or <Ctrl+↓> keys. Each time <Ctrl+↑> is pressed, the highlighted item will be moved up one space in the object list. Similarly, each time <Ctrl+↓> is pressed, the highlighted item will be moved down one space in the object list.

flags

The flags that control the presentation and operation of the pull-down item are listed in the field on the right half of the window. The flags are:

WNF_AUTO_SORT—Assigns a compare function to alphabetize the items within the pull-down item's sub-menu.

WNF_BITMAP_CHILDREN—Used to denote that some of the pull-down item's pop-up items contain bitmaps. This flag must be set if the pull-down item's sub-menu will contain non-string objects.

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

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

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

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

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

Add Item

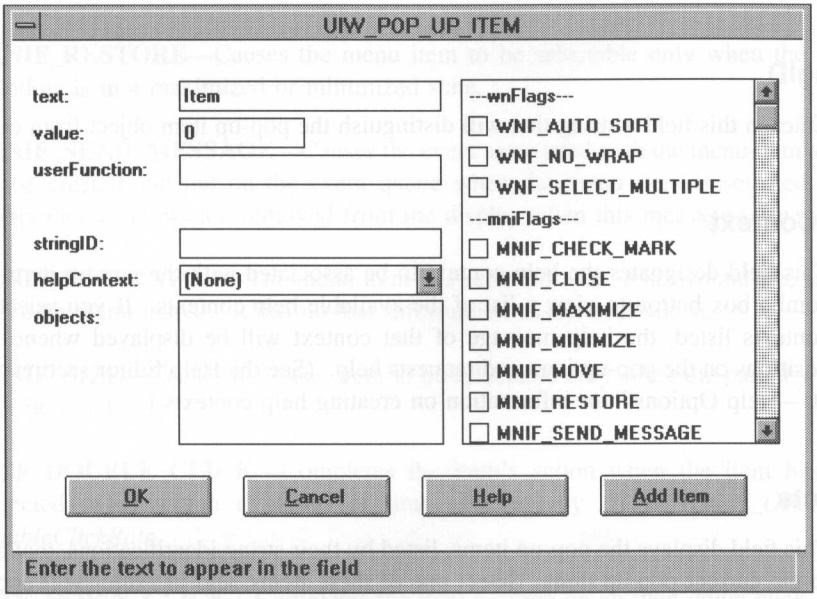
Selecting this button causes a pop-up item to be added to the pull-down item. (For more information on pop-up items, refer to the Pop-Up Item section below.)

POP-UP ITEM

A pop-up item is used to display and select options associated with a list of menu items. It can be attached to a pull-down item (as the second level of selection within a pull-down menu), or to another pop-up item. It can only be created by selecting the "Add Item" button contained in either the pull-down item's editor or the pop-up item's editor.

The multi-level effect of a pop-up menu or a pull-down menu is further achieved by adding sub-pop-up items to the parent pop-up item. (For more information, see "Add Item" of this section.) Zinc Designer will allow you to continue adding additional levels as long as there is available memory for them.

To modify the pop-up item, the editor must be called. This can only be done by selecting the item from the "objects" field of either the pop-up menu's editor or the pull-down item's editor. Upon doing so, the following window appears:



text

Enter in this field text exactly as you want it to appear on the pop-up item. It will be automatically left justified.

value

This field allows you to enter in a value that serves as a unique identification for a pop-up item. This allows you to define one user-function that looks at the pop-up item values, instead of several user-functions that are tied to each pop-up item object. If the MNIF_-SEND_MESSAGE flag is set, the value must be an event type.

userFunction

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

NOTE: Do not attach a user function to a pop-up item that has any sub pop-up items attached to it or that has the BTF_SEND_MESSAGE flag selected. In either case, the user function will not be called.

stringID

Enter in this field a string that will distinguish the pop-up item object from other objects.

helpContext

This field designates the help context to be associated with the pop-up item. Select the combo box button to view a list of the available help contexts. If you select one of the contexts listed, the help message of that context will be displayed whenever the user positions on the pop-up item and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

objects

This field displays the pop-up items, listed by their string identifications, that are attached to the current pop-up item. Select one of these items, and its editor will appear.

To delete a pop-up item from the object list, position the cursor (i.e., highlight bar) on the desired item and press <Ctrl+Del>. To reorder the pop-up items within the object list, position the cursor (i.e., highlight bar) on the desired item and press the <Ctrl+↑> or <Ctrl+↓> keys. Each time <Ctrl+↑> is pressed, the highlighted item will be moved up one space in the object list. Similarly, each time <Ctrl+↓> is pressed, the highlighted item will be moved down one space in the object list.

flags

The flags that control the presentation and operation of the pop-up item are listed in the field on the right half of the window. The flags are:

MNIF_CHECK_MARK—Marks the first position of the menu item's string information with a check-mark if the item has been selected (i.e., the WOS_SELECTED status flag is set).

MNIF_MAXIMIZE—Causes the menu item to be selectable only when the parent window can be maximized.

MNIF_MINIMIZE—Causes the menu item to be selectable only when the parent window can be minimized.

MNIF_MOVE—Causes the menu item to be selectable only when the parent window can be moved.

MNIF_RESTORE—Causes the menu item to be selectable only when the parent window is in a maximized or minimized state.

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

MNIF_SEPARATOR—The menu item is a separator (i.e., a horizontal line used to separate menu items). It has no text information associated with it.

MNIF_SIZE—Causes the menu item to be selectable only when the parent window can be sized.

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

BTF_DOWN_CLICK—Completes the item's action on an item down-click, rather than on a down-click and release action.

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

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

WOF_BORDER—Draws a single line border around the pop-up item. In text mode, no border is drawn.

WOF_JUSTIFY_CENTER—Center-justifies the text information associated with the pop-up item.

WOF_JUSTIFY_RIGHT—Right-justifies the text information associated with the pop-up item.

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

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

WOF_NON_SELECTABLE—Prevents the pop-up item from being selected.

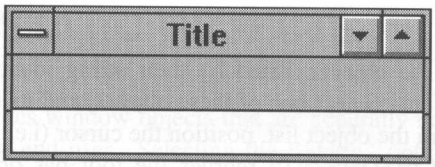
WOAF_NON_CURRENT—Prevents the item from being made current. If this flag is set, users will not be able to select the item from the keyboard nor with the mouse.

Add Item

Selecting this button causes an additional pop-up item to be added to the current pop-up item.

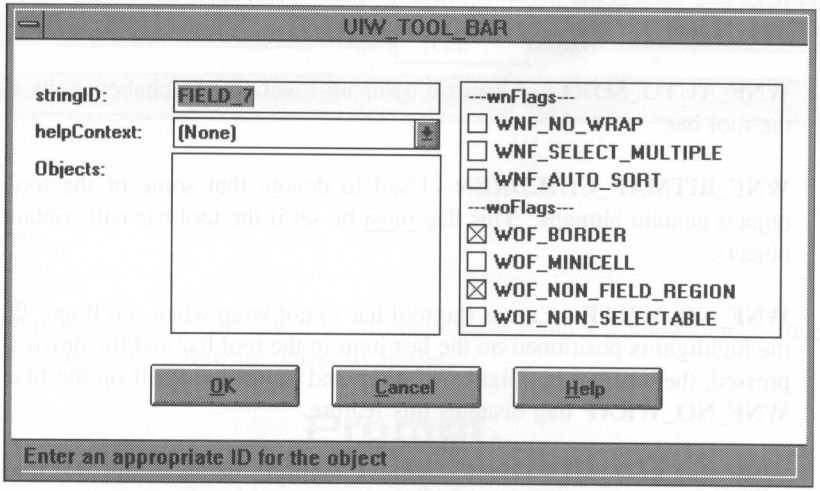
TOOL BAR

A tool bar is used as a controlling structure for a set of selectable window objects. It differs from the pull-down menu in that a variety of objects can be added to it—not just string items. The tool bar will automatically occupy the upper-most area available in a window, positioning itself directly below the pull-down menu, if one exists. Multiple tool bars may be added to a window. Selecting "Tool Bar" and attaching it to a window causes the following object to appear:



An object can be added to the tool bar by selecting the desired object from the control window's menu or object bar and placing it on the resource window's tool bar. The control window's object bar itself is an example of a group of bitmapped buttons that have been attached to a tool bar.

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



stringID

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

helpContext

This field designates the help context to be associated with the tool bar. Select the combo box button to view a list of the available help contexts. If you select one of the contexts listed, the help message of that context will be displayed whenever the user positions on the tool bar and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

objects

This field displays the objects, listed by their string identifications, that are currently attached to the tool bar. Select one of these objects, and its editor will appear. To delete a tool bar object from the object list, position the cursor (i.e., highlight bar) on the desired object and press <Ctrl+Del>. To reorder the tool bar objects within the object list, position the cursor (i.e., highlight bar) on the desired object and press the <Ctrl+↑> or <Ctrl+↓> keys. Each time <Ctrl+↑> is pressed, the highlighted object will be moved up one space in the object list. Similarly, each time <Ctrl+↓> is pressed, the highlighted object will be moved down one space in the object list.

flags

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

WNF_AUTO_SORT—Assigns a compare function to alphabetize the items within the tool bar.

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

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

WNF_SELECT_MULTIPLE—Allows multiple items within the tool bar to be selected.

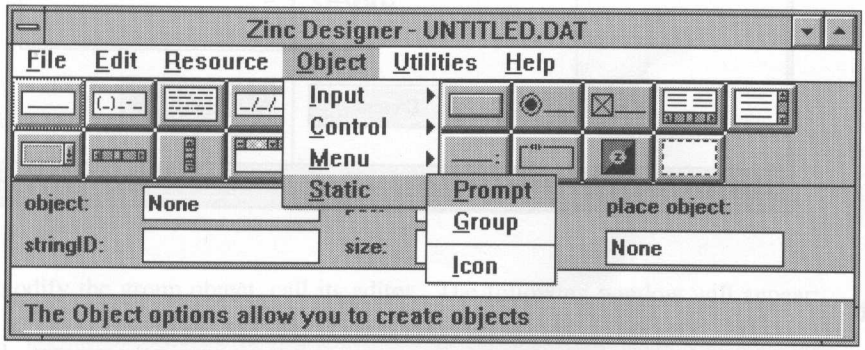
WOF_BORDER—In graphics mode, this flag draws a single line border around the tool bar. In text mode, no border is drawn.

WOF_NON_FIELD_REGION—The tool bar is not a form field. If this flag is set the tool bar will occupy the top-most space within the parent window.

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

CHAPTER 29 - STATIC OBJECTS

The static category includes window objects that are generally not designed to be edited nor interacted with by an end user. Selecting the "Static" option causes the following associated menu to appear:

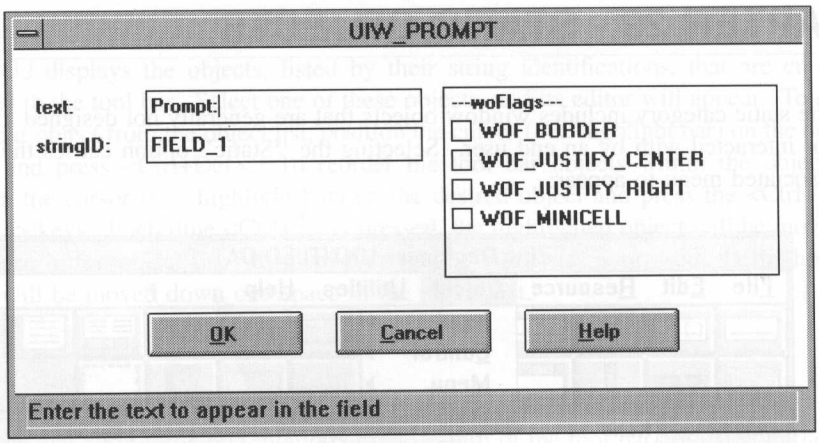


PROMPT

A prompt object is used to provide lead information for another window object. Selecting "Prompt" causes the following object to appear:

Prompt

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



text

Enter in this field a text string exactly as you want it to appear in the prompt. It will be automatically centered vertically. If either the WOF_JUSTIFY_CENTER or the WOF_-JUSTIFY_RIGHT flag is set and text string is longer than the length of the prompt field, the field must be sized in order to display the entire text.

stringID

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

flags

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

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

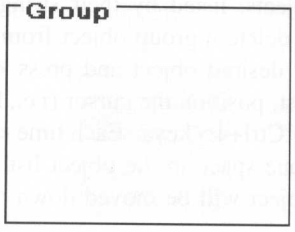
WOF_JUSTIFY_CENTER—Center-justifies the text within the prompt.

WOF_JUSTIFY_RIGHT—Right-justifies the text within the prompt.

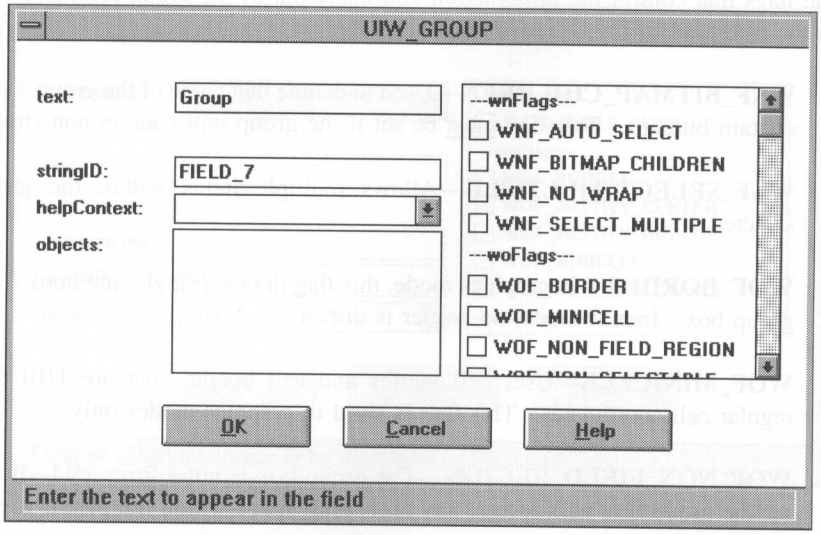
WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell coordinates. This flag is valid in graphics modes only.

