

Z

i

n

c

PROGRAMMER'S
TUTORIAL

INTERFACE
LIBRARYTM

VERSION 3.0

Message passing. . .

Zinc™ Interface Library™

Programmer's Tutorial

Version 3.0

Zinc Software Incorporated
Pleasant Grove, Utah

Copyright © 1990-1992 Zinc Software Incorporated 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".

TABLE OF CONTENTS

Introduction	1
---------------------------	---

SECTION I—HELLO WORLD!

Chapter 1—Initializing the Library	9
---	---

- Include files
- The screen display
- The event manager
- The window manager
- A simple window
- Program flow
- Cleanup
- Run-time features

Chapter 2—Help and Error Systems	21
---	----

- The help system
- The error system
- Multiple windows
- Program flow
- Cleanup
- Run-time features

Chapter 3—Using Zinc Designer	37
--	----

- Creating a file
- Creating a window
- Creating a window object
- Creating additional windows
- Saving the file
- Window access
- Run-time features

SECTION II—DICTIONARY

Chapter 4—Discovering the Object of C++	51
--	----

- Discovering objects
- Data hiding
- Constructors and destructors

Why use classes?
Deriving classes/Inheritance
Function overloading
Operator overloading
Local variables
Conclusion

Chapter 5—Event Flow 59

Program execution
Class definitions
Creating the window
Following events
Conclusion

Chapter 6—The Zinc Data File 69

Program execution
Class definitions
Creating the window
Using the data file
UI_STORAGE_OBJECT
UI_STORAGE
Conclusion

SECTION III—ZINC APPLICATION PROGRAM

Chapter 7—Getting the Right Design 81

Goals
High level design
Implementation
Accelerator keys
Structured programming

Chapter 8—Control Options 91

Control program flow

Chapter 9—Display Options 97

Display program flow

Chapter 10—Window Options 105

Window program flow

Chapter 11—Event Options	113
Event program flow	
Monitoring library events	

Chapter 12—Help Options	125
Help program flow	
General library help	

SECTION IV—DERIVED CLASSES

Chapter 13—Macro Device	133
Program execution	
Class definition	
Conceptual operation	
Class information	
Enhancements	

Chapter 14—Help Bar	145
Program execution	
Class definitions	
Using HELP_BAR	
Event function (DOS)	
Event function (Windows)	
Information function	
Enhancements	

Chapter 15—Virtual List	155
Program execution	
Class definitions	
Conceptual operation	
VIRTUAL_ELEMENT	
VIRTUAL_LIST	
Enhancements	

Chapter 16—Customized Displays	167
Conceptual design	
Class implementation	
Conclusion	

SECTION V—PERSISTENT OBJECTS

Chapter 17—Graphic Objects	181
C and C++	
Basic storage and retrieval	
Abstract storage and retrieval	
Chapter 18—Zinc Window Objects	195
Lists and class abstraction	
Implementation	
Conclusion	

SECTION VI — MISCELLANEOUS INFORMATION

Appendix A—Compiler Considerations	203
Borland	
Integrated Development Environment (IDE)	
Makefiles	
Zortech	
Workbench (ZWB)	
Makefiles	
Microsoft	
Programmer's Workbench (PWB)	
Makefiles	
Appendix B—Compiled BGI Files	209
Appendix C—Example Programs	213
ANALOG	
BIO	
CALC	
CALENDAR	
CHECKBOX	
CLOCK	
COMBOBOX	
DIRECT	
DRAW	
DISPLAY	
ERROR	
FILEEDIT	
GRAPH	

MESSAGES
NOTEPAD
PERIODIC
PHONEBK
PIANO
PUZZLE
SATELLIT
SERIAL
SPY
VALIDATE

Appendix D—Zinc Coding Standards 219

- Naming
 - Classes and structures
 - Functions
 - Variables
 - Constants
- Organization
 - Class scopes
 - Files
- Comments
 - Files
 - Functions
 - Variables
 - Blocks
- Indentation
 - Classes and structures
 - Functions
 - Function calls
 - Case statements
 - If and for statements
 - Multi-line equates

Appendix E—Questions and Answers 229

- Ahh!...getting help
- Borland BGI dependencies
- Borland IDE compiling
- Borland linker warnings
- Changing object flags
- Changing the map tables
- Checking the selected objects
- Closing the current window
- Display/Mouse remaining active

- DOS extenders
- Finding an object in a window
- Finding the current window
- Finding the parent window
- Fix-up overflow errors
- International language
- Making a window current
- Making a window object current
- Other platforms
- “Out-of-memory” errors
- Preventing the modification of objects
- Putting a single object in multiple windows
- Re-displaying objects and windows
- Royalties
- Undetected graphics mode
- Using the Q_NO_BLOCK flag
- Using member functions as user functions
- Using .ICO and .BMP files

Index 241

INTRODUCTION

The purpose of this manual is to help you get started using Zinc Interface Library 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, 1991, 444 pages.
- Ellis, Margaret A. and Bjarne Stroustrup *Annotated C++ Reference Manual* Reading, MA: Addison-Wesley, 1990, 447 pages.
- Dewhurst, Stephen C. Kathy T. and 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.
- Winblad, Ann L.; King, Samuel D. and King, David R. *Object-Oriented Winblad*, Ann L., Edwards, Samuel D. and King, David R. *Object-Oriented Software* Reading, MA: Addison-Wesley, 1990, 291 pages.
- *Zortech C++, Compiler Reference*. Arlington, MA: Zortech Incorporated, 1990, 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 1—Hello World! tells you how to initialize (first four tutorials) and modify (last tutorial) the main components of Zinc Interface Library. 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).
- overriding the pre-defined help and error system stubs with Zinc supported help and error window systems (second tutorial).

- using persistent window objects created with the interactive design tool (third tutorial).
- re-defining the keyboard and color mapping associated with Zinc class objects (fourth tutorial).

We recommend you read this section before later sections, so you understand the Zinc initialization process used by all subsequent tutorials in this manual.

Section 2—Dictionary describes the transition from C to C++, building Zinc Interface Library applications, using Zinc Designer, and using the Zinc data file for load/store operations.

Section 3—Zinc Application Program describes the overall design and implementation issues you should be concerned about when creating applications using C++ and Zinc Interface Library. This set of tutorials examines an application program to show how Zinc Interface Library'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 normal applications.

Section 4—Derived Classes contains a set of tutorials that show how Zinc Interface Library 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 input 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 window object and how to cause other window objects to interact with it.

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 graphics display classes are implemented from the base UI_DISPLAY class.

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

These tutorials begin by comparing the basic storage techniques employed by both C and C++. They then progress to a discussion of the implementation techniques used to store Zinc window objects.

Section 6—Appendixes addresses other topics that may be useful when developing applications. The following information is contained in appendix chapters:

Compiler considerations—This appendix tells about the compiler dependencies that need to be set when you are using Zinc Interface Library. It addresses Borland's IDE and makefile dependencies, Zortech's ZWB and makefile dependencies, and Microsoft's PWB and makefile dependencies.

Compiled BGI files—This appendix shows you how to compile and link Borland BGI files into your application program.

Example Programs—This appendix lists support programs that are designed to help you with specific implementation issues you may have while using Zinc Interface Library.

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 example code contained in the Zinc Interface Library manuals.

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

NOTE: All the figures in these tutorial were taken from the Windows environment. The actual presentation of a particular window may vary slightly, depending on the type of display mode you are currently using.

If you need assistance with the tutorial programs or example programs, or have questions in general, please contact our technical support group (801) 785-8998. They are available on weekdays between the hours of 8:00 a.m. and 5:00 p.m., mountain standard time.

In addition, our bulletin board system is continually up-to-date with example programs, updates and ideas concerning Zinc Interface Library. This service is available 24 hours a day by calling (801) 785-8997 with 300-2400 baud or by calling (801) 785-8995 with 300-9600 baud. The Zinc bulletin board system operates at 8 data bits, no parity and 1 stop bit.

No set of tutorials can address all 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 Interface Library.

SECTION I HELLO WORLD!

SECTION I
HELLO WORLD

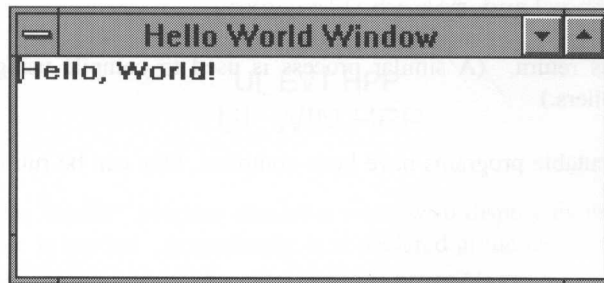
CHAPTER 1 – INITIALIZING THE LIBRARY

The first tutorial program in this section shows you how to set up the basic Zinc Interface Library 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";
}
```

This 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 show how the window will look once the program is complete.



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

- Declaration of include files
- Definition of the screen display (graphics and text)
- 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 minute to read “*Appendix A—Compiler Considerations.*” Most of the programs in these tutorials are have been created to run in either DOS or Microsoft Windows.

Running the program

Before the Hello1 program can be run, it must be compiled. To compile the DOS version of Hello1, using Borland C++, type the following:

```
make -fborland.mak hello1.exe
```

and then press return. To compile the MS Windows version of Hello1, type the following:

```
make -fborland.mak whello1.exe
```

and then press return. (A similar process is used to compile using the Microsoft or Zortech compilers.)

Once the executable programs have been compiled, they can be run by typing:

```
hello1
```

at the DOS prompt. The MS Windows version of the program must be run from inside Windows or by typing: `win whello1` while starting Windows from the command line. To exit either version of the program hold down the <Alt> key and press <F4> or click on the system button with the mouse.

Include files

The first step in writing the “Hello World!” program is declaring the proper include file. Zinc Interface Library allows access to the following include files:

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 all windows and window objects.

These files do not require nor contain any compiler specific include files. (The Microsoft Windows version does include the **WINDOWS.H** header file in **UI_GEN.HPP**.) 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 Interface Library 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>
```

Because of the include file hierarchy, declaring the **UI_WIN.HPP** file also causes the **UI_EVT.HPP**, **UI_DSP.HPP**, and **UI_GEN.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 (__BCPLUSPLUS__) | defined (__TCPLUSPLUS__)
// Borland version.
// Initialize the display, trying for graphics first.
UI_DISPLAY *display = new UI_BGI_DISPLAY;
if (!display->installed)
{
    delete display;
    display = new UI_TEXT_DISPLAY;
}
```

```

#endif
#if defined _MSC_VER
    // Microsoft version.
    // Initialize the display, trying for graphics first.
    UI_DISPLAY *display = new UI_MSC_DISPLAY;
    if (!display->installed)
    {
        delete display;
        display = new UI_TEXT_DISPLAY;
    }
#endif
#if defined __ZTC__
    // Zortech version.
    // Initialize the display, trying for graphics first.
    UI_DISPLAY *display = new UI_FG_DISPLAY;
    if (!display->installed)
    {
        delete display;
        display = new UI_TEXT_DISPLAY;
    }
#endif
#ifdef _WINDOWS
    // Microsoft Windows version.
    // Initialize the Windows display.
    UI_DISPLAY *display = new UI_MSWINDOWS_DISPLAY(hInstance, hPrevInstance,
        nCmdShow);
#endif

```

The `UI_DISPLAY` class is used by all Zinc Interface Library classes that present information to the screen, whether in text or graphics modes of operation. 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 code above with:

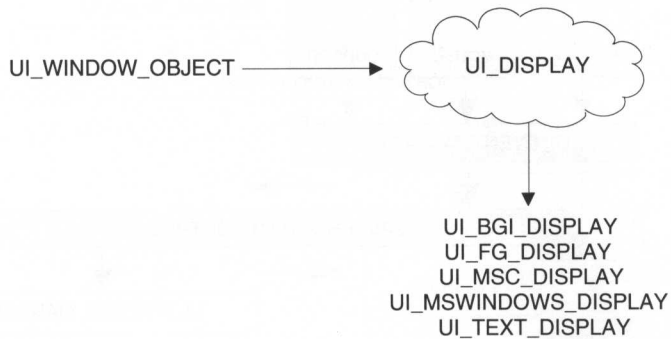
```

// 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.