GROUP

The group object is used to allow physical grouping of window objects. For example, a series of radio buttons can be grouped together by first creating a group object and then adding the radio buttons. Selecting "Group" causes the following object to appear:



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



text

Enter in this field text exactly as you want it to appear in the upper left corner of the group object's border. If the text string is longer than the width of the group box, only the portion that fits will be displayed.

stringID

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

objects

This field displays the objects, listed by their string identifications, that are currently attached to the group. To delete a group object from the object list, position the cursor (i.e., highlight bar) on the desired object and press <Ctrl+Del>. To reorder the group objects within the object list, position the cursor (i.e., highlight bar) on the desired object and press the <Ctrl+↑> or <Ctrl+↓> keys. Each time <Ctrl+↑> is pressed, the highlighted object will be moved up one space in the object list. Similarly, each time <Ctrl+↓> is pressed, the highlighted object will be moved down one space in the object list.

flags

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

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

WNF_SELECT_MULTIPLE—Allows multiple items within the group to be selected.

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

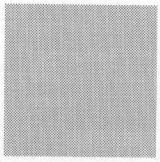
WOF_MINICELL—Uses cell widths and cell heights that are 1/10 the size of regular cell coordinates. This flag is valid in graphics modes only.

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

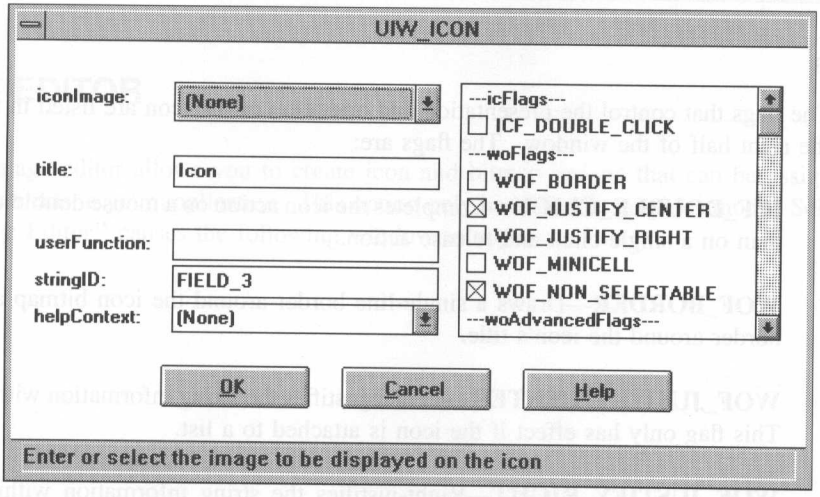
WOF_NON_SELECTABLE—Prevents the group from being selected. If this flag is set, the user will not be able to position on the group.

ICON

An icon is used to display a 32x32 pixel bitmap image to the screen. It is part of the static category because it is often present in an application as an indicator of some sort that cannot be interacted with; however, an icon can also be created for interaction purposes, such as a question mark icon that displays help when selected. Selecting "Icon" causes the following object to appear:



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



iconImage

This field designates the image to be associated with the icon. Select the combo box button to view a list of the available images. If you select one of the images listed, it will be displayed on the icon. (See the Image Editor section of "Chapter 30—Utilities Options" for information on creating bitmap images.)

title

Enter in this field the text exactly as you want it to appear in the rectangular region below the icon. If you do not want a title for the icon, delete the default string "Icon."

stringID

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

helpContext

This field designates the help context to be associated with the icon. Select the combo box button to view a list of the available help contexts. If you select one of the help contexts listed, the help message of that context will be displayed whenever the user positions on the icon and requests help. (See the Help Editor section of "Chapter 30—Help Options" for information on creating help contexts.)

flags

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

ICF_DOUBLE_CLICK—Completes the icon action on a mouse double-click, rather than on a single-click and release action.

WOF_BORDER—Draws a single-line border around the icon bitmap and another border around the icon's title.

WOF_JUSTIFY_CENTER—Center-justifies the string information within the icon. This flag only has effect if the icon is attached to a list.

WOF_JUSTIFY_RIGHT—Right-justifies the string information within the icon. This flag only has effect if the icon is attached to a list.

WOF_NON_SELECTABLE—Indicates that the icon object cannot be selected. If this flag is set, the user will not be able to select the icon.

CHAPTER 30 - UTILITIES OPTIONS

The utilities category provides options that allow you to create images and help utilities to be used throughout your application. Selecting "Utilities" causes the following menu to appear:

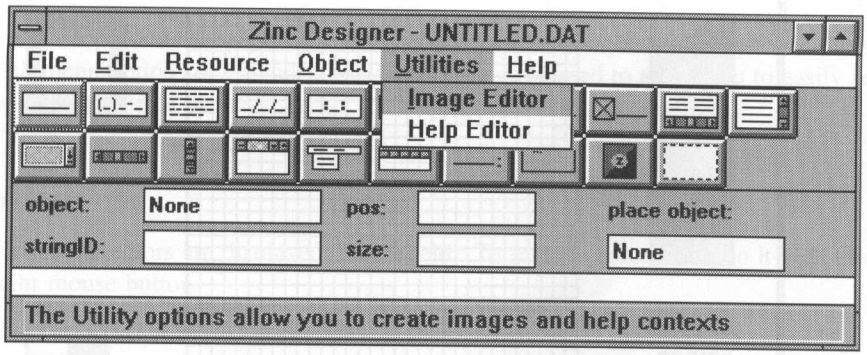
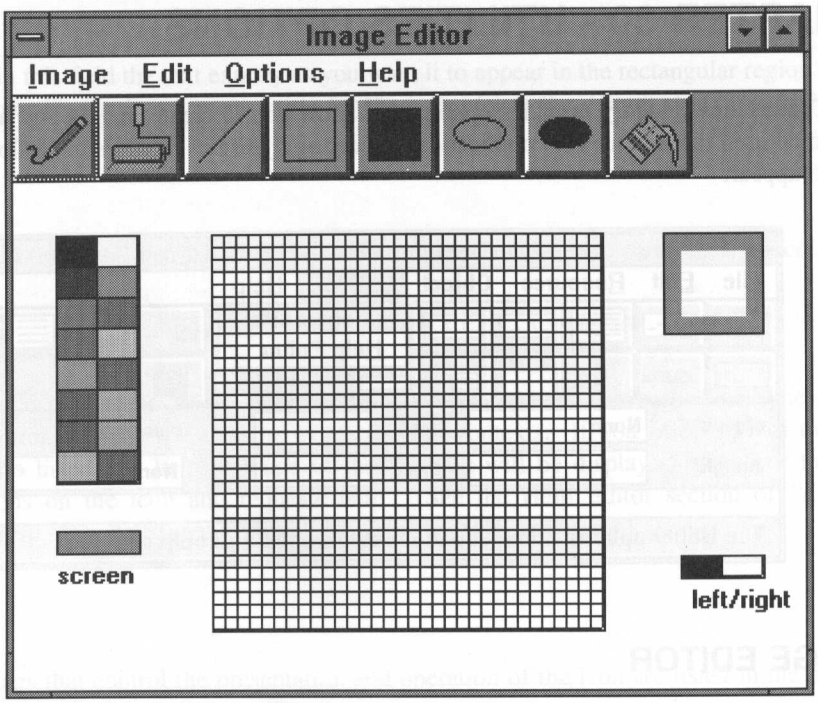


IMAGE EDITOR

The image editor allows you to create icon and bitmap images that can be assigned to other objects in your application. This option only has effect in graphics mode. Selecting "Image Editor" causes the following window to appear:



The menu bar

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

Image—This menu consists of options that control the creation of images and exiting the Image Editor. The selectable items on this menu are: New..., Load..., Save, Save As..., Import, Delete... and Exit.

Edit—This menu consists of options that edit images. The edit options are: Undo, Clear, Pencil, Brush, Line, Rectangle, Rectangle - solid, Ellipse, Ellipse - solid and Fill.

Options—This menu consists of option settings. The options are: Grid and Brush Size.

Help—This option provides general help for the Image Editor.

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

The tool bar

The tool bar presents the pencil, brush, line, rectangle, rectangle solid, ellipse, ellipse solid and fill menu options of the pull-down menu. It is designed to allow you to easily select these options with a mouse.

The color palette

The available colors are displayed in this field. To select a color, click on it with the left or right mouse button.

screen

This field is used when you want to have part of your image to be transparent (i.e., to show through to the screen behind it). Whichever mouse button is used to click on this field will have the ability to draw the transparent region. For example, if you want to create a window icon that displays the area underneath it, you could draw the frame for the window with a color from the color field and then click on the "Screen" field and fill the frame in.

The drawing field

This field is the drawing area where you create your icon. You can paint one pixel at a time by positioning on it and pressing a mouse button, or you can paint in continuous motion by holding down a mouse button and dragging the cursor.

The image field

This area displays the image in its actual size as you create it.

left/right

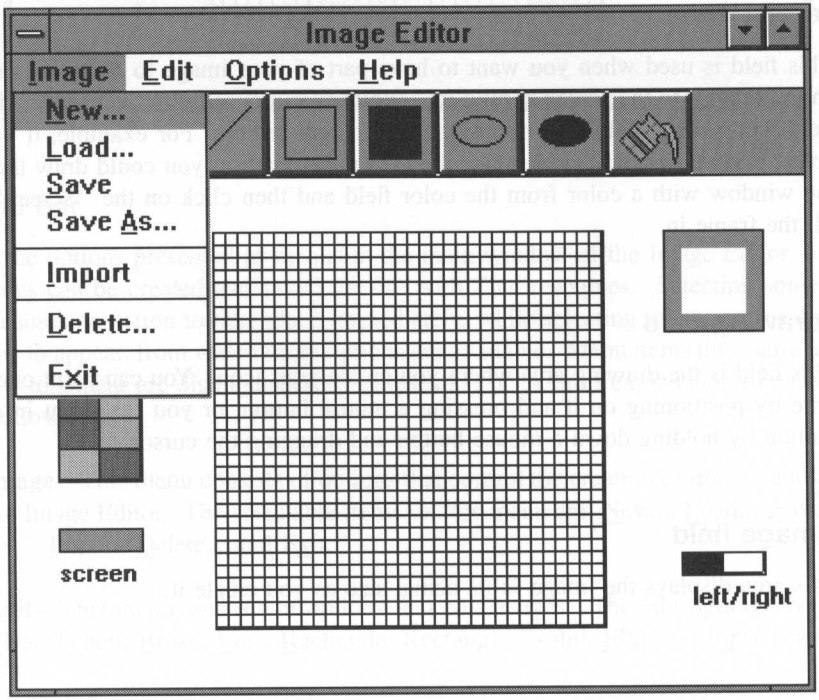
The box below "left" displays the color selected by the left mouse button. When you position the cursor anywhere in the "bitmap" area and press the left mouse button, the color shown in the "left" box will be painted onto the pixel underneath the cursor.

The box below "right" displays the color selected by the right mouse button. When you position the cursor anywhere in the "bitmap" area and press the right mouse button, the color shown in the "right" box will be painted onto the pixel underneath the cursor.

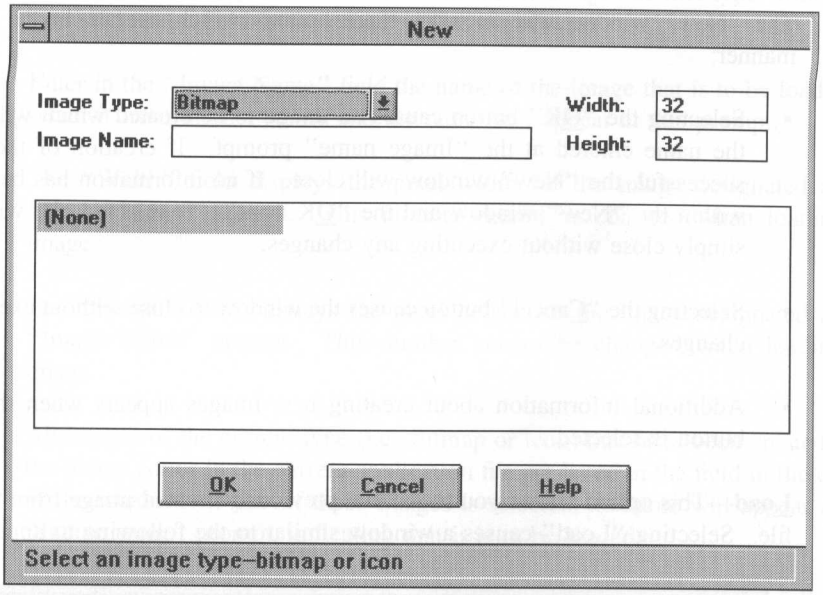
If either the right or the left mouse buttons have selected the "Screen" field instead of a color from the colors field, the appropriate box of the "left/right" field will be grey.

Image options

The image options of the window's pull-down menu control the general operations of the image editor. Selecting "Image" causes the following menu to appear:



New—This option allows you to create a new image. Selecting it causes a window similar to the following to appear:



Interaction with the fields of the "New" window is accomplished as follows:

- The "Image Type" field designates the type of image—bitmap or icon—to be created. Select the combo box button or press the <down arrow>; then select one of these two options. If the image will be designed for use with an icon, select "Icon." For use with all other objects, select "Bitmap."

- If you want to create a new image, enter the name for the new image in the "Image Name" field.

- Enter in the "Width" field the desired pixel width for the image. If the image type is icon, the width must be 32.

- Enter in the "Height" field the desired pixel height for the image. If the image type is icon, the width must be 32.

- Other images of the current type (i.e., bitmap or icon) that have been created with the image editor in the current application file are listed in the field in the center of the window. If one of these images is selected, its name will appear at the "Image name" prompt, indicating that it is to be loaded. (For more

information on loading a previously created image, see the explanation for the "Load" option below.)

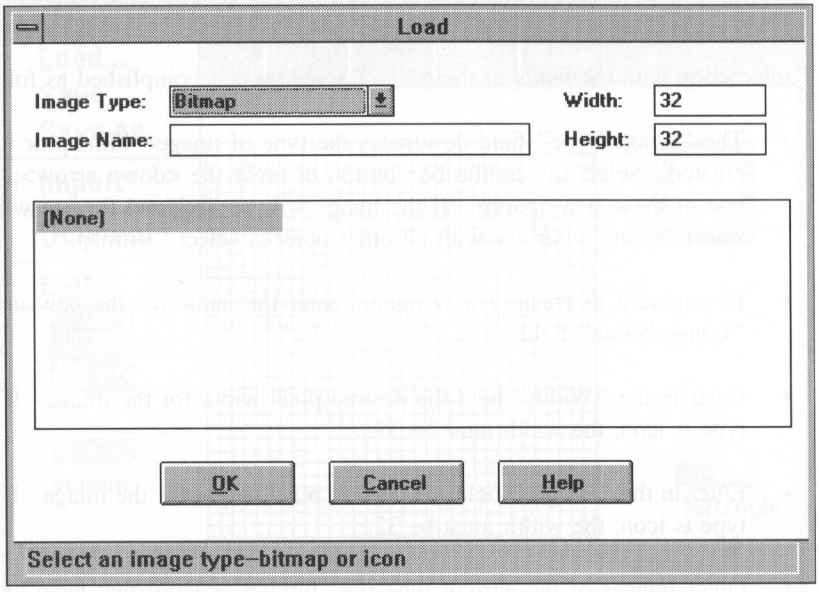
The "New" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes an image to be created which will be given the name entered at the "Image name" prompt. If creation of the image is successful, the "New" window will close. If no information has been entered within the "New" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about creating new images appears when the "Help" button is selected.

Load—This option allows you to recall a previously created image from the current file. Selecting "Load" causes a window similar to the following to appear:



Interaction with the fields of the "Load" window is accomplished as follows:

- The "Image Type" field designates the type of the image—bitmap or icon—to be loaded. Select the combo box button or press the <down arrow> and select one of these two options to view the available images of that type, which will be listed in the field in the center of the window.

- Enter in the "Image Name" field the name of the image that is to be loaded, or select it from the list below and the name will appear at this prompt.

- The "Width" field displays the pixel width for the image designated at the "Image Name" prompt. This number cannot be changed when loading an image.

- The "Height" field displays the pixel height for the image designated at the "Image Name" prompt. This number cannot be changed when loading an image.

- All images of the current type (i.e., bitmap or icon) that have been created with the image editor in the current application file are listed in the field in the center of the window. If one of these images is selected, its name will appear at the "Image name" prompt, indicating that it is to be loaded.

The "Load" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes the image designated at the "Image name" prompt to be loaded. If the image is successfully loaded, the "Load" window will close. If no information has been entered within the "Load" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

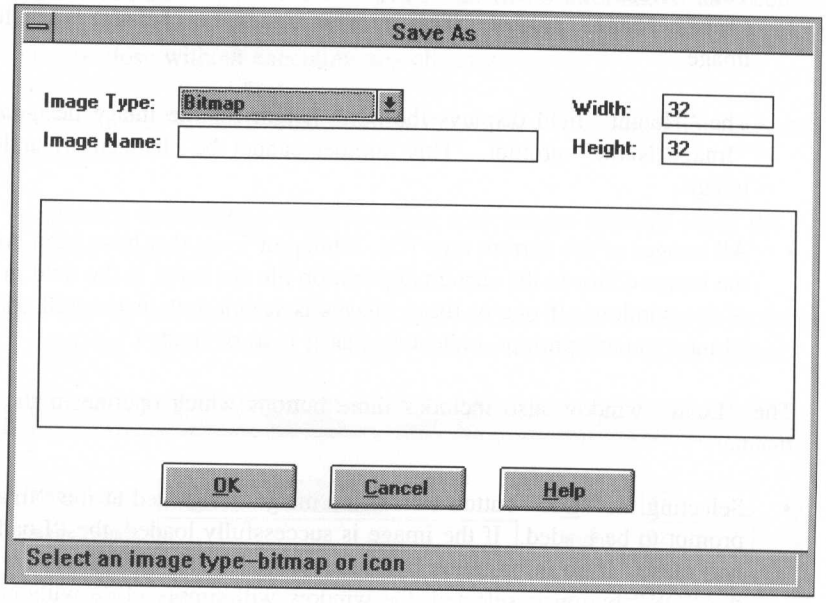
- Additional information about loading images appears when the "Help" button is selected.

Once the image has been loaded and appears in the drawing area, it can be modified in any way. When the Image | Save option is subsequently selected, the image will be saved in its present condition, replacing the original version. (Refer to the Save and Save As options of this section for more information on saving images.)

Save—Selecting this option causes the current image to be saved in its present condition. If the image has never been named, the "Save As" window will appear

and allow you to name it by entering a name at the "Image Name" prompt. When you select the "OK" button, the "Save As" window will close and the image will be saved under that name. (See the Save As section for further details on how to save a image for the first time.)

Save As—This option allows you to either save an image that has not been previously named or to save the current image under another name. Selecting "Save As" causes the following window to appear:



Interaction with the fields of the "Save As" window is accomplished as follows:

- The "Image Type" field designates the type of image—bitmap or icon—to be saved. Select the combo box button or press the <down arrow>; then select one of these two options. If the image will be designed for use with an icon, select "Icon." For use with all other objects, select "Bitmap."

- Enter the name for the image in the "Image Name" field.

- Enter in the "Width" field the desired pixel width for the image. If the image type is icon, the width must be 32.

- Enter in the "Height" field the desired pixel height for the image. If the image type is icon, the width must be 32.

- Other images of the current type (i.e., bitmap or icon) that have been created with the image editor in the current application file are listed in the field in the center of the window. If one of these images is selected, its name will appear at the "Image name" prompt, and the current image will replace the previous information of the specified image when the "OK" button is selected.

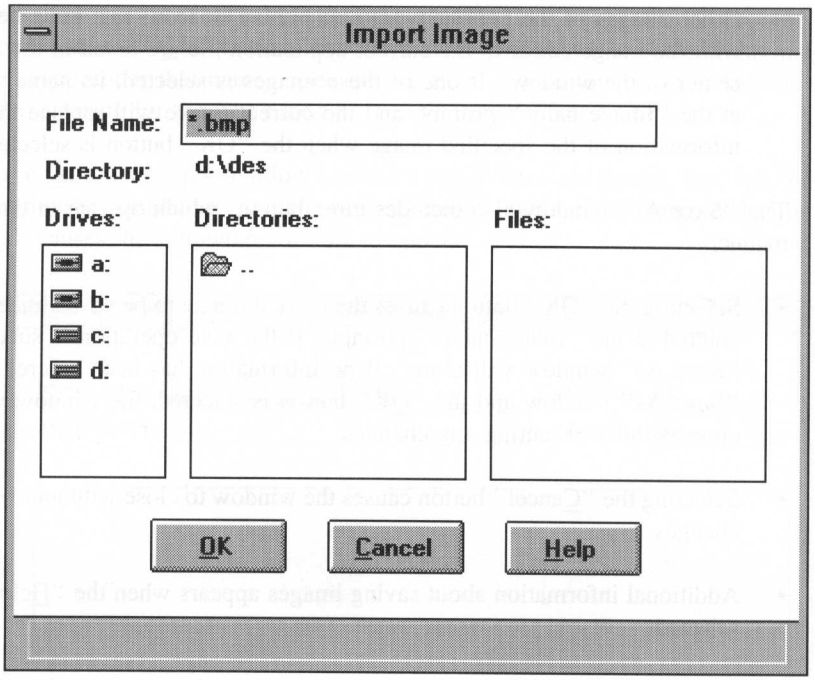
The "Save As" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes the current image to be saved under the name entered at the "Image name" prompt. If the save operation is successful, the "Save As" window will close. If no information has been entered within the "Save As" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about saving images appears when the "Help" button is selected.

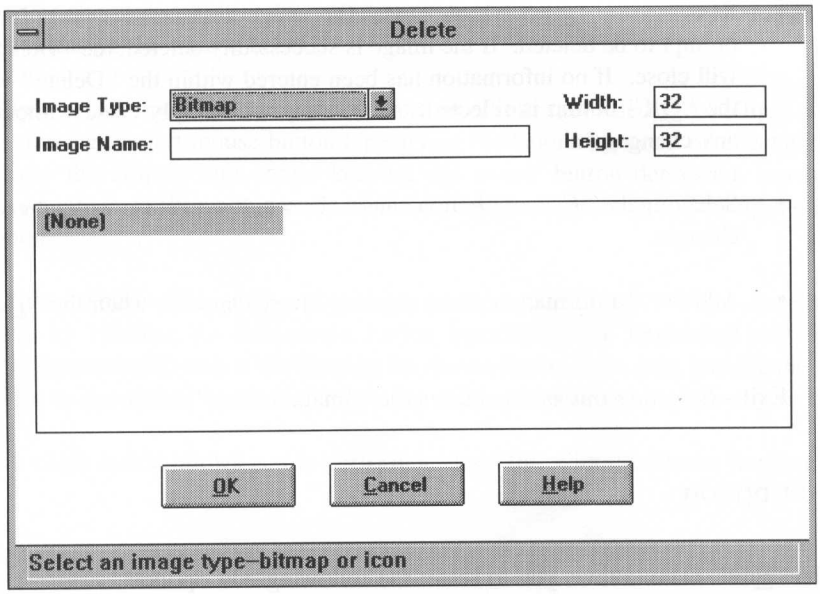
Import—This option allows you to import an image from another file. Images can be imported from **.BMP**, **.DAT** or **.ICO** files. Selecting "Import" causes a window similar to the following to appear:



Interaction with the fields of the "Import Image" window is accomplished as follows:

- The name of the file to be imported is typed in the field adjacent to the prompt "File name:" or it can be selected from the "Files" list.
- Pressing "Ok" will cause the image to be imported into the "Image Editor" window.
- Pressing "Cancel" will cause the "Import Image" window to be closed. No change is made in the "Image Editor" window.
- Additional information about importing images appears when the "Help" button is selected.

Delete—This option allows you to delete an image. Selecting "Delete" causes a window similar to the following to appear:



Interaction with the fields of the "Delete" window is accomplished as follows:

- The "Image Type" field designates the image type of the image—bitmap or icon—to be deleted. Select the combo box button or press the <down arrow> and select one of these two options to view the available images of that type.

- Enter in the "Image Name" field the name of the image to be deleted, or select it from the list in the center of the window and the name will appear at this prompt.

- The "Width" field displays the pixel width for the image designated at the "Image Name" prompt. This field is not editable.

- The "Height" field displays the pixel height for the image designated at the "Image Name" prompt. This field is not editable.

- All images of the current type (i.e., bitmap or icon) that have been created with the image editor in the current application file are listed in the field in the center of the window. If one of these images is selected, its name will appear at the "Image Name" prompt, indicating that it is to be deleted.

The "Delete" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes the image designated at the "Image name" prompt to be deleted. If the image is successfully deleted, the "Delete" window will close. If no information has been entered within the "Delete" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about deleting images appears when the "Help" button is selected.

Exit—Selecting this option closes the "Image Editor" window.

Edit option

The "Edit" menu allows you to select the following edit options:

Undo—This option allows you to undo the last modification (during the current edit) to an image.

Clear—This option allows you to clear the image grid.

Pencil—This option sets the current drawing pen to the width of a pencil (i.e., one pixel wide).

Brush—This option sets the current drawing brush to the width specified in "Options | Brush Size". The default width is 3 pixels.

Line—This option allows you to draw a line. A line is drawn by clicking the left mouse button (specifying the beginning point) and moving the mouse cursor to the ending point and releasing the mouse button. The width of the line depends upon the brush size.

Rectangle—This option allows you to create a rectangle (not filled). A rectangle is made by clicking the left mouse button (specifying the beginning point) and, while keeping the mouse button depressed, moving the mouse cursor to the ending point and releasing the mouse button.

Rectangle - Solid—This option allows you to create a rectangle (filled). A rectangle is made by clicking the left mouse button (specifying the beginning point) and, while

keeping the mouse button depressed, moving the mouse cursor to the ending point and releasing the mouse button.

Ellipse—This option allows you to create an ellipse (not filled). An ellipse is made by clicking the left mouse button (specifying the beginning point of a rectangle that defines the ellipse) and, while keeping the mouse button depressed, moving the mouse cursor to the ending point of the defining rectangle and releasing the mouse button.

Ellipse - Solid—This option allows you to create an ellipse (filled). An ellipse is made by clicking the left mouse button (specifying the beginning point of the defining rectangle) and, while keeping the mouse button depressed, moving the mouse cursor to the ending point of the defining rectangle and releasing the mouse button.

Fill—This option allows you to perform a 'flood fill' with the current drawing color.