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 an abstract class from which all Zinc Interface Library 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:

```

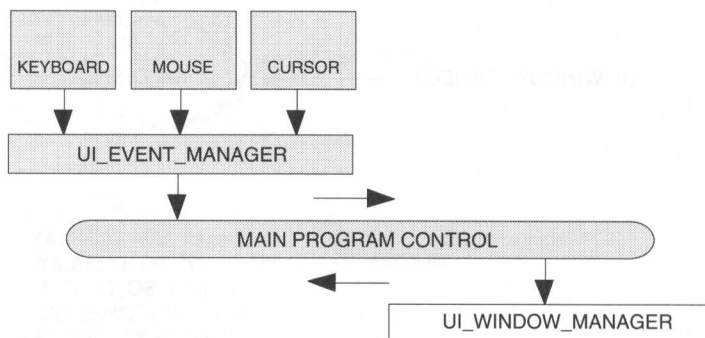
// Initialize the event manager and add three devices to it.
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
*eventManager
+ new UID_KEYBOARD
+ new UID_MOUSE
+ new UID_CURSOR;
  
```

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 overload operator `UI_EVENT_MANAGER::operator +`.

The Conceptual Design chapter of the *Programmer's Guide* discusses the interaction between input devices and the event manager within Zinc Interface Library. The figure below reviews this interaction:



The window manager

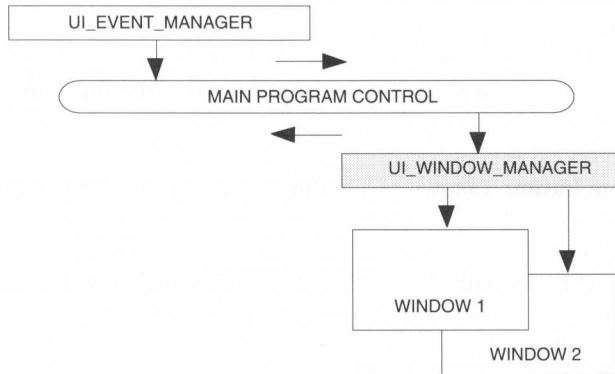
The final basic component of Zinc Interface Library is the window manager, which is added with the following code:

```
// 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 Interface Library.

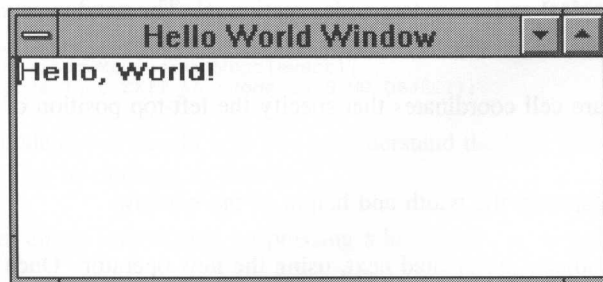
The Conceptual Design also briefly discusses the interaction of the window manager within Zinc Interface Library. The figure below reviews this interaction:



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 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 **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:

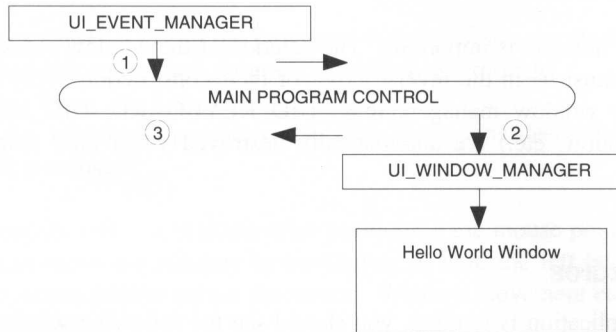
- 5 and 5 are cell coordinates that specify the left-top position of the window on the screen.
- 40 and 6 specify the width and height of the window.

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 object oriented programs is different from structured programs. The “Hello World!” program has a very simple program flow, as is 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;
do
{
    // Get input from the user.
    UI_EVENT event;
    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. 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 Interface Library 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.

Restore—Press <Alt+F5> to restore the original size of the window 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.

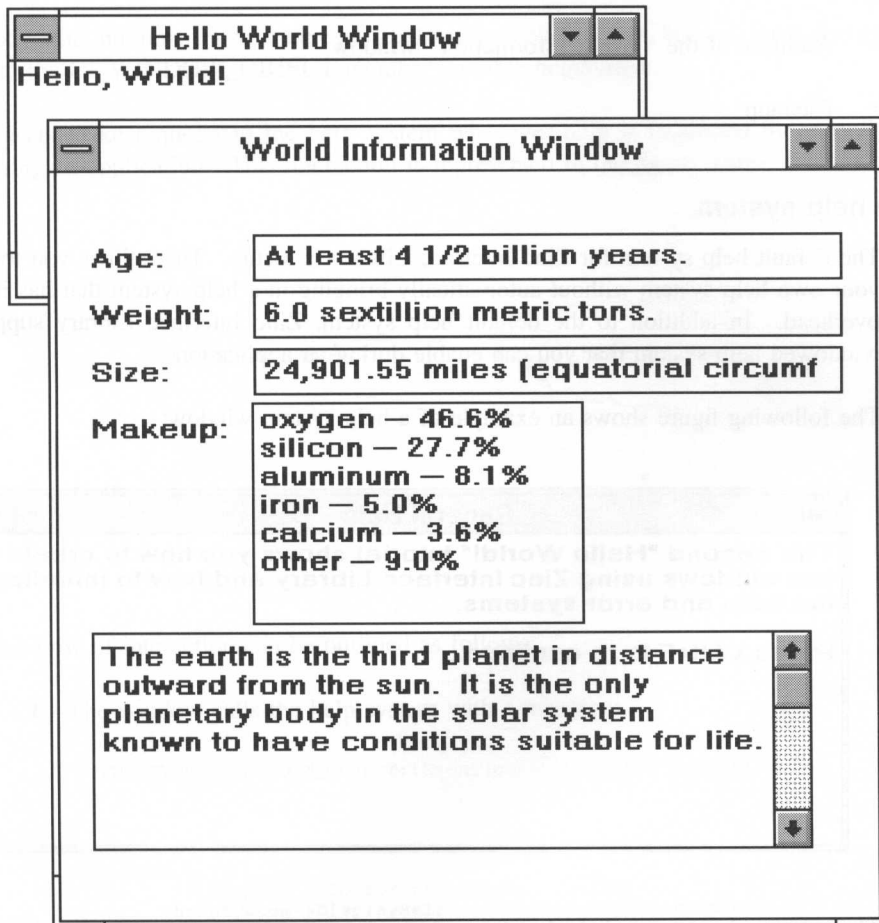
Restore—Click the left button while the mouse pointer is positioned on the maximize button (if the window is in a maximized state) or on the minimize button (if the window is in a minimized state) to restore the original window size.

Exit—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 Interface Library.

CHAPTER 2 – HELP AND ERROR SYSTEMS

Congratulations on completing the first tutorial, where you learned how to set up the basic Zinc Interface Library elements. This tutorial extends the capabilities of the first “Hello World!” tutorial to add windowed help and error systems, an exit function, as well as a “World Information” window. The final outcome should be similar to the following:



The code for this tutorial is located in `\ZINC\TUTOR\HELLOHELLO2.CPP`.

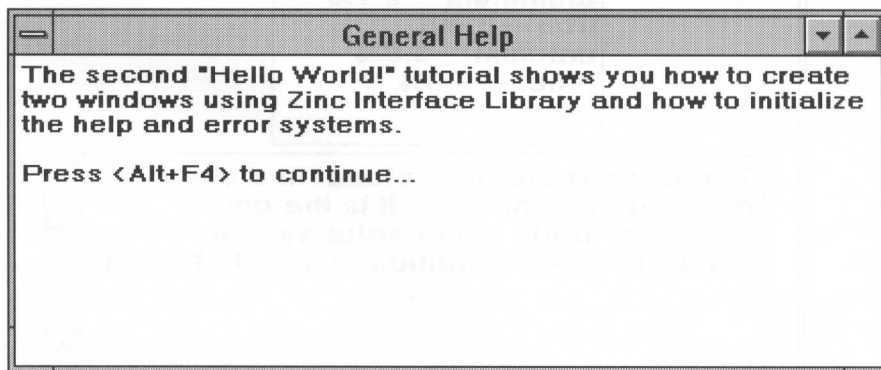
Since this program is an extension of the original “Hello World!” program, only the new components of it will be discussed in this tutorial. These new components are:

- Creation of the help system
- Creation of the error system
- Addition of the exit function
- Addition of the “World Information” window
- Cleanup

The help system

The default help system for Zinc Interface Library is a stub. This allows you to create your own help system without automatically bringing in a help system that has its own overhead. In addition to the default help system, Zinc Interface Library supports a windowed help system that you can enable during an application.

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:

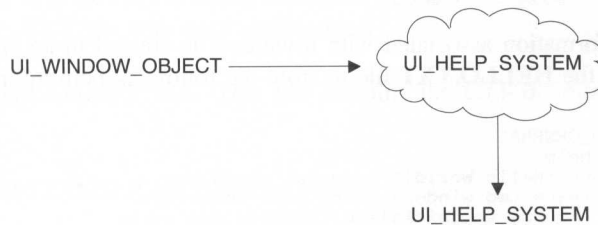
```
UI_WINDOW_OBJECT::helpSystem = new UI_HELP_SYSTEM("hello.dat",  
    &windowManager, HELP_GENERAL);
```

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 within the program.

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;
```