Options option

The "Options" menu option allows you to change the following edit options:

Grid—This option, when selected, displays a grid on the image drawing field. The grid is displayed by default.

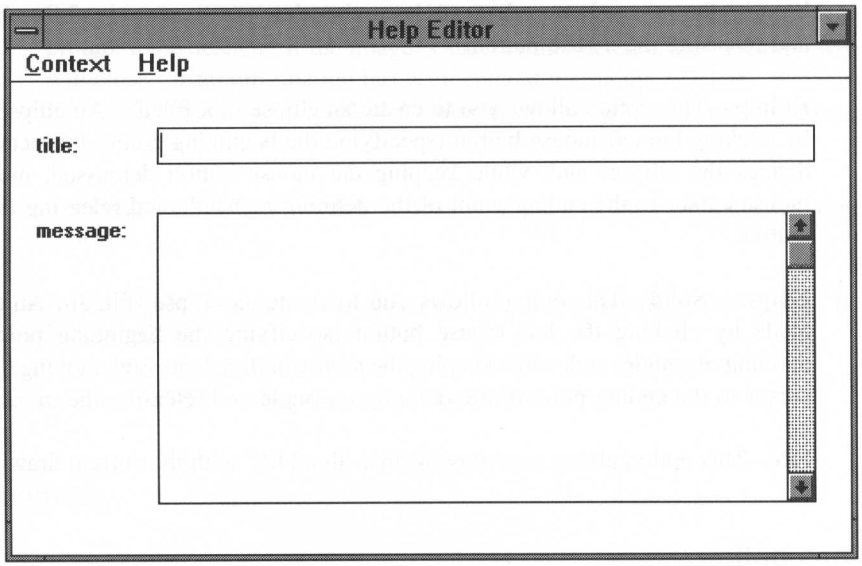
Brush Size—This option allows you to change the current size of the drawing brush. The available sizes are: 2x2, 3x3, 4x4 and 5x5.

Help option

The help option, when selected, displays help on creating images using the image editor.

HELP EDITOR

The help editor allows you to create help contexts to be used throughout your application. Selecting "Help Editor" causes the following window to appear:



The menu bar

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

Context—This menu consists of options that control the creation of help contexts and exiting the Help Editor. The selectable items on this menu are: New..., Load..., Save, Save As..., Delete... and Exit.

Help—This option provides general help for the Help Editor.

All of these menu items are discussed in more detail in sections that follow.

title

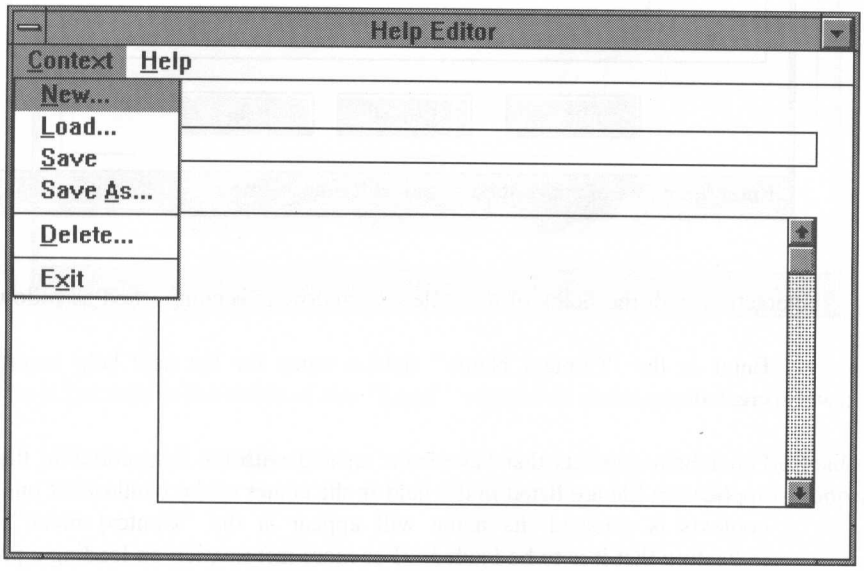
Enter in this field a title to be displayed on the help window's title bar.

message

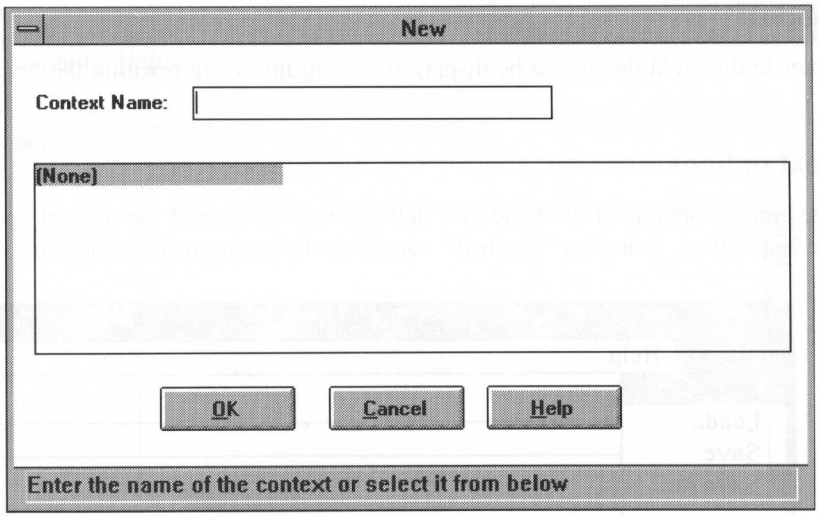
Enter in this field the text to be displayed as help information within the help window.

Context options

The context options of the window's pull-down menu control the general operations of the help editor. Selecting "Context" causes the following menu to appear:



New—This option allows you to create a new help context. Selecting it causes a window similar to the following to appear:



Interaction with the fields of the "New" window is accomplished as follows:

- Enter in the "Context Name" field a name for the new help context to be created.

- Other help contexts that have been created with the help editor in the current application file are listed in the field in the center of the window. If one of these contexts is selected, its name will appear at the "Context name" prompt, indicating that it is to be loaded. (For more information on loading a previously created context, see the explanation for the "Load" option below.)

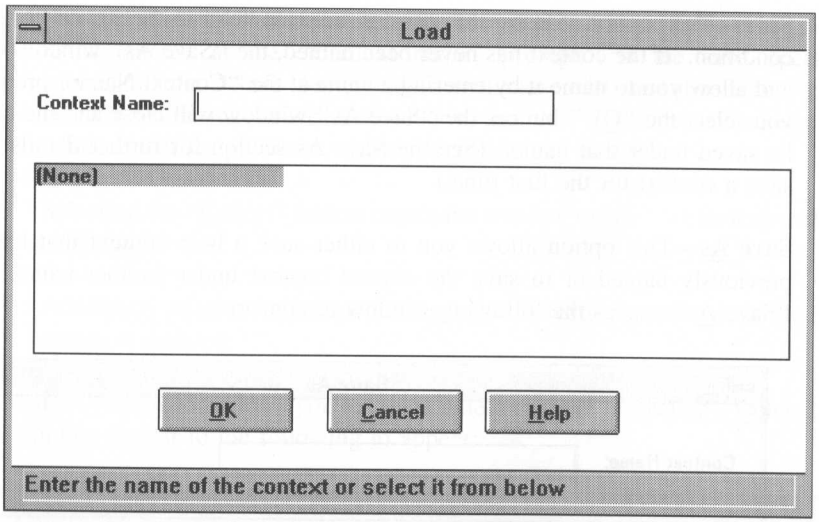
The "New" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes a help context to be created which will be given the name entered at the "Context name" prompt. If creation of the context is successful, the "New" window will close. If no information has been entered within the "New" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about creating new contexts appears when the "Help" button is selected.

Load—This option allows you to recall a previously created context of the current file. Selecting "Load" causes a window similar to the following to appear:



Interaction with the fields of the "Load" window is accomplished as follows:

- Enter in the "Context Name" field the name of the context to be loaded, or select it from the list in the center of the window and the name will appear at this prompt.

- Other help contexts that have been created with the help editor in the current application file are listed in the field in the center of the window. If one of these contexts is selected, its name will appear at the "Context name" prompt, indicating that it is to be loaded.

The "Load" window also includes three buttons which operate in the following manner:

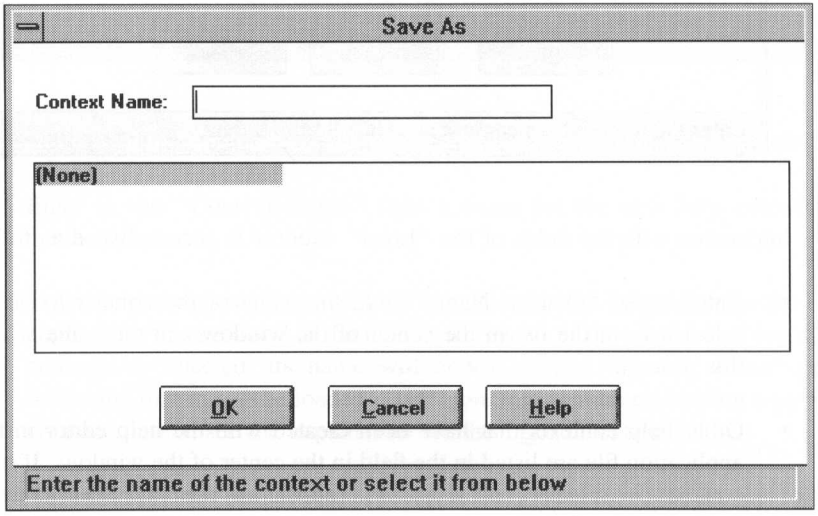
- Selecting the "OK" button causes the help context indicated at the "Context name" prompt to be loaded. If the context is successfully loaded, the "Load" window will close. If no information has been entered within the "Load" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about loading contexts appears when the "Help" button is selected.

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

Save As—This option allows you to either save a help context that has not been previously named or to save the current context under another name. Selecting "Save As" causes the following window to appear:



Interaction with the fields of the "Save As" window is accomplished as follows:

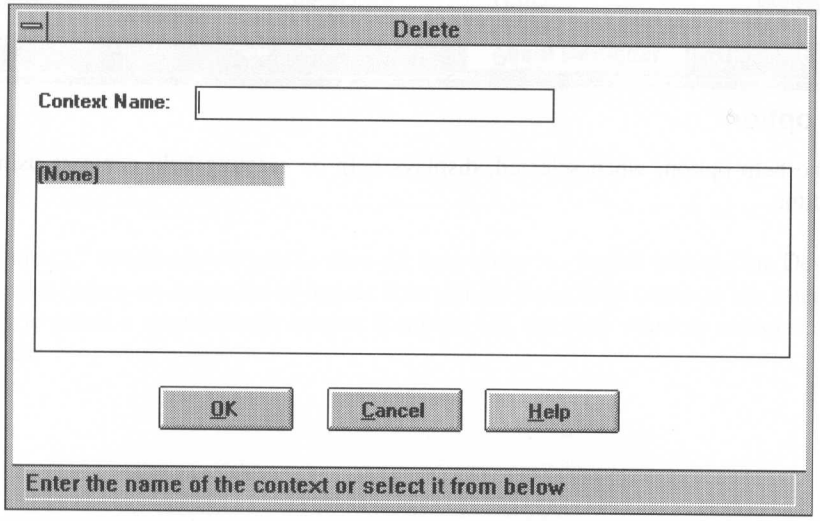
- Enter in the "Context Name" field a name for the current context or select one from the list in the center of the window and the name will appear at this prompt.

- Other help contexts that have been created with the help editor in the current application file are listed in the field in the center of the window. If one of these contexts is selected, its name will appear at the "Context name" prompt, indicating that the current context is to replace the previous information.

The "Save As" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes the current help context to be saved under the name entered at the "Context name" prompt. If the save operation is successful, the "Save As" window will close. If no information has been entered within the "Save As" window and the "OK" button is selected, the window will simply close without executing any changes.
- Selecting the "Cancel" button causes the window to close without executing any changes.
- Additional information about saving help contexts appears when the "Help" button is selected.

Delete—This option allows you to delete a help context. Selecting "Delete" causes a window similar to the following to appear:



Interaction with the fields of the "Delete" window is accomplished as follows:

- Enter in the "Context Name" field the name of the help context to be deleted.
- Other help contexts that have been created with the help editor in the current application file are listed in the field in the center of the window. If one of these

images is selected, its name will appear at the "Context name" prompt, indicating that it is to be deleted.

The "Delete" window also includes three buttons which operate in the following manner:

- Selecting the "OK" button causes the help context designated at the "Context name" prompt to be deleted. If the context is successfully deleted, the "Delete" window will close. If no information has been entered within the "Delete" window and the "OK" button is selected, the window will simply close without executing any changes.

- Selecting the "Cancel" button causes the window to close without executing any changes.

- Additional information about deleting contexts appears when the "Help" button is selected.

Exit—Selecting this option closes the "Help Editor" window.

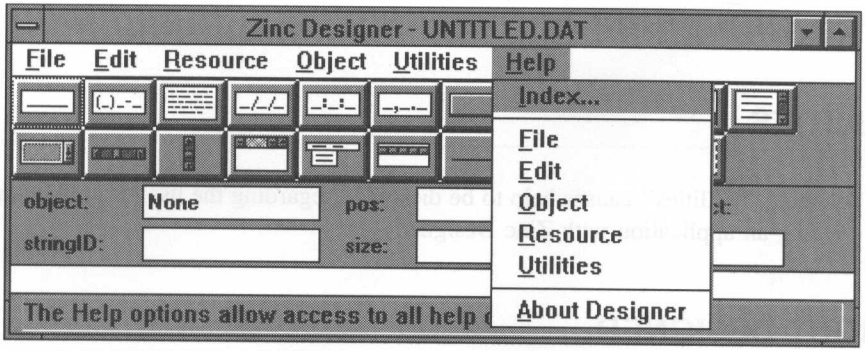
Help option

The help option, when selected, displays help on creating help contexts using the help editor.

CHAPTER 31 - HELP OPTIONS

The Help category is available so that you can receive help at any time during Zinc Designer's execution. The various options represent the different areas within Zinc Designer where help information is available.

Selecting "Help" causes the following menu to appear:



INDEX

The "Index..." option allows you to view all help contexts created within Zinc Designer. Selecting it causes an index list to appear from which these help contexts are selectable. When you select a specific help context from the list, the help window associated with it appears.

FILE

Selecting "File" causes help to be displayed regarding the use of File options in creating an application with Zinc Designer.

EDIT

Selecting "Edit" causes help to be displayed regarding the use of Edit options in creating an application with Zinc Designer.

OBJECT

Selecting "Object" causes help to be displayed regarding the use of Object options in creating an application with Zinc Designer.

RESOURCE

Selecting "Resource" causes help to be displayed regarding the use of Resource options in creating an application with Zinc Designer.

UTILITIES

Selecting "Utilities" causes help to be displayed regarding the use of Utilities options in creating an application with Zinc Designer.

ABOUT DESIGNER

Selecting "About Designer" causes information to be displayed regarding the general contents and specifics of Zinc Designer (e.g., the current version number and copyright information).

SECTION VIII
APPENDICES

APPENDIX A – COMPILER CONSIDERATIONS

This appendix explains the initial configuration you must implement in order to compile your applications with Zinc Application Framework.

Borland

The following libraries are for use with Borland:

- **DOS_ZIL.LIB** (basic DOS library)
- **D16_ZIL.LIB** (16-bit DOS library)
- **DOS_GFX.LIB** (DOS UI_GRAPHICS_DISPLAY)
- **D16_GFX.LIB** (DOS UI_GRAPHICS_DISPLAY for 16-bit DOS extender)
- **BC_LGFX.LIB** (Borland-specific GFX graphics library)
- **BC_16GFX.LIB** (Borland-specific GFX graphics library for 16-bit DOS extender)
- **DOS_BGI.LIB** (DOS UI_BGI_DISPLAY)
- **WIN_ZIL.LIB** (MS Windows library)
- **OS2_ZIL.LIB** (IBM OS/2 library)

The following table lists the types of applications that can be built using Borland and Zinc and the Zinc libraries that are required for each:

Type of application	Required libraries	Comments
DOS with BGI (also supports text mode)	DOS_ZIL.LIB DOS_BGI.LIB	Must also select Options|Linker|Libraries and turn on "Graphics Library" in IDE or link GRAPHICS.LIB in the makefile
DOS with GFX (also supports text mode)	DOS_ZIL.LIB DOS_GFX.LIB BC_LGFX.LIB	Must also select Options|Linker|Libraries and turn off "Graphics Library"
DOS 16-bit (PharLap) with GFX (also supports text mode)	D16_ZIL.LIB D16_GFX.LIB BC_16GFX.LIB	Must also select Options|Linker|Libraries and turn off "Graphics Library"
Windows	WIN_ZIL.LIB	
OS/2	OS2_ZIL.LIB	

For information on building applications with DOS-extenders, please see the **READ.ME** file.

Integrated Development Environment (IDE)

DOS

To compile applications for DOS in the IDE, the following options must be selected within the IDE:

- Select Options|Compiler|Code Generation.
  Choose the Large model.

- Select Options|Directories.
  In the include directory, enter the path of your Zinc include file. For example, if Zinc is installed on Drive C, the include file directory would be **C:\ZINC\INCLUDE**.

- Select Project|Open to create a project file.
  Enter both your source files and the proper Zinc libraries, as shown in the table above.

Windows

To compile applications for Windows in the IDE, the following options must be selected within the IDE:

- Select Options|Application.
  Choose Window App.

- Select Options|Compiler|Code Generation.
  Choose the Large model.

- Select Project|Open to create a project file.
  Enter both your source files, the proper Zinc library as shown above, the **.RC** file and the **.DEF** file.

NOTE: Although you can compile from within the IDE, you must be in Windows to actually run a Windows application.

OS/2

To compile applications for OS/2 in the IDE, follow the instructions found in the **READ.ME** file.

Makefiles

To compile applications using a makefile, the **TCC**, **BCC** or **BCCX** command-line compilers must be used. Each tutorial program has a sample makefile, **BORLAND.MAK**, that can be used as a template for other programs. The options listed next to CPP_OPTS in the makefile are recommended by Zinc. The makefile can be run by typing the following:

```
make -fborland.mak dos          (for DOS)
```

or

```
make -fborland.mak windows      (for Windows)
```

or

```
make -fborland.mak os2          (for OS/2)
```

Before using the makefile the following may need to be changed:

- Be sure to update your **TURBOC.CFG** file in the compiler's BIN directory. The following lines should be changed to reflect the path of the compiler and Zinc Application Framework:

```
-I.;C:\ZINC\INCLUDE;C:\BORLANDC\INCLUDE
-L.;C:\ZINC\LIB\BTCPP310;C:\BORLANDC\LIB
```

- The following line should also be changed in **TLINK.CFG** to reflect the path of the compiler and Zinc Application Framework:

```
-L.;C:\ZINC\LIB\BTCPP310;C:\BORLANDC\LIB
```

- In order to compile Microsoft Windows applications, you must be using Microsoft Windows Version 3.X and Borland C++ Version 3.1 or later. All of the options specified in the Borland sample makefile must be used. In addition, the -WE compiler option and /Twe link option, which build the application as a Windows executable program, must be included.

- In order to compile IBM OS/2 applications, you must be running IBM OS/2 Version 2.X and the Borland compiler for OS/2 2.X. All of the options specified in the Borland sample makefile must be used.

- The appropriate Zinc **.LIB** files, as shown in the table above, should be linked in.

Here is a sample "generic" makefile:

```
# GENERIC tutorial makefile

#    make -fborland.mak dos             (makes the DOS generic program)
#    make -fborland.mak windows         (makes the Windows generic program)
#    make -fborland.mak os2             (makes the OS/2 generic program)

# Be sure to update your TURBOC.CFG file to include the Zinc paths, e.g.:
#   -I.;C:\ZINC\INCLUDE;C:\BORLANDC\INCLUDE
#   -L.;C:\ZINC\LIB\BTCPP310;C:\BORLANDC\LIB
# and your TLINK.CFG file to include the Zinc paths, e.g.:
#   -L.;C:\ZINC\LIB\BTCPP310;C:\BORLANDC\LIB

## Compiler and linker: (Add -v to CPP_OPTS and /v to LINK_OPTS for debug.)

# ----- DOS compiler options-------------------------------------
DOS_CPP_OPTS=-c -ml -O -w
DOS_LINK_OPTS=/c /x
DOS_OBJS=c0l
# --- Use the next line for UI_GRAPHICS_DISPLAY ---
```

```
DOS_LIBS=dos_zil dos_gfx bc_lgfx emu mathl cl
# --- Use the next line for UI_BGI_DISPLAY ---
#DOS_LIBS=dos_zil dos_bgi graphics emu mathl cl

# ----- Windows compiler options--------------------------------
WIN_CPP_OPTS=-c -ml -O -WE -w
WIN_LINK_OPTS=/c /C /Twe /x
WIN_OBJS=c0wl
WIN_LIBS=win_zil mathwl import cwl

# ----- OS/2 compiler options----------------------------------
OS2_CPP_OPTS=-c
OS2_LINK_OPTS=/c /B:0x10000 /aa
OS2_OBJS=c02.obj
OS2_LIBS=os2_zil.lib c2.lib os2.lib

CPP=bcc
LINK=tlink

.cpp.obj:
    $(CPP) $(DOS_CPP_OPTS) {$< }

.cpp.obw:
    $(CPP) $(WIN_CPP_OPTS) -o$*.obw {$< }

.cpp.obo:
    $(CPP) $(OS2_CPP_OPTS) -o$*.obo {$< }

# ----- DOS ---------------------------------------------------
dos: generic.exe

generic.exe: generic.obj
    $(LINK) $(DOS_LINK_OPTS) @&&!
$(DOS_OBJS)+generic.obj
$*, ,$(DOS_LIBS)
!

# ----- Windows -----------------------------------------------
windows: wgeneric.exe

wgeneric.exe: generic.obw
    $(LINK) $(WIN_LINK_OPTS) @&&!
$(WIN_OBJS)+generic.obw
$*, ,$(WIN_LIBS),wgeneric.def
!
    rc wgeneric.rc $<

# ----- OS/2 --------------------------------------------------
os2: ogeneric.exe

ogeneric.exe: generic.obo
    $(LINK) $(OS2_LINK_OPTS) @&&!
$(OS2_OBJS)+generic.obo
$*, ,$(OS2_LIBS),ogeneric.def
!
    rc ogeneric.rc $<
```

Microsoft

The following libraries are for use with Microsoft:

- **DOS_ZIL.LIB** (basic DOS library)

- **D16_ZIL.LIB** (16-bit DOS library)
- **DOS_GFX.LIB** (DOS UI_GRAPHICS_DISPLAY)
- **D16_GFX.LIB** (DOS UI_GRAPHICS_DISPLAY for 16-bit DOS extender)
- **MS_LGFX.LIB** (Microsoft-specific GFX graphics library)
- **MS_16GFX.LIB** (Microsoft-specific GFX graphics library for 16-bit DOS extender)
- **DOS_MSC.LIB** (DOS UI_MSC_DISPLAY)
- **WIN_ZIL.LIB** (MS Windows library)
- **WNT_ZIL.LIB** (Windows NT library)

The following table lists the types of applications that can be built using Microsoft and Zinc and the Zinc libraries that are required for each:

Type of application	Required libraries	Comments
DOS with MSC graphics (also supports text mode)	DOS_ZIL.LIB DOS_MSC.LIB	Must also link in GRAPHICS.LIB
DOS with GFX (also supports text mode)	DOS_ZIL.LIB DOS_GFX.LIB MS_LGFX.LIB	Must not link in GRAPHICS.LIB
DOS 16-bit (PharLap) with GFX (also supports text mode)	D16_ZIL.LIB D16_GFX.LIB MS_16GFX.LIB	Must not link in GRAPHICS.LIB
Windows	WIN_ZIL.LIB	
Windows NT	WNT_ZIL.LIB	

Before using the Microsoft compiler the following may need to be changed:

- Change the environment variable for your include files' path by entering **`set include=`**, followed by the paths for your Microsoft include files and your Zinc include files. For example, if your include files are all on Drive C, enter:

```
set include=.;C:\ZINC\INCLUDE;C:\C700\INCLUDE
```

- Change the environment variable for the libraries by entering **set lib=**, followed by the paths for your Microsoft libraries and your Zinc libraries. For example, if your libraries are all on Drive C, enter:

```
set lib=.;C:\ZINC\LIB\MSCPP700;C:\C700\LIB
```

NOTE: Probably the easiest place to change the environment variables is in your **AUTOEXEC.BAT** file.

For information on building applications with DOS-extenders, please see the **READ.ME** file.

Programmers Workbench (PWB)

To compile DOS or Windows applications in the PWB, the following options must be selected within the PWB:

- Select Project|New Project.
  Select Set Project Template. Choose C++ and either DOS EXE or Windows 3.X EXE, as desired.
  Add your source files to the project.

- Select Options|Language Options|C++ Compiler Options.
  Choose the Large Memory Model.

- Select Options|Link Options.
  Set the stack size to 5120.
  Add the appropriate libraries (as shown in the table above).
  Select Additional Global Options and add /SEGMENTS:256 to the existing options list.

Makefiles

Each tutorial program has a sample makefile, **MICROSFT.MAK**, that can be used as a template for other programs. The options listed next to CPP_OPTS in the makefile are recommended by Zinc. The makefile can be run by typing the following:

```
nmake -fmicrosft.mak dos        (for DOS)
nmake -fmicrosft.mak windows    (for Windows)
nmake -fmscwnt.mak winnt        (for Windows NT)
```

- In order to compile Microsoft Windows applications, you must be using Microsoft Windows Version 3.X. All of the options specified in the Microsoft sample makefile must be used. In addition, the -Gsw compiler option, which compiles the application as a Windows executable program, must be included.

- The appropriate Zinc **.LIB** files, as shown in the table above, should be linked in.