In this program, there is only one help window associated with the help system, thus only one help window is added to the window manager. (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 Interface
Library 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 <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 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:

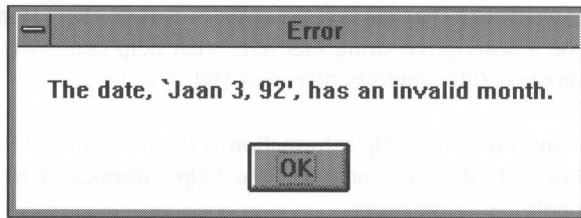
```
#define HELP_GENERAL          1 // General help.
#define HELP_HELLO_WORLD     2 // Hello World.
#define HELP_WORLD_INFORMATION 3 // World Information.
```

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>
#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 Interface Library. 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 *,
                               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 following example, 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 it onto the event queue. When the “OK” button is pressed, an `L_EXIT` message is passed to the window manager 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 the exit function:

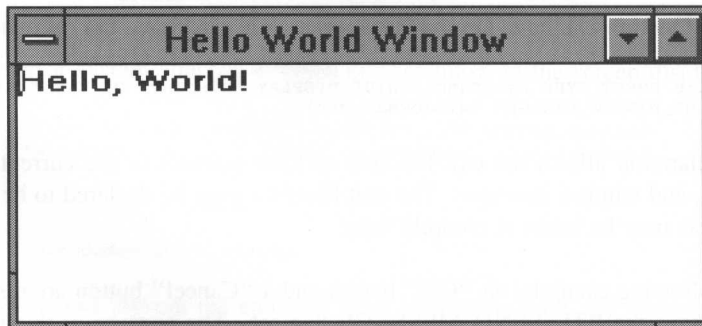
```

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;
    UI_WINDOW *window = new UI_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:



```

// 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 Interface Library objects have a **Generic()** function: **UIW_WINDOW** and **UIW_SYSTEM_BUTTON**. The **UIW_WINDOW::Generic()** member function automatically creates a window with the border, maximize button, minimize button, system button, and title. The following code 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);
}

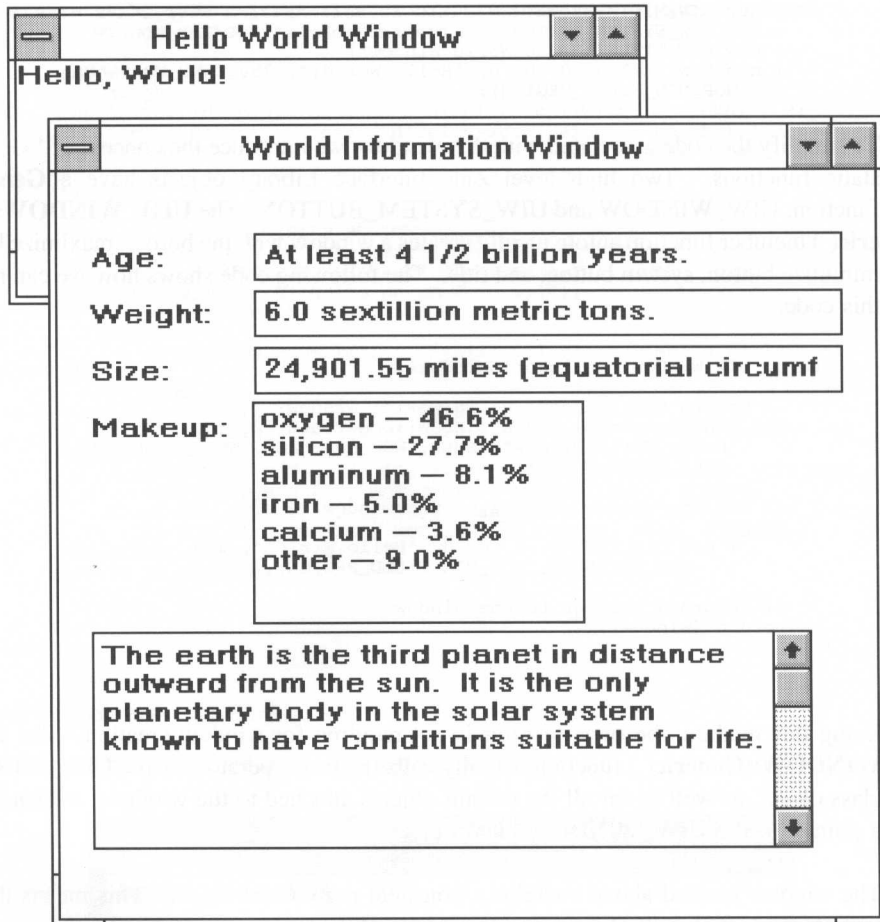
```

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 example program **BIO.CPP** as an example.)

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 is shown below:



The following code is used to create this window:

```
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 (equatorial circumference).", 50)

+ new UIW_PROMPT(2, 4, "Makeup...")
+ &(*new UIW_VT_LIST(12, 4, 20, 4)
  + 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 this window (1) and that used to create the first window (2):

```

new UIW_TEXT(2, 8, 46, 4,                                     (1)
  "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);

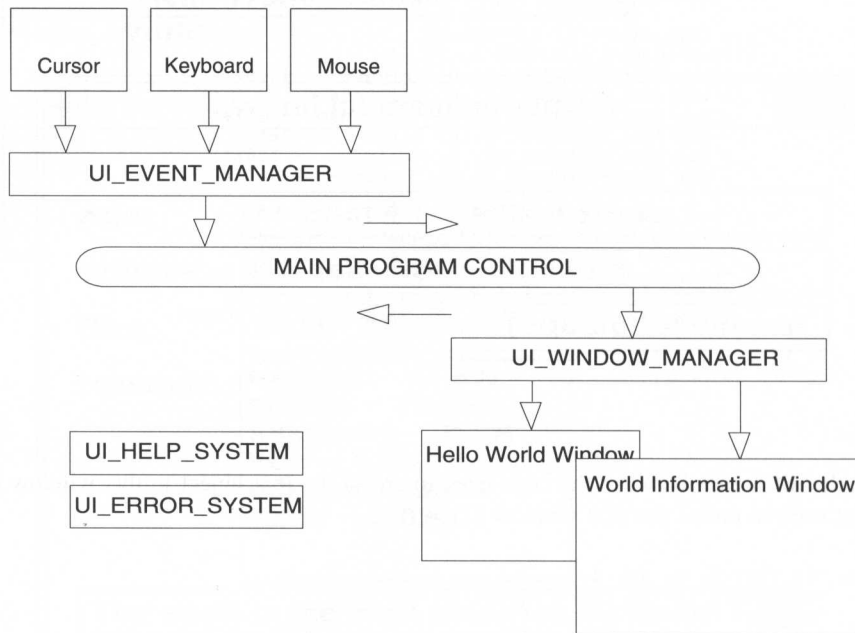
new UIW_TEXT(0, 0, 0, 0,                                     (2)
  "Hello World!", 256, TXF_NO_FLAGS,
  WOF_NON_FIELD_REGION);

```

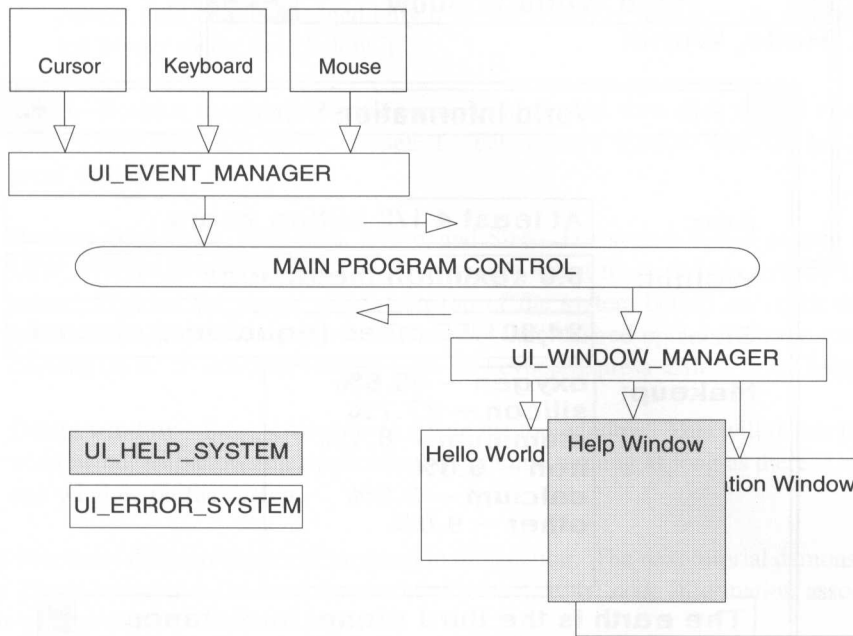
The first code sample defines a position and size indicator and does not set the WOF_NON_FIELD_REGION flag. Instead, it uses WOF_BORDER to display the boundaries of the field's region.