Here is a sample "generic" makefile for DOS and Windows 3.X:

```
# GENERIC makefile

#    nmake -fmicrosft.mak dos          (makes the DOS generic program)
#    nmake -fmicrosft.mak windows      (makes the Windows generic program)

# Be sure to set the LIB and INCLUDE environment variables for Zinc, e.g.:
#    set INCLUDE=.;C:\ZINC\INCLUDE;C:\C700\INCLUDE
#    set LIB=.;C:\ZINC\LIB\MSCPP700;C:\C700\LIB

## Compiler and linker: (Add -Zi to CPP_OPTS and /CO to LINK_OPTS for
debug.)

# ----- DOS compiler options ---------------------------------
DOS_CPP_OPTS=-c -AL -BATCH -Gs
DOS_LINK_OPTS=/NOD /NOI /BATCH /STACK:5120 /SEGMENTS:256
DOS_OBJS=
# --- Use the next line for UI_GRAPHICS_DISPLAY ---
DOS_LIBS=dos_zil dos_gfx ms_lgfx llibce graphics oldnames
# --- Use the next line for UI_MSC_DISPLAY ---
#DOS_LIBS=dos_zil dos_msc llibce graphics oldnames

# ----- Windows compiler options -----------------------------
WIN_CPP_OPTS=-c -AL -BATCH -Gsw -DWINVER=0x0300
WIN_LINK_OPTS=/NOD /NOI /BATCH /STACK:5120 /SEGMENTS:256
WIN_OBJS=
WIN_LIBS=win_zil libw llibcew oldnames

CPP=cl
LINK=link

.cpp.obj:
    $(CPP) $(DOS_CPP_OPTS) $<

.cpp.obw:
    $(CPP) $(WIN_CPP_OPTS) -Fo$*.obw $<

# ----- DOS --------------------------------------------------
dos: generic.exe

generic.exe: generic.obj
    $(LINK) $(DOS_LINK_OPTS) @<<zil.rsp
$(DOS_OBJS)+generic.obj
$*,NUL,$(DOS_LIBS),NUL
<<

# ----- Windows ----------------------------------------------
windows: wgeneric.exe
```

```
wgeneric.exe: generic.obw
    $(LINK) $(WIN_LINK_OPTS) @<<zil.rsp
$(WIN_OBJS)+generic.obw
$*,NUL,$(WIN_LIBS),wgeneric.def
<<
    rc -30 -k wgeneric.rc $*.exe
```

Here is a sample "generic" makefile for Windows NT:

```
# ----- General Definitions----------------------------------
!include <ntwin32.mak>
VERSION=winnt

# ----- Windows compiler options----------------------------
WIN_CPP=$(cc)
WIN_LINK=$(link)
WIN_LIBRARIAN=lib

WIN_CPP_OPTS=$(cflags) $(cvars)
#WIN_CPP_OPTS=$(cflags) $(cvars) /Zi
#WIN_LINK_OPTS=$(conflags)
WIN_LINK_OPTS=$(guiflags)
#WIN_LINK_OPTS=$(guiflags) /DEBUG:MAPPED,FULL /DEBUGTYPE:CV
#WIN_LIB_OPTS=/machine:i386 /subsystem:CONSOLE
WIN_LIB_OPTS=/machine:i386 /subsystem:WINDOWS

WIN_OBJS=
#WIN_LIBS=$(conlibs) wnt_zil.lib
WIN_LIBS=$(guilibs) wnt_zil.lib

.SUFFIXES : .cpp

.cpp.obj:
    $(WIN_CPP) $(WIN_CPP_OPTS) $<

wgeneric.exe: generic.obj
    $(WIN_LINK) $(WIN_LINK_OPTS) -out:wgeneric.exe $(WIN_OBJS) generic.obj
$(WIN_LIBS)
```

Zortech

The following Zinc libraries are for use with Zortech:

- **DOS_ZIL.LIB** (basic DOS library)
- **D32_ZIL.LIB** (32-bit DOS library)
- **DOS_GFX.LIB** (DOS UI_GRAPHICS_DISPLAY)
- **D32_GFX.LIB** (DOS UI_GRAPHICS_DISPLAY for 32-bit DOS extender)
- **ZT_LGFX.LIB** (Zortech-specific GFX graphics library)

- **ZT_32GFX.LIB** (Zortech-specific GFX graphics library for 32-bit DOS extender)
- **DOS_FG.LIB** (DOS UI_FG_DISPLAY)
- **WIN_ZIL.LIB** (MS Windows library)
- **OS2_ZIL.LIB** (IBM OS/2 library)

The following table lists the types of applications that can be built using Zortech and Zinc and the Zinc libraries that are required for each:

Type of application	Required libraries	Comments
DOS with FG (also supports text mode)	DOS_ZIL.LIB DOS_FG.LIB	Must also link in FG.LIB
DOS with GFX (also supports text mode)	DOS_ZIL.LIB DOS_GFX.LIB ZT_LGFX.LIB	Must not link in FG.LIB
DOS 32-bit with GFX (also supports text mode)	D32_ZIL.LIB D32_GFX.LIB ZT_32GFX.LIB	Must not link in FG.LIB
Windows	WIN_ZIL.LIB	
OS/2	OS2_ZIL.LIB	

Before using the makefile the following may need to be changed:

- Change the environment variable for your include files' path by entering **`set include=`**, followed by the paths for your Zortech include files and your Zinc include files. For example, if your include files are all on Drive C, enter:

```
set include=.;C:\ZINC\INCLUDE;C:\ZORTECH\INCLUDE
```

- Change the environment variable for the libraries by entering **`set lib=`**, followed by the paths for your Zortech libraries and your Zinc libraries. For example, if your libraries are all on Drive C, enter:

```
set lib=.;C:\ZINC\LIB\ZTCPP300;C:\ZORTECH\LIB
```

NOTE: Probably the easiest place to change the environment variables is in your **AUTOEXEC.BAT** file.

For information on building applications with DOS-extenders, please see the **READ.ME** file.

Workbench (ZWB)

To compile multi-module DOS, Windows 3.X or OS/2 2.X applications in the ZWB, the following options must be selected within the ZWB:

- First create a makefile according to the description in the section below.

- Select Compile|Make Options.
  Enter `-fzortech.mak <platform>` where `<platform>` is DOS, WINDOWS or OS2, as desired.

- Select Compile|Make

To compile single-module DOS, Windows 3.X or OS/2 2.X applications in the ZWB, the following options must be selected within the ZWB:

- Select Compile|Compile Options|Code Generation.
  Choose the Large Memory Model.
  In the Command Line field, enter the appropriate libraries, as shown in the table above.
  Choose OS Support of DOS, Windows or OS/2, as desired.
  Choose More and set Structure Alignment to Byte.

Makefiles

Each tutorial program has a sample makefile, **ZORTECH.MAK**, that can be used as a template for other programs. The options listed next to CPP_OPTS in the makefile are recommended by Zinc. The makefile can be run by typing the following:

```
make -fzortech.mak dos          (for DOS)
make -fzortech.mak windows      (for Windows)
make -fzortech.mak os2          (for OS/2)
```

- In order to compile Microsoft Windows applications, you must be using Microsoft Windows Version 3.X. All of the options specified in the Zortech sample makefile

must be used. In addition, the -W2 compiler option, which compiles the application as a Windows executable program, must be included.

- The appropriate Zinc **.LIB** files, as shown in the table above, should be linked in.

Here is a sample "generic" makefile:

```
# GENERIC makefile

#    make -fzortech.mak dos              (makes the DOS generic program)
#    make -fzortech.mak windows          (makes the Windows generic program)
#    make -fzortech.mak os2              (makes the OS/2 generic program)

# Be sure to set the LIB and INCLUDE environment variables for Zinc, e.g.:
#    set INCLUDE=.;C:\ZINC\INCLUDE;C:\ZTC\INCLUDE
#    set LIB=.;C:\ZINC\LIB\ZTCPP300;C:\ZTC\LIB

## Compiler and linker: (Add -g to CPP_OPTS and /CO to LINK_OPTS for debug.)

# ----- DOS compiler options ---------------------------------
DOS_CPP=ztc
DOS_LINK=blinkx
DOS_LIBRARIAN=zorlibx

DOS_CPP_OPTS=-c -a1 -bx -ml
DOS_LINK_OPTS=/NOI
DOS_OBJS=
# --- Use the next line for UI_GRAPHICS_DISPLAY ---
DOS_LIBS=dos_zil dos_gfx zt_lgfx
# --- Use the next line for UI_FG_DISPLAY ---
#DOS_LIBS=dos_zil dos_fg fg

# ----- Windows compiler options-----------------------------------
WIN_CPP=ztc
WIN_LINK=blinkx
WIN_LIBRARIAN=zorlibx

WIN_CPP_OPTS=-c -a1 -bx -ml -W2
WIN_LINK_OPTS=/NOI
WIN_OBJS=
WIN_LIBS=win_zil

# ----- OS/2 compiler options--------------------------------------
OS2_CPP=ztc
OS2_LINK=blinkos2
OS2_LIBRARIAN=zorlib

OS2_CPP_OPTS=-c -mf
OS2_LINK_OPTS=/BASE:0x10000 /PM:PM
OS2_OBJS=
OS2_LIBS=os2_zil

.c.obj:
    $(DOS_CPP) $(DOS_CPP_OPTS) $<

.cpp.obj:
    $(DOS_CPP) $(DOS_CPP_OPTS) $<

.c.obw:
    $(WIN_CPP) $(WIN_CPP_OPTS) -o$*.obw $<

.cpp.obw:
    $(WIN_CPP) $(WIN_CPP_OPTS) -o$*.obw $<
```

```
.c.obo:
    $(OS2_CPP) $(OS2_CPP_OPTS) -o$*.obo $<

.cpp.obo:
    $(OS2_CPP) $(OS2_CPP_OPTS) -o$*.obo $<

# ----- DOS-------------------------------------------------------
dos: generic.exe

generic.exe: generic.obj
    $(DOS_LINK) $(DOS_LINK_OPTS) $(DOS_OBJS)+generic.obj,$*, ,
        $(DOS_LIBS),NUL

# ----- Windows---------------------------------------------------
windows: wgeneric.exe

wgeneric.exe: generic.obw
    $(WIN_LINK) $(WIN_LINK_OPTS) $(WIN_OBJS)+generic.obw,$*, ,
        $(WIN_LIBS),wgeneric.def
    rc -k wgeneric.rc $*.exe

# ----- OS/2------------------------------------------------------
os2: ogeneric.exe

ogeneric.exe: generic.obo
    $(OS2_LINK) $(OS2_LINK_OPTS) $(OS2_OBJS)+generic.obo,$*, ,
        $(OS2_LIBS),ogeneric.def
    rc ogeneric.rc $*.exe
```

Motif

Notes on using Motif

The Motif version of Zinc Application Framework was developed for Motif version 1.1 and the X Window System version 11R4. Whenever possible, the library is made compatible with Motif version 1.0 and X11R3, so it should not be difficult to port to systems that do not yet support version 1.1. For example, rather than using **XtVaSetValues( )**, the older combination of **XtSetArg( )** and **XtSetValues( )** is often used. Zinc Application Framework has been compiled and run on systems with Motif 1.2 and X11R5 with only very minor changes.

Since there are so many different Motif systems in use, Zinc cannot provide makefiles and instructions for every system. The Motif version of Zinc Application Framework is provided only in source code form and not as a previously compiled library. As a result, each site will need to compile its own library. The next section should make it easier to 'tailor' the makefiles to your specific system.

NOTE: Some implementations of X-Windows use "class" and "new" for variable names and so are not compatible with C++ (and subsequently are not compatible with Zinc Application Framework). A different version of X-Windows should be obtained in these situations.

Makefiles

Each tutorial program has a sample makefile, **Makefile**, that can be used as a template for other programs. The makefile can be run by typing the following:

```
make
```

Before using the makefile the following may need to be changed:

- Change the compiler's include file search path. By default, the Zinc source for Motif assumes that the Motif include files are in a directory called **Xm**, and that the Xt and X include files are in a directory called **X11**. The parent directory of Xm and X11 must be in your compiler's include search path. If, for example, the include files are located in **/usr/include/Motif1.1/Xm** and the Xt and X include files are located in **/usr/include/X11R4/X11**, the following should be added to the makefile:

  ```
  -I/usr/include/Motif1.1  -I/usr/include/X11R4
  ```

- Change the compiler's library file search path. Zinc applications need to be linked with the Motif version of Zinc Application Framework, **libXm.a**, the Xt library, **libXt.a**, and the X Window library **libX11.a**. To inform the linker of the location of these files use a command similar to the following:

  ```
  -L/usr/lib/Motif1.1  -lXm  -L/usr/lib/X11R4  -lXt  -lX11
  ```

NOTE: By default, the Zinc installation program will install the Zinc include files to **/usr/include** and the library files to **/usr/lib**.

Here is a sample "generic" makefile:

```
PROGS = generic

CXXFLAGS= -g -I../../include -I/usr/include/X11R4 -I/usr/include/Motif1.1
LFLAGS=
LIBS= -L../../lib -l_mtf_zil -L/usr/lib/Motif1.1 -lXm -L/usr/lib/X11R4 -lXt
-lX11 -lcodelibs -lc /usr/lib/end.o

motif: generic

generic: generic.o
    $(CXX) $(LFLAGS) -o $@ $? $(LIBS)

clean:
    -rm -f generic *.o
```

Porting to another system

When Zinc is ported to additional Motif systems, various differences may occur. The Zinc include file, **UI_ENV.HPP**, specifies environment specific macros and declarations. There are three main defines that are of interest. The following defines should be verified:

- either ZIL_LITTLEENDIAN or ZIL_BIGENDIAN must be defined
- either ZIL_BITS16 or ZIL_BITS32 must be defined
- ZIL_LOAD_MOTIF must be defined. (**NOTE:** ZIL_LOAD_MOTIF is a macro that causes ZIL_MOTIF and ZIL_X11 to be defined according to values in the Zinc Application Framework and X include files.)

APPENDIX B – EXAMPLE PROGRAMS

Zinc Software continually improves and updates example programs that provide additional help for programmers on particular library topics. The following list describes the example programs that are available in the Zinc Application Framework examples directory. For additional updates and examples, keep in contact with our Bulletin Board Service and the Zinc Software technical support group.

All example programs are located in individual sub-directories under **\ZINC\EXAMPLE**. An explanation of the files contained in each directory is given below:

***.CPP**—These files contain the source code for the example program.

***.HPP**—These files (if any) contain class definitions and constants used by the example program.

***.TXT**—These files (if any) can be used with the **GENHELP.EXE** utility to add help contexts to a new or existing **.DAT** file.