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

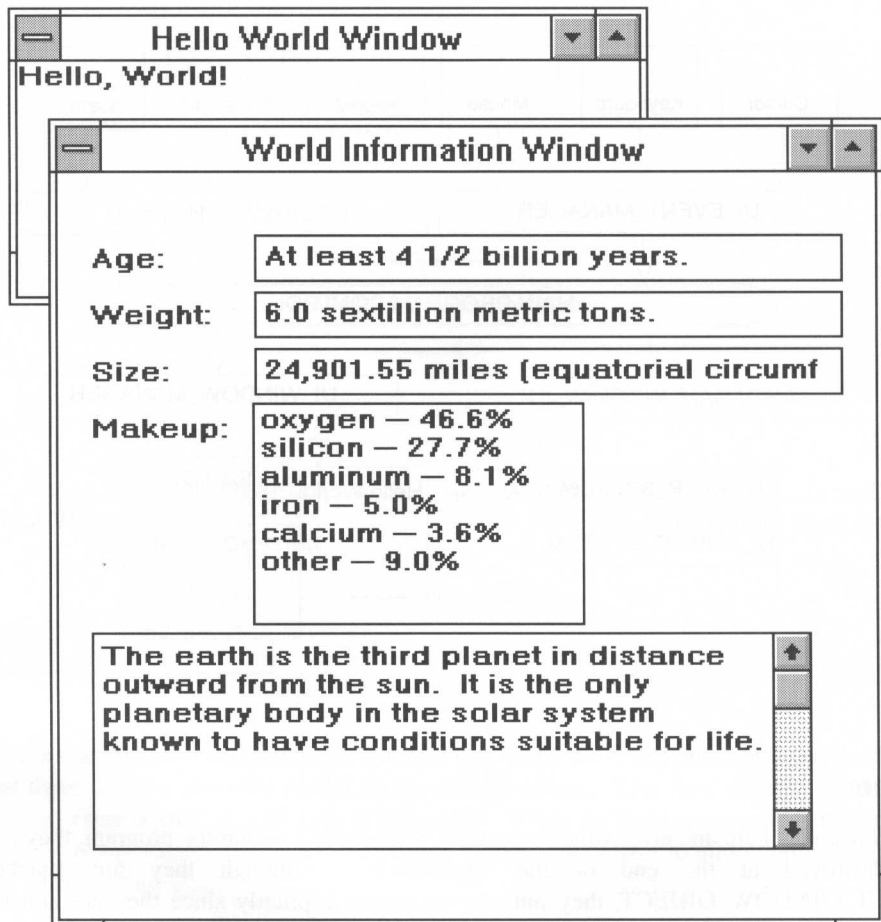
Since new help and error window systems were created within the program, they must be destroyed at the end of the application. Although they are attached to `UI_WINDOW_OBJECT`, they must be destroyed explicitly since they are attached by static pointers.

```

// 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.
- <Up-Arrow> to move to the field immediately above the current field. The current field will be changed only if its left border is vertically aligned with the left border of the field above it.

- <Down-Arrow> to move to the field immediately below the current field. The current field will be changed only if its left border is vertically aligned with the left border of the field below it.

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, and Size—The system button created in the `UIW_WINDOW::GENERIC()` 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.

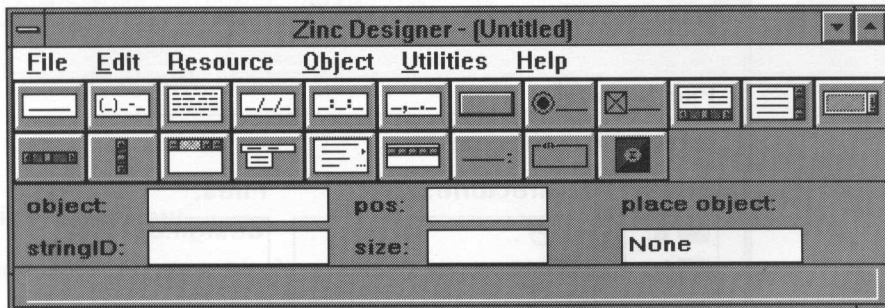
CHAPTER 3 – USING ZINC DESIGNER

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\HELLOHELLO3.-CPP`.

Zinc Designer lets you create windows interactively and then incorporate them inside 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>
```

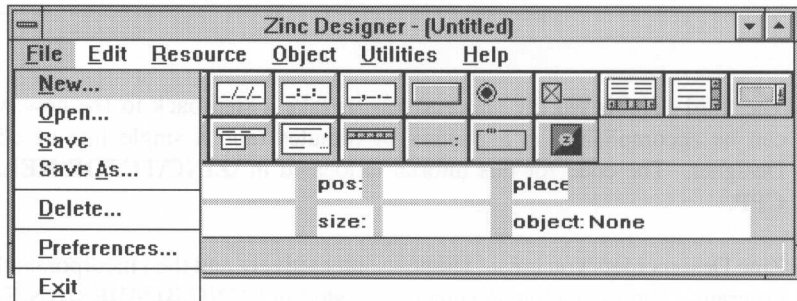
Once the application is running, the following window should be visible on the screen:



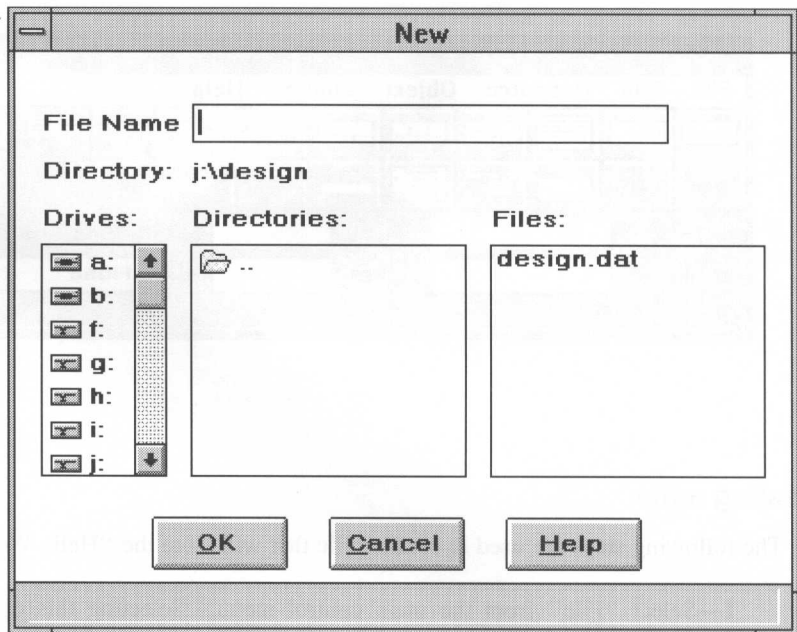
Creating a file

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:



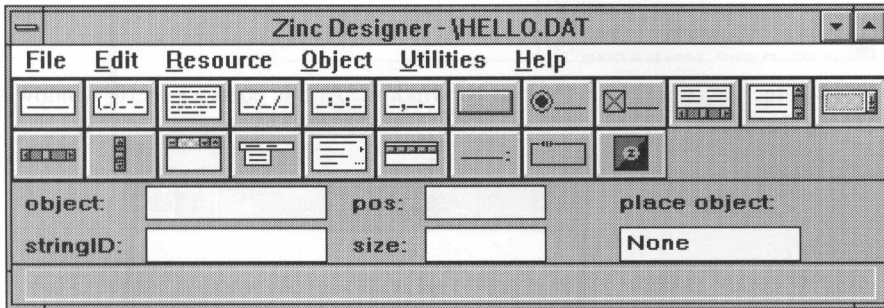
3—Enter the file name by typing `hello` in the field adjacent to the “File name” prompt.

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

4—Create the file by selecting the “OK” button.

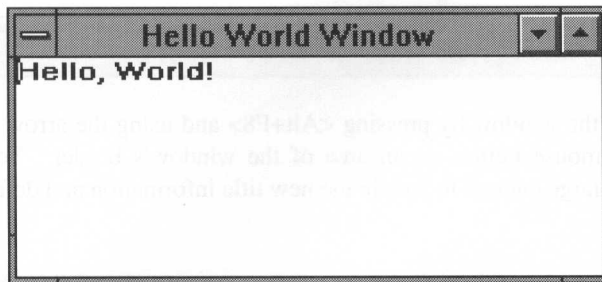
Once the OK button has been selected, Zinc Designer does the following:

- 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



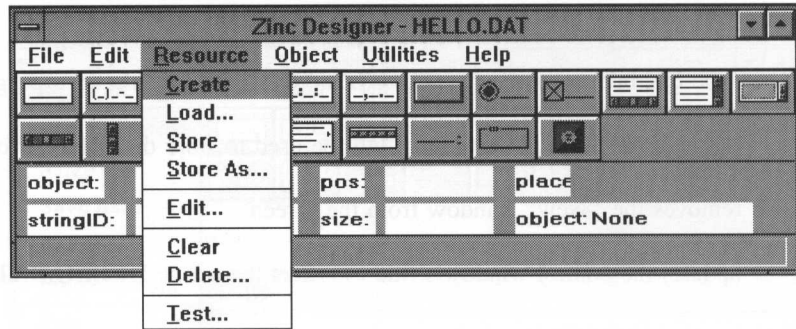
Creating a window

The window we created in the second “Hello, world” was:

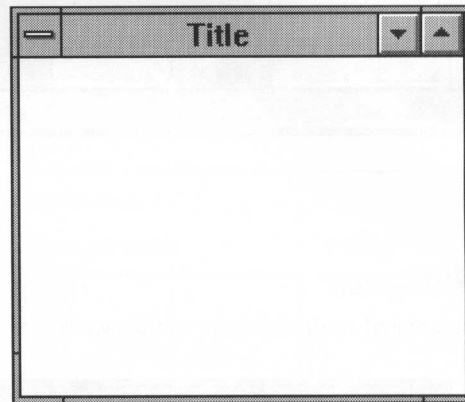


This window is created interactively in the following steps:

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

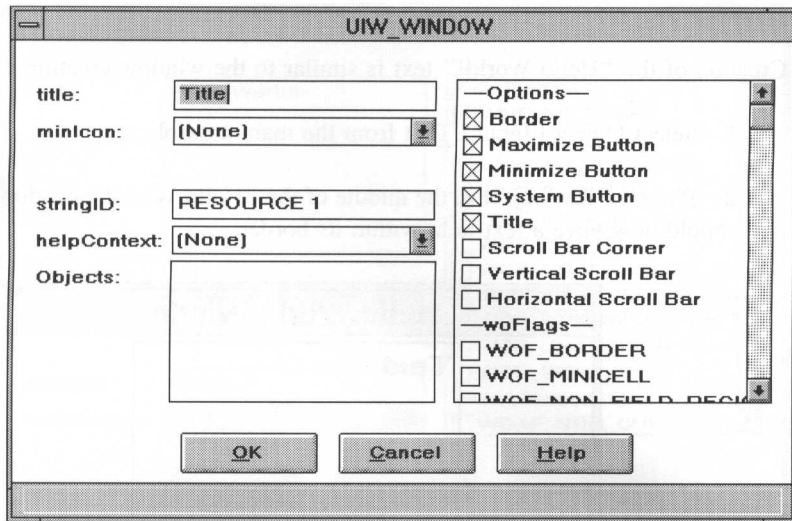


2-Select “Create” from the pop-up window. At this point a generic window appears on the screen:



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:

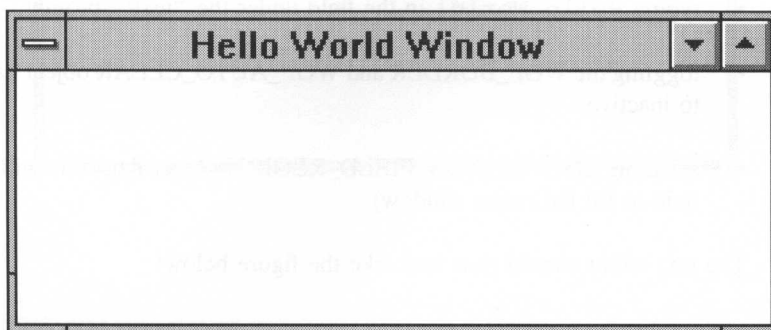


5—Enter the window identification by typing `HELLO_WORLD_WINDOW` in the field adjacent to the “stringID:” prompt.