***.DAT**—These files (if any) contain any persistent window objects (created with Zinc Designer) used in the example.

.DEF**, ***.RC**—These files (if any) are the environment specific definition and resource files required when compiling for Windows or OS/2. (**NOTE:** The **W*. files are for Windows and the **O*.*** files are for OS/2.)

***.MAK**—These files are the compiler-dependent makefiles associated with the example program. (See "Chapter 1—Initializing the Library" for information on compiling for each Zinc-supported platform.)

ANALOG

This program displays a constantly updating, sizeable analog clock to the graphics display. This is accomplished by implementing a multiple inheritance class derived from UI_DEVICE and UIW_WINDOW.

Concepts demonstrated: multiple inheritance, derived devices and graphics.

BIO

This program uses a class derived from UI_WINDOW_OBJECT to display sine wave representations of a person's biorhythm in the lower portion of a window while allowing the user to enter date information in the upper portion of the window. The window is sizeable, and the sine wave graphics are dynamically sized within the window by use of the WOF_NON_FIELD_REGION flag.

Concepts demonstrated: non-field region objects and graphics.

CALC

This program uses a CALCULATOR class derived from UIW_WINDOW to display a calculator, which consists of a UIW_REAL class object and several UIW_BUTTON class objects inside of a window. This program demonstrates how to attach user functions to UIW_BUTTON class objects and how to call a non-static class member function from a static user function.

Concepts demonstrated: UIW_REAL usage, calling non-static member functions.

CALENDAR

This program creates a sizeable calendar for which the spacing of the days and weeks is dynamically changed according to the size of the calendar. This is accomplished by deriving classes from UI_WINDOW_OBJECT and UIW_WINDOW.

Concepts demonstrated: dynamically sizing window objects, UI_DATE usage, using a custom event map table.

CHECKBOX

This program demonstrates the use of checkboxes and radio buttons (UIW_BUTTON class).

Concepts demonstrated: checkbox usage, radio button usage, user functions and setting default selected status.

CLOCK

This program displays a constantly updating digital clock to the graphics or text display. This is accomplished by implementing a multiple inheritance class derived from UI_-DEVICE and UIW_WINDOW.

Concepts demonstrated: derived UI_DEVICE class and multiple inheritance.

COMBOBOX

This program demonstrates the use of the UIW_COMBO_BOX class.

Concepts demonstrated: combobox usage, user functions and setting the default selected status.

DIRECT

This program displays filenames as UIW_STRING class objects attached to a UIW_VT_LIST class object. The program allows the user to change directories by selecting UIW_STRING class objects which are attached to a UIW_VT_LIST class object. If the user double-clicks on a file name, the UIW_STRING class object will call a user function that will display the file name, file size and file date in a window.

Concepts demonstrated: file access, directory manipulation, UIW_VT_LIST usage and user functions.

DRAW

This program creates an icon builder/editor similar to the editor in Zinc Designer. A new object, UIW_BITMAP, is defined.

Concepts demonstrated: graphics, UIW_ICON usage and UIW_BITMAP class created.

DISPLAY

This program demonstrates the functionality of the UI_BGI_DISPLAY, UI_-FG_DISPLAY, UI_MSC_DISPLAY, UI_TEXT_DISPLAY and UI_MSWINDOWS_-DISPLAY classes. The program uses the **RegionDefine( )**, **Rectangle( )**, **Text( )** and **TextWidth( )** member functions to draw graphics information to the screen.

Concepts demonstrated: graphics and using various graphics modes.

ERROR

This program demonstrates how the UI_ERROR_SYSTEM can be called from a user function.

Concepts demonstrated: UI_ERROR_SYSTEM features and usage.

FILEEDIT

This program implements a file text editor complete with directory and file manipulation functionality. This program uses classes derived from the UIW_WINDOW class. It also uses the UI_HELP_SYSTEM class.

Concepts demonstrated: file access, directory manipulation, UIW_VT_LIST usage, user functions and UIW_TEXT usage.

FREESTOR

This program implements a free store exception handler. When the **new( )** operator fails to allocate memory, Freestor can be used to allow the user application to recover gracefully. (**NOTE:** This example program is for DOS only.)

Concepts demonstrated: out of memory detection.

GRAPH

This program displays line graphs, bar graphs and pie graphs inside of several overlapping windows.

Concepts demonstrated: derived graphic window objects and graphics in various environments.

MESSAGES

This program displays two buttons in a window. If either button is pressed, a menu window appears, displaying several options. If any of the options in the menu window are selected, the menu window will disappear, and the selected option's text will appear

on the button that was originally selected. This is accomplished by using a class derived from UIW_BUTTON which understands a programmer-defined event type. A UI_EVENT class object of this type then uses its 'data' member to point to the new character array.

Concepts demonstrated: sending messages to specific objects.

PERIODIC

This program creates a periodic table of elements.

Concepts demonstrated: Zinc Designer integration and user functions.

PHONEBK

This program implements a phone number storage/retrieval system. It uses the UI_STORAGE and UI_STORAGE_OBJECT classes to save the phone number entries in the Zinc data file.

Concepts demonstrated: using the Zinc data file and user functions.

PUZZLE

This program creates a "15's" puzzle using a class derived from UIW_WINDOW and a group of UIW_BUTTON class objects.

Concepts demonstrated: sizing of button objects and changing of button appearance.

SPY

This program displays the textual representation of event types in a window as the events occur in a typical Zinc application. This is accomplished by deriving an Event Manager class from the UI_EVENT_MANAGER class. This class prints the first message on the queue each time eventManager->Get( ) is called. The textual representation of each event it output to a TTY_WINDOW.

Concepts demonstrated: derived UIW_EVENT_MANAGER, "scrolling" TTY_-WINDOW and event translation.

VALIDATE

This program attaches a validate function to several UIW_BIGNUM class objects in order to display the sum of these objects in an additional non-selectable UIW_BIGNUM class object. This program demonstrates how to call a non-static class member function using a static validate function.

Concepts demonstrated: calling non-static functions and UIW_BIGNUM usage.

APPENDIX C – ZINC CODING STANDARDS

Zinc Software has an internal document that specifies standards for all code written for internal, as well as external, distribution. The purpose of these standards is to improve the readability, organization and maintenance of source code and header files. This document is printed in this appendix so that you can understand the coding standards we use when writing the example programs, tutorial programs and source code modules you receive when you purchase this product.

Naming

Classes and structures

Class names should be self-explanatory and should be in upper-case lettering, with underscores used to separate words. Some example class and structure names are shown below.

```
struct UI_EVENT
struct UI_PALETTE_MAP
class UI_ELEMENT
class UI_EVENT_MANAGER : public UI_LIST
class UIW_BUTTON : UI_WINDOW_OBJECT
```

In addition, the following prefixes are used in conjunction with Zinc Application Framework:

UI_ is used to denote a general user interface class object or structure.

UID_ is used to denote a device class object or structure.

UIW_ is used to denote a window interface class object or structure.

Functions

Functions should be self explanatory and should be in name-case format (i.e., first letter upper-case lettering, all remaining character in lower-case lettering) with no underscores used to separate words. In addition, the function name should describe the operation that is performed by the function.

Some example class and regular function names are shown below:

```
UI_ELEMENT *Previous(void);

EVENT_TYPE Event(const UI_EVENT &event);

static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *directory,
    UI_STORAGE_OBJECT *object);
```

Variables

Variable names should be self-explanatory and should be in lower-case lettering for the first word, then be name-case for each word thereafter. Global variables should be preceded by an underscore. Some example variable names are shown below.

```
extern UI_STORAGE *_storage;

int UIW_BORDER::width = 4;
static UI_EVENT_MAP *eventMapTable;

UI_PALETTE_MAP *paletteMapTable;
```

Each variable should be declared on a separate line when it is needed by the function. When declaring a list of variables, the following order should be followed:

1—External variables

2—Static variables

3—Variables with complex structures

4—All other variables according to need within the application

In addition, only one space (not tabs) should exist between the type and the variable. Comments should be aligned evenly after the variable list.

Constants

Constant variables should be self-explanatory and should be in upper-case lettering, with an underscore separating the words.

Some example constant names are shown below:

```
const int TRUE  = 1;
const int FALSE = 0;
```

```
const WOF_NO_FLAGS WOF_NO_FLAGS          = 0x0000;
const WOF_NO_FLAGS WOF_JUSTIFY_CENTER    = 0x0001;
```

In addition to the information described above:

- Constants should be placed before the definition of the class for which they apply, or at the beginning of the module.
- If several related constants are defined, the definitions should be grouped together with a preceding comment.
- Constant values should be tab aligned to the right.
- Comments for each line, if needed, should be aligned to the right of the value.

Organization

Class scopes

The declaration of a class in an include file should list public members first, protected members next and private members last. Each major section should list static member variables first, member variables next and member functions last, listed in alphabetical order. (The constructor and destructor should be listed first.) In addition, each scope section should contain a short comment telling where its members are documented. The following example shows a class containing the three scope sections:

```
class EXPORT UI_TIME : public UI_INTERNATIONAL
{
public:
    static char *amPtr;
    static char *pmPtr;

    UI_TIME(void);
    virtual ~UI_TIME(void);
    .
    .
    .

    void Export(char *string, TMF_FLAGS tmFlags);
    .
    .
    .

    long operator=(long hundredths);

private:
        long value;
};
```

Files

Source code modules that contain class member functions should contain the copyright notice, then any include files, followed by static member variables, and finally, member functions, described in alphabetical order. An example of **BORDER.CPP** file layout is shown below:

```
//  Zinc Application Framework - BORDER.CPP
//  COPYRIGHT (C) 1990-1993.  All Rights Reserved.
//  Zinc Software Incorporated.  Pleasant Grove, Utah  USA

#include "ui_win.hpp"
#include <string.h>

int UIW_BORDER::width = 4;

UIW_BORDER::UIW_BORDER(void) :
    UI_WINDOW_OBJECT(0, 0, 0, 0, WOF_NON_FIELD_REGION, WOAF_NON_CURRENT)
{
    .
    .
    .

}

UIW_BORDER::~UIW_BORDER(void)
{
    .
    .
    .

}

EVENT_TYPE UIW_BORDER::Event(const UI_EVENT &event)
{
    .
    .
    .

}
```

Comments

Files

Each source file (**.CPP** or **.HPP**) should contain a three line comment that contains the library or program name, the name of the file and copyright information. A sample header is shown below:

```
//  Zinc Application Framework - BUTTON.CPP
//  COPYRIGHT (C) 1990-1993.  All Rights Reserved.
//  Zinc Software Incorporated.  Pleasant Grove, Utah  USA
```

The copyright information should be copied as shown above. The copyright year should include the original year when the product was created and all subsequent years when major revisions were made.

Functions

Each routine may be preceded by a short description giving the routine's purpose and any related algorithms. If the routine name intuitively describes the routine, no comment is needed. The example below shows the use of a function comment:

```
// This member function displays the biorhythm information in the window.
// As the size of the window object changes (by changing the parent window)
// the size of the biorhythm chart also changes.  A horizontal change
// results in a change in the number of days displayed.  A vertical change
// results in a dynamic change in the height of the biorhythm curve.
void BIORHYTHM::UpdateBiorhythm()
{
    .
    .
    .
}
```

Variables

Function arguments and local variables should only have descriptive comments if their names are not descriptive. These comments should be lined up on a right tab region. In addition, all comments should start with a capital letter and be followed by a period. An example of three variable declarations is shown below.

```
EVENT_TYPE ccode;    // The control code for an event.

long fileOffset;

int cardFile;        // File handle for the disk file.
```

Blocks

Block comments are used to describe a group of related code. Most block comments should be one line, if possible, and reside immediately above the block being commented. If more than a one line comment is needed, the extra lines should each begin with the double slash.

Block comments should be indented to match the indentation of the line of code following it. A single blank line should precede the comment and the block of code should follow immediately after. Small blocks of code that do a specific job should be commented but

not individual lines (unless the line is complex or not intuitive). Some example block comments are shown below.

```
// Destroy all of the items within the list.
Destroy();

// When the user selects a button from the current window, ccode
// is checked to see what type of event was received.
switch (ccode)
{
    .
    .
    .

}
```

Indentation

Classes and structures

Structures and classes should have all members listed on individual lines and should be indented with one tab from the left margin. Several sample indentations are shown below:

```
class EXPORT UI_DEVICE : public UI_ELEMENT
{
    friend class EXPORT UI_EVENT_MANAGER;
public:
    static ALT_STATE altState;
    static UI_DISPLAY *display;
    static UI_EVENT_MANAGER *eventManager;

    int installed;
    DEVICE_TYPE type;
    DEVICE_STATE state;

    virtual ~UI_DEVICE(void);
    virtual EVENT_TYPE Event(const UI_EVENT &event) = 0;

    // List members.
    UI_DEVICE *Next(void);
    UI_DEVICE *Previous(void);

protected:
    UI_DEVICE(DEVICE_TYPE _type, DEVICE_STATE _state);
    static int CompareDevices(void *device1, void *device2);
    virtual void Poll(void) = 0;
};
```

Functions

The main body of routines should have braces below the function declaration. All function code should be indented one tab. An example of this indentation is shown below:

```
void UIW_BUTTON::DataSet(const char *string)
{
    // Reset the button's string information.
    .
    .
    .

}
```

Function calls

Parameters in a function call should be listed with each argument, followed by a comma and one space. If a routine call cannot fit on one line on the screen, it should be broken with the next half of the call indented one tab farther over. It should be split after a comma or logic symbol if possible. Several examples of this calling convention are shown below:

```
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);
    .
    .
    .

}

UIW_WINDOW *window = UIW_WINDOW::Generic(2, 2, 40, 6, "Hello World Window",
    NULL, WOF_NO_FLAGS, WOAF_NO_FLAGS, HELP_HELLO_WORLD);

// Add the window objects to the window.
*window
    + new UIW_TEXT(0, 0, 0, 0, "Hello, World!", 256,
        WNF_NO_FLAGS, WOF_NON_FIELD_REGION);
```

Case statements

The reserved word **case** should be aligned with the **switch** statement, but all code information should be indented an additional tab. Each additional level of logic should be indented one tab. The colon should immediately follow each case and the statement(s) should start on a new line. The break should also be on a separate line. An example of this organization is shown below:

```
EVENT_TYPE UIW_PROMPT::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    EVENT_TYPE ccode = event.type;
    switch (ccode)
    {
```

```
        case S_CREATE:
        case S_SIZE:
            .
            .
            .
            break;

        case S_CURRENT:
        case S_NON_CURRENT:
        case S_DISPLAY_ACTIVE:
        case S_DISPLAY_INACTIVE:
            if (UI_WINDOW_OBJECT::NeedsUpdate(event, ccode))
                UI_WINDOW_OBJECT::Text(prompt, 0, ccode, lastPalette);
            break;

        default:
            ccode = UI_WINDOW_OBJECT::Event(event);
            break;
        }

        // Return the control code.
        return (ccode);
    }
```

If and for statements

Statements following an **if** or **for** should be indented one tab, and simple conditionals should use the in-line **?** operator. An example of these statements is shown below:

```
left = (left < 1) ? 1 : right;

if (event->type == E_KEY &&
    (event->rawCode == ESCAPE || event->rawCode == BACKSPACE ||
     event->rawCode == ENTER))
{
    offset = length;
    length = 0;
}
for (number = 0; number < noOfCalls; number++)
    ;   // Do nothing.
```

The braces enclosing the block should be aligned with the "if" or "for." If no statement exists for the "for" loop, the semicolon should be placed on the next line.