6—Save the identification by selecting the “OK” button.

7—Enter `Hello World Window` in the “title:” field.

Your window should now look similar to the figure below:

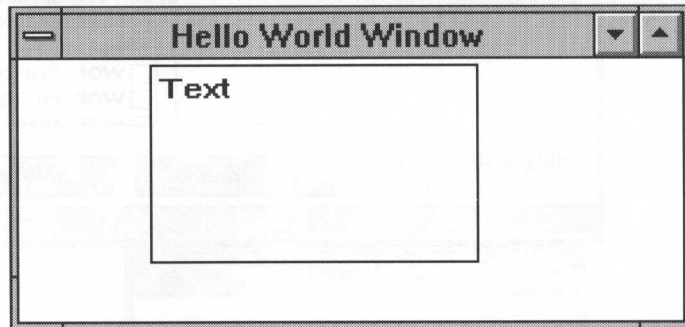


Creating a window object

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

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

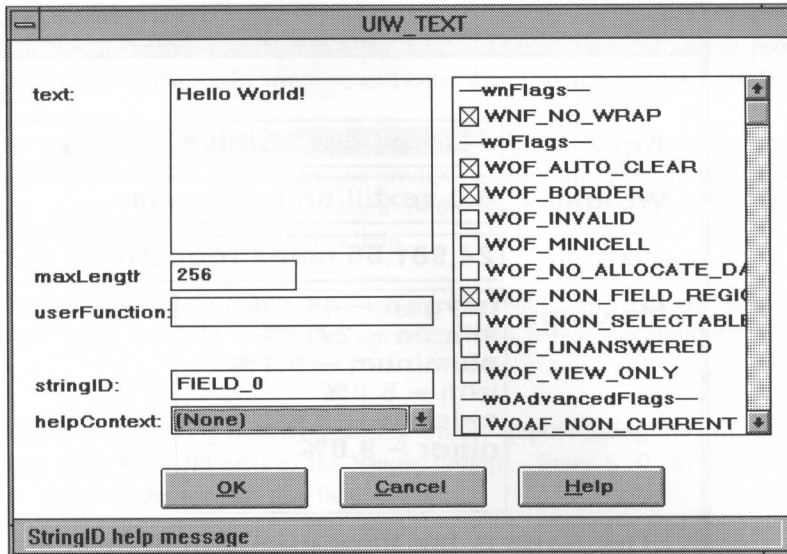
2—Place the text object in the middle of the “Hello World!” window. Your window should now have a text field within its border:



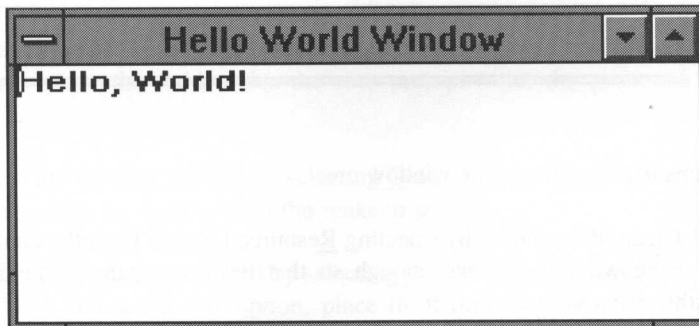
3—Change the default information associated with the text object by:

- calling the text editor
- typing 256 in the field adjacent to the “maxLength:” prompt
- typing Hello World! in the field under the “text:” prompt
- toggling the WOF_BORDER and WOF_AUTO_CLEAR object flags from active to inactive
- selecting the WOF_NON_FIELD_REGION object flag (this will cause the text field to fill the entire window)

The text editor should now look like the figure below:

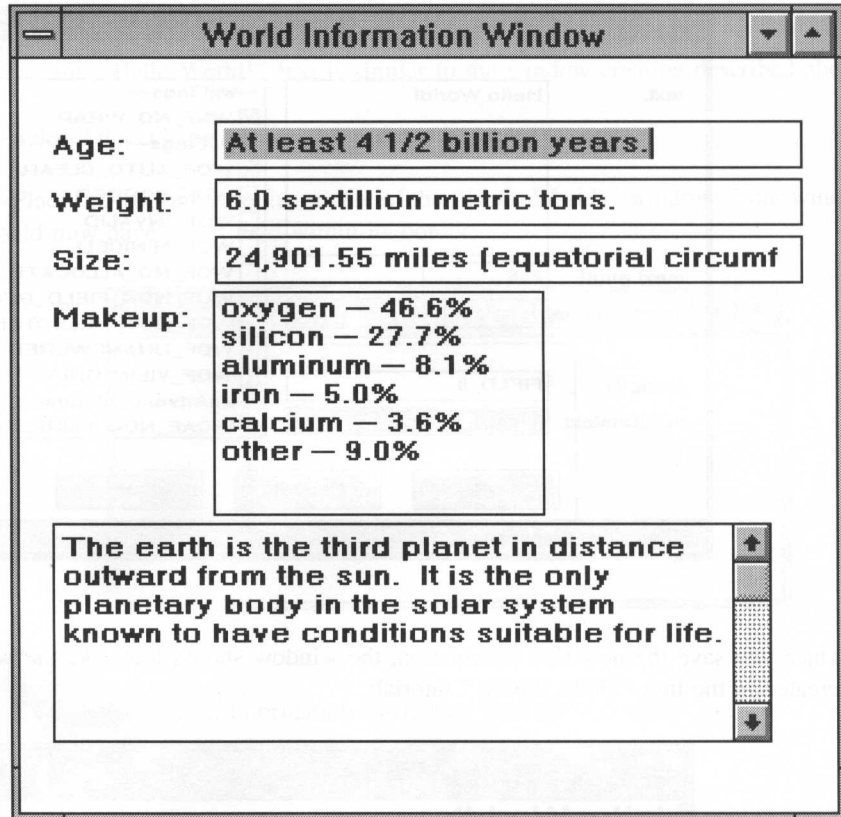


Once you save the new text information, the window should look like the window we created in the first “Hello World!” tutorial:



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 **R**esource | **C**reate 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 identification by calling the window editor and entering `WORLD_INFORMATION_WINDOW` as the *stringID*.
- 3—Change the window title by calling the title editor and entering the title `World Information Window`.

4—Create the age prompt by selecting Object | Static | Prompt from the control menu, then 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.

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

6—Create the weight prompt by selecting Object | Input | String from the control menu, then place the field under the age prompt. Change the prompt's text to `Weight:`.

7—Create the weight string field by selecting Object | Input | String from the control menu, then place the field next to the weight prompt. Enter `6.0 sextillion metric tons` as the text for this field.

8—Create the size prompt by selecting Object | Static | Prompt from the control menu, then place the field under the weight prompt. Set `Size:` as the prompt's text.

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

10—Create the makeup prompt by selecting Object | Input | Prompt from the control menu, then place the field under the size prompt. Set the prompt's text to be `Makeup:`.

11—Create the makeup list box by selecting Object | Input | Vt-List from the control menu, then place the field next to the makeup prompt.

12—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%

13—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 100 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.

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 created—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 code inside this function depends on the type of objects that were created in the designer.

A HELLO.HPP file is created—This file contains the numeric identifications (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~HELLO_WORLD_WINDOW")
+ new UIW_WINDOW("HELLO~HELLO_WORLD_INFORMATION");
```

In the code above, `HELLO_WORLD_WINDOW` and `HELLO_WORLD_INFORMATION` 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~HELLO_WORLD_WINDOW")
+ UI_WINDOW_OBJECT::New("HELLO~HELLO_WORLD_INFORMATION");
```

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 third tutorial program in this section. The final “Hello World!” tutorial introduces you to the color and event mapping systems used by Zinc Interface Library.

SECTION II DICTIONARY

01-10-10
11-11-10

Chapter 4 – DISCOVERING 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 language.

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. After completing this tutorial, you should be able to understand:

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

In order to run the programs, it is necessary to compile them. To compile **WORD1A.-EXE**, type the following:

```
make word1a.exe
```

and then press return. To compile **WORD1B.EXE**, type the following:

```
make word1b.exe
```

and then press return.

Once the executable 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 the program type the following at the DOS prompt:

```
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 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 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 may not 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 if the file **WORD1A.C**, is used to read a word from the dictionary, check to see if the word is the same as the search word and then to print it out.

```

while (!feof(file))
{
    ReadWord(file, &word);
    if (!strcmpi(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 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 attributes and functions of the base class. A good example of this is found in the file **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); }
};

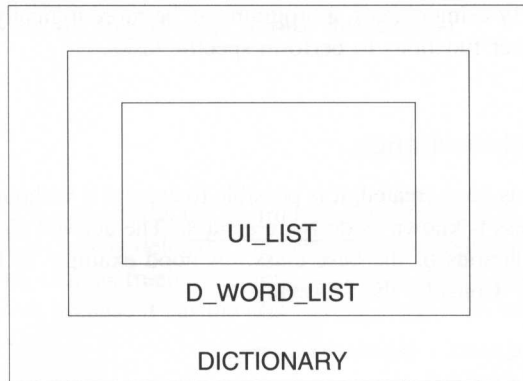
class DICTIONARY : public D_WORD_LIST
{
public:
    int found;

    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 have 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 parameters 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. The line of code "int count = 3 + 9;" will still give the desired value. 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 the function main of **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 Interface Library 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()` and `WinMain()` (for Microsoft Windows) functions. It, also, contains 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 data base file.

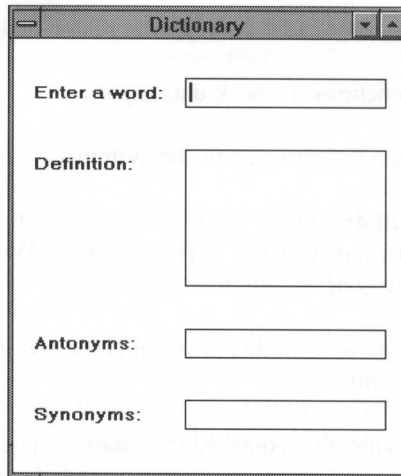
BORLAND.MAK, **MICROSOFT.MAK**, and **ZORTECH.MAK**—These are the makefiles associated with the virtual list program. You can compile the DOS version, **WORD2.EXE**, by typing `make -fborland.mak word2.exe` at the command line prompt. To make the Windows version, **WORD2.EXE**, type `make -fborland.mak word2.exe` at the command line. (**NOTE:** When compiling with either Microsoft or Zortech, substitute the name of that compiler as the name of the make file on the command line.)

WORD.DEF—This is the definition file used when compiling a Microsoft Windows application.

WWORD.RC—This is the resource file used when compiling a Microsoft Windows application.

Program execution

The operation of the dictionary 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.

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();
    ~DICTIONARY_WINDOW();

    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 this 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 the 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() : 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 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 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 editable 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, DICTONARY_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 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 *cocode* is the value returned. However, if -1 is returned, the calling window object will be informed that some error occurred and that it should remain the current object.
- *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 DICTONARY_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.
- *cocode_{in}* is the logical interpretation of the event that caused this function to be called.

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

```
#pragma argsused
EVENT_TYPE DICTONARY_WINDOW::LookUpWord(UI_WINDOW_OBJECT *object,
    UI_EVENT &event, EVENT_TYPE cocode)
{
    .
    .
    .
}
```

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 cocode is S_CURRENT. In the dictionary tutorial, if the input string field is just becoming current, then a new word has not be entered and the function returns without doing anything. Examine the initial check in **LookUpWord()**:

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

If the input field is becoming non-current, then 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 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, then 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 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.

```
do
{
    // Get input from the user.
    UI_EVENT event;
    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. After polling and putting new events on the queue, the **Get()** function removes the first event from the queue (i.e., the "g" key event) and returns it to the calling function. When program control returns from the **Get()** function, the event returned is given 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 an 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, then 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—MS Windows

The MS Windows version of Zinc Interface Library 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 Windows 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 **WinMain()** :

```
do
{
    // Get input from the user.
    UI_EVENT event;
    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 data base you can use this program as a template and instead of using the class **DICTIONARY**, you will make the appropriate calls to your data base program.

CHAPTER 6 – THE ZINC DATA FILE

This tutorial demonstrates data base interaction with Zinc Interface Library 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()** and **WinMain()** (for Microsoft Windows) functions. It also contains 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 stringID's of the file objects in the data file. It also contains the help file header information for the **WORD_WIN.DAT** file.

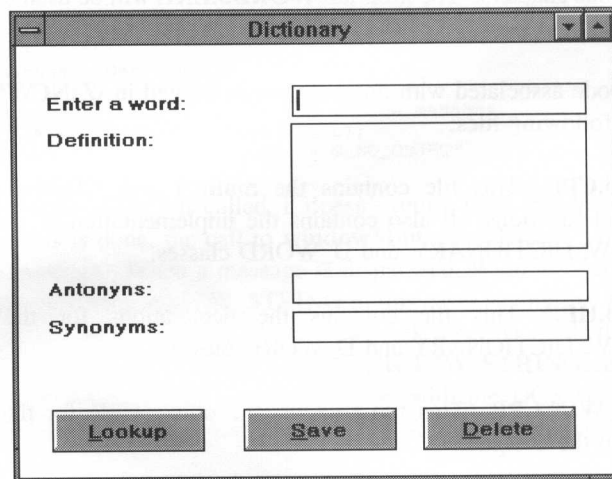
BORLAND.MAK, **MICROSFT.MAK**, and **ZORTECH.MAK**—These are the makefiles associated with the dictionary program. You can compile the DOS version, **WORD3.EXE**, by typing `make -fborland.mak word3.exe` at the command line prompt. To make the Windows version, **WWORD3.EXE**, type `make -fborland.mak wword3.exe` at the command line. (**NOTE:** When compiling with either Microsoft or Zortech, substitute the name of that compiler as the name of the make file on the command line.)

WORD.DEF—This is the definition file used when compiling a Microsoft Windows application.

WORD.RC—This is the resource file used when compiling a Microsoft Windows application.

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:



The screenshot shows a window titled "Dictionary". Inside the window, there are four input fields and three buttons. The first field is labeled "Enter a word:" and is a single-line text box. The second field is labeled "Definition:" and is a larger multi-line text box. The third field is labeled "Antonyms:" and is a single-line text box. The fourth field is labeled "Synonyms:" and is a single-line text box. At the bottom of the window, there are three buttons: "Lookup", "Save", and "Delete". Each button has a small icon on its left side.

At this point, the dictionary data base 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();

    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, TRUE);
    ~DICTIONARY();

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

The definition for the **D_WORD** class is as follows:

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

    D_WORD(const char *name, UI_STORAGE *file, UI_STORAGE_OBJECT *object);
    D_WORD(const char *name);
    ~D_WORD();
    static D_WORD *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
};
```

D_WORD uses the following member variables:

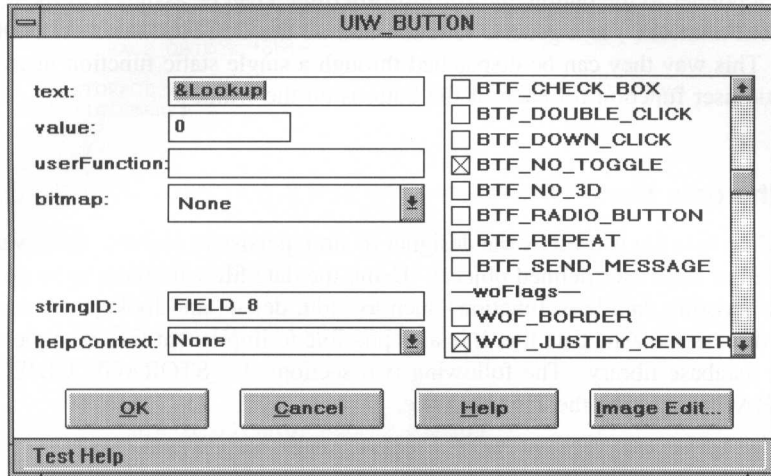
- *word* is a variable that contains the actual word entry in the dictionary.
- *definition* is a variable that contains the definition string of the word.
- *antonym* is a the list of antonyms that apply to the dictionary entry.
- *synonym* is the list of synonyms that apply to the dictionary entry.

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. Steps on using the designer are contained in the “*Using Zinc Designer*” chapter.

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 is `LOOKUP_BUTTON` and should be entered in the 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, 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 *numberID* (*numberID* is available by using the `#define` values created by the designer.) of the "Lookup" button. This will return a `UI_WINDOW_OBJECT` 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.
unsigned short id = LOOKUP_BUTTON;
UIW_BUTTON *button;
button = (UIW_BUTTON *)INFORMATION(GET_NUMBERID_OBJECT, &id, ID_WINDOW);
button->userFunction = DICTIONARY_WINDOW::ButtonFunction;
.
.
}

```

With a pointer to the button, the **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 static 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 data base 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_WORD` class is derived from `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_WORD` is derived from `UI_STORAGE_OBJECT`, 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 **Store()**.

The constructor

There are two constructors used for the `D_WORD` class. This first takes a `const char *` as a parameter and is used to create a `D_WORD` class for a new word that is not in the data file.

```

D_WORD::D_WORD(const char *name)
{
    // Used to create a new word that is not in the file.
    word = definition = antonym = synonym = NULL;
}

```

The second constructor is used to read in the D_WORD from the data file or if it is not found, it will create one with all of the fields set to NULL.

```

#pragma argsused
D_WORD::D_WORD(const char *name, UI_STORAGE *file,
    UI_STORAGE_OBJECT *object)
{
    UI_STORAGE_OBJECT *element = file->FindFirstObject(name);
    if (!element)
    {
        // The object was not found in the file.
        wasLoaded = FALSE;
        word = definition = antonym = synonym = NULL;
    }
    else
    {
        wasLoaded = TRUE;

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

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

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

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

```

If the word entry is found in the file, then its entire object is read in. In turn, each of the object's fields are read. Each field is preceded by an unsigned long 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 read in, a function called **D_WORD::New()** is called. The **New()** function discussed here is a member of a class and not the **new** operator of C++.

In this tutorial, the **New()** function is used only to distinguish between constructors. The reason for having a static **New()** function in a class is to be able to take the address of a procedure that will call the constructor. A good example of this can be seen in the implementation of an object list. Examine the following object table taken from the file **WORD_WIN.CPP**:

```
UI_ITEM_UI_WINDOW_OBJECT::objectTable[] =
{
    { ID_BORDER, &UIW_BORDER::New, "BORDER", 0 },
    { ID_BUTTON, &UIW_BUTTON::New, "BUTTON", 0 },
    { ID_PROMPT, &UIW_PROMPT::New, "PROMPT", 0 },
    { ID_STRING, &UIW_STRING::New, "STRING", 0 },
    { ID_TEXT, &UIW_TEXT::New, "TEXT", 0 },
    { ID_WINDOW, &UIW_WINDOW::New, "WINDOW", 0 },
    { ID_OBJECT_LIST, &UIW_OBJECT_LIST::New, "OBJECT_LIST", 0 },
    { ID_END, NULL, NULL, 0 }
};
```

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 Store function

The purpose of the **Store()** function is to store the object into the file. Each of the object's members are saved. Succeeding fields are saved in the same manner. Each

object is stored with a call to **Store()**. In order to “commit” the object to permanent storage, **UI_STORAGE::Save()** must be called.

UI_STORAGE

The class **DICTIONARY** is derived from **UI_STORAGE**. The **UI_STORAGE** class is used to read or write Zinc Interface Library 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., objects) 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. Since the class **DICTIONARY** is derived from **UI_STORAGE**, it can use inherited functions such as: **Load()**, **Store()** and **Subtract()** to manage the words in the dictionary.

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 enhancements ideas for **DICTIONARY** program include: creating multiple types of persistent objects for use in the same data file or using a third party data base instead of the Zinc data file.

SECTION III ZINC APPLICATION PROGRAM

SECTION III
ZINC INTERFACE LIBRARY 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 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()** and **WinMain()** (for Microsoft Windows) functions. These functions are used to initialize and restore the application.

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 if persistent window objects are to be used.

SUPPORT.DAT—This is a binary data file that 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. This file 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::Option_Help()** member function. It processes all the messages that are produced when a “Help” menu item is selected from the main control window.

BORLAND.MAK, MICROSOFT.MAK, and ZORTECH.MAK—These are the makefiles associated with the virtual list program. You can compile the DOS version, **ZINCAPP.EXE**, by typing `make -fborland.mak zincapp.exe` at the command line prompt. To make the Windows version, **WZINCAPP.EXE**, type `make -fborland.mak wzincapp.exe` at the command line. (NOTE: When compiling with either Microsoft or Zortech, substitute the name of that compiler as the name of the make file on the command line.)

WINDOW.CPP—This file contains the **CONTROL_WINDOW::Option_Window()** member function. This function is used to invoke the proper window that was selected from the main control window.

These functions process all the messages that are produced when a menu item is selected from the main control window.

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 consist 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.