Multi-line equates

Each multi-line equate should be listed on a separate line as shown below:

```
windowID[0] =
    windowID[1] =
    windowID[2] =
    windowID[3] =
    windowID[4] = ID_WINDOW_OBJECT;
```

Each of the successive equates is indented one tab more than the first.

APPENDIX D – QUESTIONS AND ANSWERS

This appendix addresses some of the most frequent questions addressed by the technical support group. Each question is addressed in the form of a question and a short answer, with the concept being identified in the side title.

Ahh!...getting help

Question: What technical support services does Zinc offer?

Answer: Zinc currently offers the following technical support services to registered users at no charge:

United States

- Telephone support:
  (801) 785-8998, 8:00 a.m. to 5:00 p.m. Mountain Standard Time

- BBS:
  (801) 785-8997, 9600 V.32 *bis* (8,N,1), 24 hours
  (801) 785-8995, 9600 HST dual standard (8,N,1), 24 hours

- FAX:
  (801) 785-8996, allow 2-5 business days for a response. If you need to send more than one page of code, please use the BBS.

Europe

- Telephone support:
  +44 (0)81 855 9918, 9:00 a.m. to 5:00 p.m. London Time

- BBS:
  +44 (0)81 317 2310, 9600 HST dual standard (8,N,1), 24 hours

- FAX:
  +44 (0)81 316 7778, allow 2-5 business days for a response. If you need to send more than one page of code, please use the BBS.

Bitmaps/Icons not displaying

Question: When a window with bitmaps or icons is loaded from a **.DAT** file, the bitmap images are not displayed. What is wrong?

Answer: Windows can be loaded from a **.DAT** file using code similar to the following:

```
UIW_WINDOW *window = new
    UIW_WINDOW("file_name.dat~window_name");
```

When the library loads bitmaps, it does so from the UI_STORAGE object specified by *UI_WINDOW_OBJECT::defaultStorage*. If *UI_WINDOW_-OBJECT::defaultStorage* is NULL (i.e., it has not been initialized), the bitmap images cannot be loaded. *UI_WINDOW_OBJECT::defaultStorage* can be initialized according to the following code:

```
UI_WINDOW_OBJECT::defaultStorage = new
    UI_STORAGE("file_name.dat");
.
.
.
UIW_WINDOW *window = new UIW_WINDOW(2, 2, 50, 15);
// Add minimize icon and bitmapped button.
*window
    + new UIW_ICON(0, 0, "myIcon", "Window",
        ICF_MINIMIZE_OBJECT)
    + new UIW_BUTTON(1, 1, 15, "Press here", BTF_NO_TOGGLE |
        BTF_AUTO_SIZE, WOF_NO_FLAGS, PressFunction, 0,
        "PRESS_BITMAP");
```

Changing object flags

Question: How can the flags of an object be changed after the object has been constructed?

Answer: The |= operator can be used to set flags, the &= operator to clear flags and the ^= operator can be used to toggle flags. The following example code shows how this is accomplished:

```
// Set the non-current flag of an object.
object->woAdvancedFlags |= WOAF_NON_CURRENT;

// Clear the auto-clear flag of an object.
object->woFlags &= ~WOF_AUTO_CLEAR;

// Toggle the selected status of an object.
object->woStatus ^= WOS_SELECTED;
```

Changing the map tables

Question: How can changes be made to the global event map table and/or to the global palette map table at compile time?

Answer: Edit the files **G_EVENT.CPP** and/or **G_PNORM.CPP** and include them in the project before the Zinc library file, and they will override the default tables included in the library. (**NOTE:** The palette map table can only be changed for DOS programs.)

Checking for selected objects

Question: How can the program determine if an object is in the selected state? (e.g. check box on, radio button on, etc.)

Answer: Test the *woStatus* of the item for the WOS_SELECTED flag. The following code shows how this can be done:

```
if(FlagSet(item->woStatus, WOS_SELECTED))
{
    .
    .
    .

}
```

Closing the current window

Question: How can the current window be closed in a user function?

Answer: In order to close the current window in a user function, the following code can be used:

```
event.type = S_CLOSE;
object->eventManager->Put(event);
```

Do not use the following code in a user function:

```
event.type = S_CLOSE;
object->windowManager->Event(event);
```

or

```
*object->windowManager
    - currentWindow;
delete currentWindow;
```

If the window is closed before leaving the user function, the window could be deleted. If the window is deleted, the object calling the user function

will also be deleted. Then when the user function is exited, it returns to an address which has been freed. In other words, this can be compared to climbing out on a limb and attempting to cut out the section of the limb between you and the main trunk. The freed memory may be corrupted—the results are unpredictable.

Compiler warning

Question 1: I get a "result of * expression not used" warning when compiling in Motif. Why?

Answer: Some compilers for Motif will produce this warning when using the overloaded + and - operators. Typecasting the result of the operation as follows will eliminate the warning:

```
(void *)&(*eventManager
    + new UID_KEYBOARD
    + new UID_MOUSE
    + new UID_CURSOR);
```

Display/Mouse remaining active

Question: Why might the display and the mouse remain active after exiting the program, even if the program deleted them?

Answer: This can occur as a result of the 'Word alignment' option being set improperly when the program is compiled.

In the Borland IDE: Options|Compiler|Code Generation, Word alignment must be off, Unsigned characters must be off and Treat enums as ints must be on. Otherwise, calls to the library will be done incorrectly.

In the Zortech ZWB: Compile|Compile Options|Code Generation, "Align" must be set to Byte and the CHAR == UCHAR option must be off.

Finding an object in a window

Question: Given a pointer to a window, how can a pointer to an object in that window be found?

Answer: The following code will get a pointer to the n^{th} object in a window. Similar code will get the n^{th} object in a list, menu or any other object derived from window.

```
UI_WINDOW_OBJECT *object =
    (UI_WINDOW_OBJECT *)window->UI_LIST::Get(n);
```

If the object has a string ID it can be found by using the following code:

```
UI_WINDOW_OBJECT *object = (UI_WINDOW_OBJECT *)
    window->Information(GET_STRINGID_OBJECT, stringID);
```

Finding the current window

Question 1: How can you determine which window is the current window attached to the Window Manager?

Answer: The member function **windowManager->First( )** will return a pointer to the current window.

Question 2: Given a pointer to a window, how can the current object in that window be found?

Answer: The member function **window->Current( )** will return a pointer to the current field in the window. (**NOTE:** That field may have sub-fields.)

Finding the parent window

Question: Given a pointer to an object in a window, how can you find the parent window?

Answer: Given a pointer to an object, such as in a user function, the following loop will exit with a pointer to the parent window named *parentWindow*:

```
for (UI_WINDOW_OBJECT *parentWindow = object;
    parentWindow->parent; parentWindow = parentWindow->parent)
    ;
```

Fix-up overflow errors

Question: What causes fix-up overflow errors?

Answer: Fix-up overflow indicates that the **.OBJ** files are not linking properly with each other or the **.LIB** files. This can be caused by compiling **.OBJ** files in some model other than large and trying to link with Zinc Application Framework (which was compiled for large model). It can also be caused by compiling the .OBJ files with one version of the compiler and trying to link with Zinc Application Framework that compiled with another. This is especially a problem when using a older version of a compiler. (Contact Customer Support about support for previous versions of compilers.)

International language

Question: Does Zinc Application Framework provide any international language support?

Answer: Zinc uses the country information provided by the operating system to determine the appropriate format and edit controls for dates, times and numbers. You can also build language specific data files using Zinc Designer and pass them to your application depending on the country information.

Making a window current

Question: How can a different window be made to be the current window?

Answer: Add the window to be current to the Window Manager. The following code shows how this can be done:

```
*windowManager + window1;
```

"Out-of-memory" compiler errors

Question: What causes 'Out of Memory' errors when compiling Zinc Application Framework programs?

Answer: Two things can cause this error. First, the source file could be too large for the compiler to handle. If so, the source file needs to be broken into smaller modules. Also, stringing too many add operations can cause the compiler to run out of memory during the compile. The example below shows how this can be accomplished:

```
*window + object1 + object2 + .... + object20;
```

could be written as:

```
*window + object1 + object2 + object3 + object4;
*window + object5 + object6 + object7 + object8;
*window + object9 + object10 + object11 + object12;
*window + object13 + object14 + object15 + object16;
*window + object17 + object18 + object19 + object20;
```

Preventing the modification of objects

Question: How can a window object be changed to be non-selectable in order to prevent users from being able to change it?

Answer: The WOF_VIEW_ONLY or the WOF_NON_SELECTABLE flags can be set by using one of the following lines of code:

```
object->woflags |= WOF_VIEW_ONLY;
object->woflags |= WOF_NON_SELECTABLE;
```

The same flags could be turned off again with the following code:

```
object->woflags &= ~WOF_VIEW_ONLY;
object->woflags &= ~WOF_NON_SELECTABLE;
```

Putting a single object in multiple windows

Question: Why can't a single instance of an object be added to two different windows?

Answer: Each object derived from UI_ELEMENT has pointers included in the class so that it can be placed in a UI_LIST. Because there is only one copy of these pointers, it can only be placed in one list. If you try to put an object into a second list, without subtracting it from the first list, the pointers are overwritten, and the first list becomes corrupt. This can result in the system hanging.

Re-displaying objects and windows

Question: How can a window object or an entire window be re-displayed?

Answer: To re-display a window object, it must be sent the S_REDISPLAY message.

```
object->Event(UI_EVENT(S_REDISPLAY));
```

To re-display an entire window, including all of the objects attached to the window, send the window an S_REDISPLAY message.

Royalties

Question: If I build an application with Zinc Application Framework, can I distribute it without having to pay Zinc any royalties?

Answer: You can distribute your application royalty-free as long as: 1) it bears a valid copyright notice, 2) it is not a library-type product, software development tool or operating system, and 3) it is not competitive with or used in lieu of Zinc. (Please refer to the Zinc Application Framework End User License Agreement.)

Undetected graphics mode

Question: Why might a DOS program not run in graphics mode, even when a graphics monitor is being used?

Answer: In order to run in graphic mode, Borland's **.BGI** (Borland Graphics Interface) files must be found if the program uses BGI graphics. When using the UI_BGI_DISPLAY, DOS will search in the directories stated in the APPEND, and Zinc will search for them in the directories stated in the PATH environment variable. Otherwise, the graphics driver will not be installed, and the program will run in text mode only. (See the **UTIL.DOC** file that came with the Borland compiler for details on linking BGI graphics drivers into your application.)

For applications using MSC graphics to run in graphics mode, Microsoft **.FON** files must be in the environment's PATH.

Using the Q_NO_BLOCK flag

Question: If the Q_NO_BLOCK flag is used when calling **eventManager->Get( ),** how can it be determined if a valid event was received, or if no events were in the event queue?

Answer: If the Q_NO_BLOCK flag is set, the return value from **eventManager-> Get( )** will be zero if an event was detected: otherwise, it will be a negative value (i.e., -1 or -2). The example below shows how you can check the status:

```
UI_EVENT event;
EVENT_TYPE ccode = eventManager->Get(event, Q_NORMAL |
    Q_NO_BLOCK);
if (ccode == 0) // An event was returned.
    .
    .
    .

else            // An event was not present.
    .
    .
    .
```

Using member functions as user functions

Question: How can a member function be used as a user function?

Answer: A member function must be declared as static in order to be used as a user function. (See the **VALIDATE** example for a demonstration of a static user function calling a non-static user function.)

Using .ICO and .BMP files

Question: How can previously created, **.ICO** or **.BMP** files be use with Zinc Application Framework?

Answer: Zinc Designer contains a utility to import **.ICO** and **.BMP** files. The icons and bitmaps can then be saved in a **.DAT** file. A utility program called **ICON2DAT.EXE** may also be used to convert **.ICO** and **.BMP** files.

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

GNU Free Documentation License
Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input

to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy

a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains

nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit

corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.