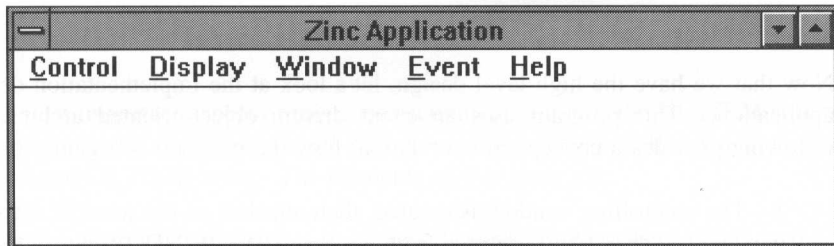
Window objects—The goal of this area is to show Zinc Interface Library as a complete user interface package. This goal is accomplished by showing the many different types of user interface objects that can be created using the library.

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.

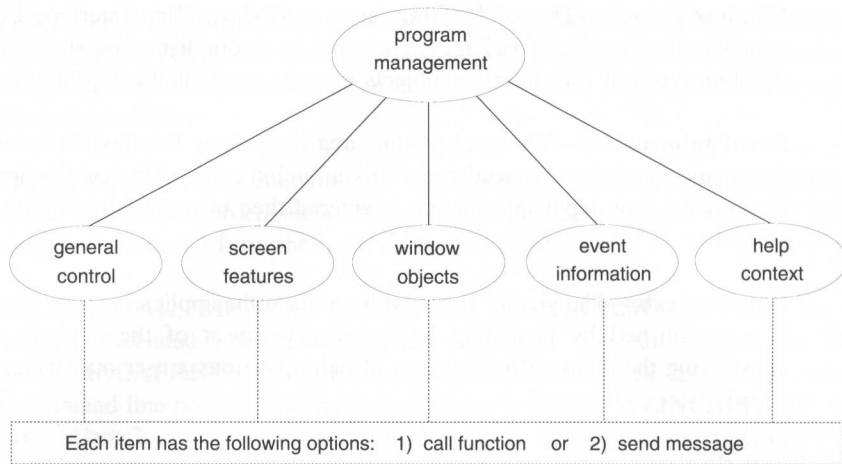
Help contexts—The goal of this area is to make the application user friendly. This is accomplished by providing help at every aspect of the application, and by considering the many different types of help questions a user may have.

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 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 some screen features control unit. It, in turn, will either send a message through the system, requesting that some action be performed, perform the action itself, 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, then 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() : UIW_WINDOW(0, 0, 76, 6, WOF_NO_FLAGS,
    WOAF_LOCKED)
{
    // Control menu items.
    UI_ITEM controlItems[] =
  
```

```

{
    { S_DEINITIALIZE, Message, "&Clear\tShift+F5",
      MNIF_NO_FLAGS },
    { S_REDISPLAY, Message, "&Refresh\tShift+F6",
      MNIF_NO_FLAGS },
    { 0, MNIF_NO_FLAGS }, // item separator
    { L_EXIT_FUNCTION, 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)
      + new UIW_PULL_DOWN_ITEM("&Control objects",
        WNF_NO_FLAGS, controlObjectItems)
      + new UIW_PULL_DOWN_ITEM("&Input objects", WNF_NO_FLAGS,
        inputObjectItems)
      + new UIW_PULL_DOWN_ITEM("&Selection objects",
        WNF_NO_FLAGS, selectionObjectItems)
      + 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 `C`ontrol menu item (`C`lear) contains the message `S_DEINITIALIZE`. This message will be passed through the system whenever the `C`ontrol | `C`lear 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 to the screen. In the case of the `C`lear menu item this string is “Clear\tShift+F5”. (The “Shift+F5” portion of the string is discussed later in this chapter.)

2—The window manager dispatches all messages to the front window (in our case the main control window). When the main control window receives its input it dispatches the information according to its 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 are operations that it knows how to handle.

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 resetting the display itself:


```

void CONTROL_WINDOW::OptionDisplay(EVENT_TYPE item)
{
#ifdef _Windows
    // 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
}

```

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 Interface Library.

Accelerator keys

There are four accelerator keys defined for this program:

<Shift+F5>—Pressing this key combination causes all but the main control window to be removed from the screen.

<Shift+F6>—Pressing this key combination causes the window manager to clear the screen and 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_F5,      S_DEINITIALIZE },
            { 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
            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 four 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. The accelerator implementation given above only applies to the scope for which it applies.

Structured programming

Quite often, structured programming techniques are used to program with Zinc Interface Library. 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);
```

```

UI_ITEM help[] = // Help menu items.
{
    { 0, HelpKeyboard, "&Keyboard", MNIF_NO_FLAGS },
    { 0, HelpMouse, "&Mouse", MNIF_NO_FLAGS },
    { 0, HelpCommands, "&Commands", MNIF_NO_FLAGS },
    { 0, HelpProcedures, "&Procedures", MNIF_NO_FLAGS },
    { 0, HelpObjects, "&Objects", MNIF_NO_FLAGS },
    { 0, HelpHelp, "&Using help", MNIF_NO_FLAGS },
    { 0, 0, "", 0 }, // item separator
    { 0, HelpZincApp, "&About ZincApp...", 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 **HelpIndex()** function might be implemented:

```

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

```

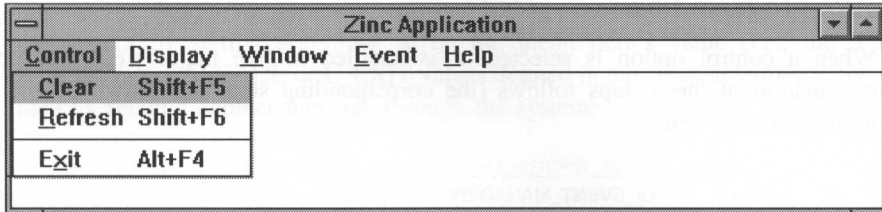
While this method of implementation works, it has several drawbacks:

- 1—It causes you to have a lot of duplicate definitions and operations. You can see from the help example above, it would take seven functions to do the work the **CONTROL_WINDOW::Option_Help()** 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 Interface Library. 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 Interface Library. 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() : UIW_WINDOW(0, 0, 76, 6, WOF_NO_FLAGS,
    WOAF_LOCKED)
{
    // Control menu items.
    UI_ITEM controlItems[] =
    {
        { S_DEINITIALIZE, Message, "&Clear\tShift+F5",
          MNIF_NO_FLAGS },
        { S_REDISPLAY, Message, "&Refresh\tShift+F6",
          MNIF_NO_FLAGS },
        { 0, 0, "", 0 }, // item separator
        { L_EXIT_FUNCTION, 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)
      + new UIW_PULL_DOWN_ITEM("&Control objects",
        WNF_NO_FLAGS, controlObjectItems)
      + new UIW_PULL_DOWN_ITEM("&Input objects", WNF_NO_FLAGS,
        inputObjectItems)
      + new UIW_PULL_DOWN_ITEM("&Selection objects",
        WNF_NO_FLAGS, selectionObjectItems))
```

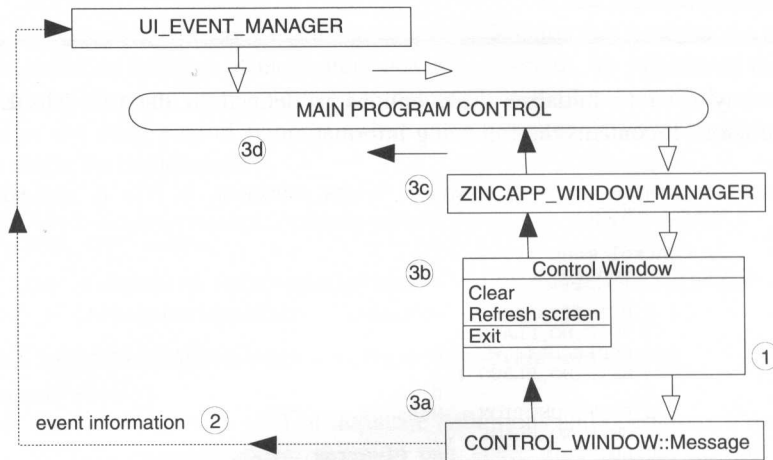
```

+ 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 `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);
}

```

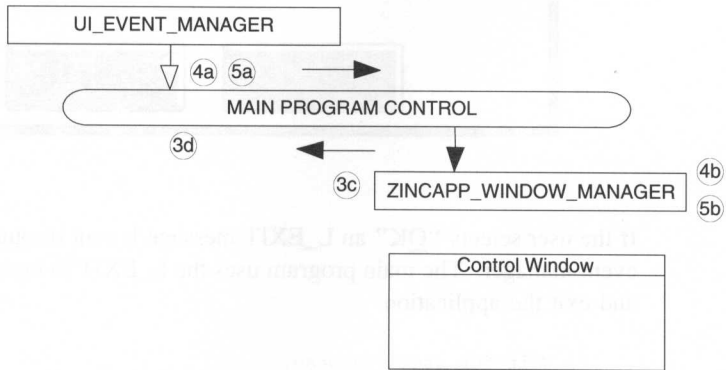
The arguments passed to `Message()` are a pointer to the selected control option (`this`), a copy of the event that caused the user function to be called (`tEvent`), and the logical interpretation (`ccode`) of the event that caused `Event()` to be called. (**NOTE:**

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 manger. 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**, **S_CASCADE** 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 **UI_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

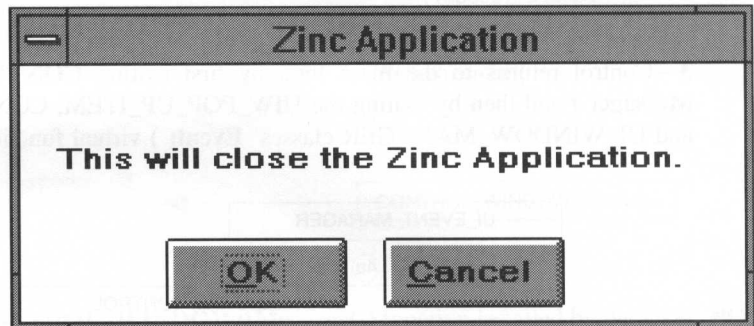
message is handled by the window manager and causes the control options to be removed from the screen.

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_Manger->Event()**. The window manager performs the following actions according to the type of message:

S_CLEAR—Causes all but the main control window to be removed from the screen.

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::Exit()** 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;
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 our part. This control works correctly because the events are passed through the system via the event manager.

`WINDOW` and `WINDOWSET` are the only window management functions in the `WINDOW` module. The `WINDOW` function is used to create a window and the `WINDOWSET` function is used to set the window's title, position, and size. The `WINDOWSET` function is also used to set the window's icon, background color, and border style. The `WINDOWSET` function is also used to set the window's state (maximized, minimized, or normal).

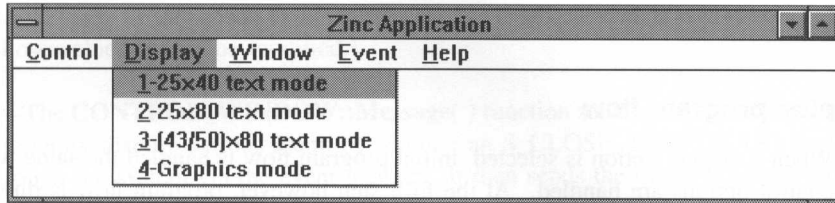
`DISPLAY`—Causes the window manager to close the window that is created in the window manager's list.

`WINDOWSET`—Sets the window's title, position, size, icon, background color, border style, and state.



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:

```
UI_ITEM displayItems[] =
{
#ifdef _Windows
    { MSG_25x40_MODE,      Message,      "&1-25x40 text mode",
      MNIF_NON_SELECTABLE },
    { MSG_25x80_MODE,      Message,      "&2-25x80 text mode",
      MNIF_NON_SELECTABLE },
    { MSG_43x80_MODE,      Message,      "&3-(43/50)x80 text mode",
      MNIF_NON_SELECTABLE },
    { MSG_GRAPHICS_MODE,  Message,      "&4-Graphics mode",
      MNIF_NON_SELECTABLE },
    { MSG_WINDOWS_MODE,   Message,      "&5-Windows 3.x mode",
      MNIF_NO_FLAGS },
#else
    { 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)
```

```

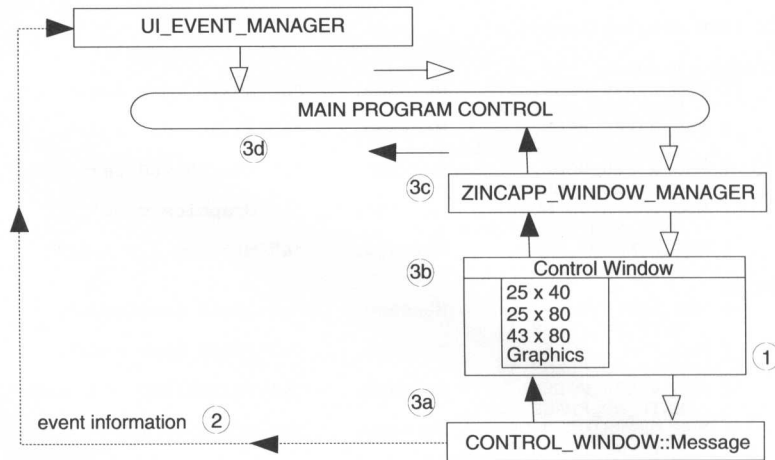
+ new UIW_PULL_DOWN_ITEM("&Control objects", WNF_NO_FLAGS,
    controlObjectItems)
+ new UIW_PULL_DOWN_ITEM("&Input objects", WNF_NO_FLAGS,
    inputObjectItems)
+ new UIW_PULL_DOWN_ITEM("&Selection objects", WNF_NO_FLAGS,
    selectionObjectItems)
+ new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS, eventItems)
+ 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 **Options_Display()** 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 = 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.)

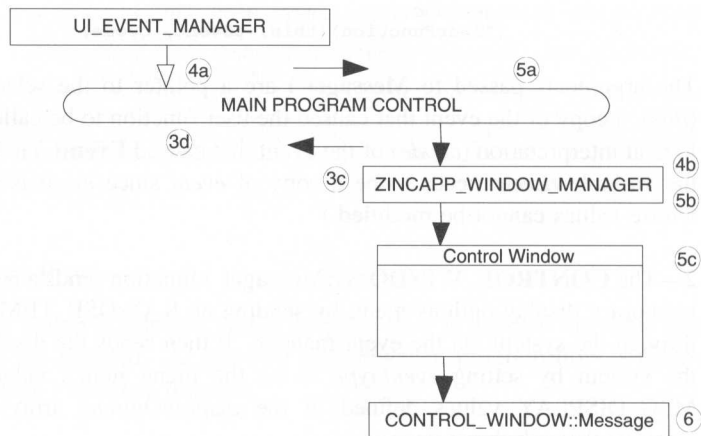
2—The CONTROL_WINDOW::Message() function sends a request to remove the temporary display options menu by sending an **S_CLOSE_TEMPORARY** message through the system via the event manger. It then sends the display request through the system by setting *event.type* to be the menu item's value (i.e., one 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 **UI_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 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 **Option-Display()** 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 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.

```

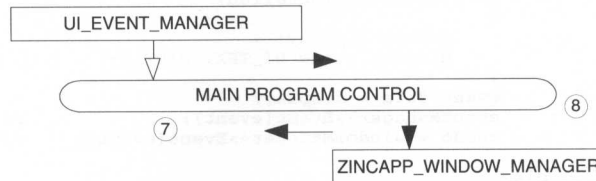
void CONTROL_WINDOW::OptionDisplay(EVENT_TYPE item)
{
#ifdef _Windows
    // Set up the default event.
    UI_EVENT event(S_RESET_DISPLAY, TDM_NONE);

    // 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 **event-Manager->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.)

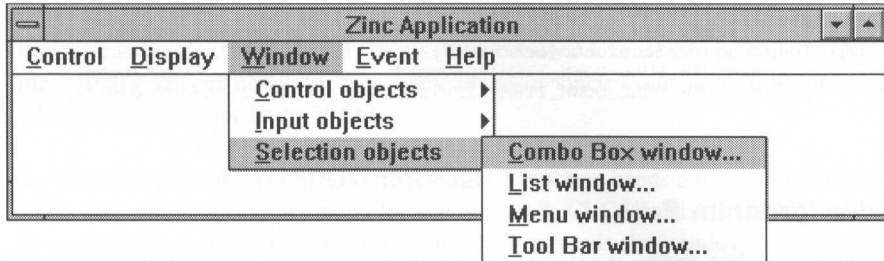
```
// Wait for user response.
EVENT_TYPE ccode;
do
{
    // Get input from the user.
    UI_EVENT event;
    eventManager->Get(event);

    // Check for a screen reset message.
    if (event.type == S_RESET_DISPLAY)
    {
        event.data = NULL;
        windowManager->Event(event);           // Tell the managers we are
        eventManager->Event(event);           // changing the display.
        delete display;
        if (event.rawCode == TDM_NONE)
        #if defined(__BCPLUSPLUS__) | defined(__TCPLUSPLUS__)
            display = new UI_BGI_DISPLAY;
        #endif
        #ifdef __ZTC__
            display = new UI_FG_DISPLAY;
        #endif
        #ifdef _MSC_VER
            display = new UI_MSC_DISPLAY;
        #endif
        else
            display = new UI_TEXT_DISPLAY(event.rawCode);
        if (!display->installed)
        {
            delete display;
            display = new UI_TEXT_DISPLAY;
        }
        event.data = display;
        eventManager->Event(event);           // Tell the managers we
        ccode = windowManager->Event(event); // changed the display.
    }
    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:

```
// Input window object menu items.
UI_ITEM inputObjectItems[] =
{
    { MSG_DATE_WINDOW,      Message,      "&Date window...",  MNIF_NO_FLAGS },
    { MSG_NUMBER_WINDOW,   Message,      "&Number window...", MNIF_NO_FLAGS },
    { MSG_STRING_WINDOW,   Message,      "&String window...", MNIF_NO_FLAGS },
    { MSG_TEXT_WINDOW,     Message,      "&Text window...",  MNIF_NO_FLAGS },
    { MSG_TIME_WINDOW,     Message,      "&Time window...",  MNIF_NO_FLAGS },
    { 0, 0, 0, 0 } // End of array.
};

// Selection window object menu items.
UI_ITEM selectionObjectItems[] =
{
    { MSG_COMBO_BOX_WINDOW, Message, "&Combo Box window...", MNIF_NO_FLAGS },
    { MSG_LIST_WINDOW,     Message, "&List window...",      MNIF_NO_FLAGS },
    { MSG_MENU_WINDOW,     Message, "&Menu window...",      MNIF_NO_FLAGS },
    { MSG_TOOL_BAR_WINDOW, Message, "&Tool Bar window...",  MNIF_NO_FLAGS },
    { 0, 0, 0, 0 } // End of array.
};

// Control window object menu items.
UI_ITEM controlObjectItems[] =
{
    { MSG_BUTTON_WINDOW,   Message, "&Button window...",  MNIF_NO_FLAGS },
    { MSG_GENERIC_WINDOW,  Message, "&Generic window...",  MNIF_NO_FLAGS },
    { MSG_ICON_WINDOW,     Message, "&Icon window...",     MNIF_NO_FLAGS },
    { MSG_MDI_WINDOW,      Message, "&MDI window...",      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)
```

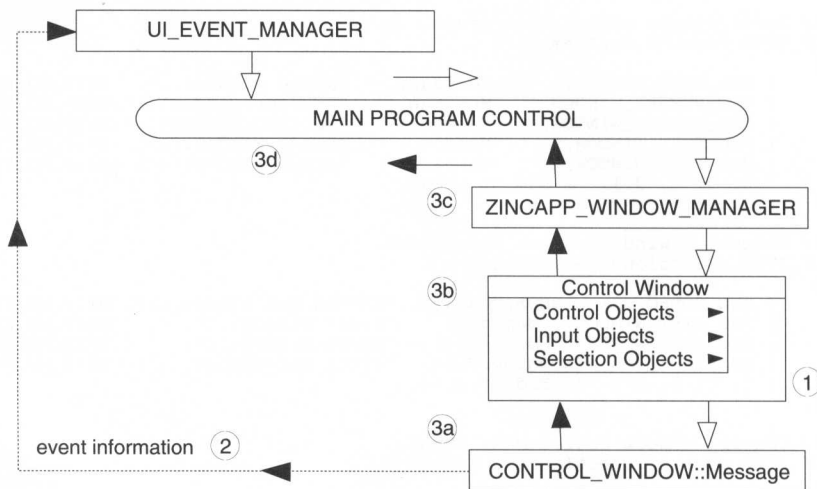
```

+ 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)
    + new UIW_PULL_DOWN_ITEM("&Control objects", WNF_NO_FLAGS,
      controlObjectItems)
    + new UIW_PULL_DOWN_ITEM("&Input objects", WNF_NO_FLAGS,
      inputObjectItems)
    + new UIW_PULL_DOWN_ITEM("&Selection objects", WNF_NO_FLAGS,
      selectionObjectItems))
  + 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 **OptionsWindow()** 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.)

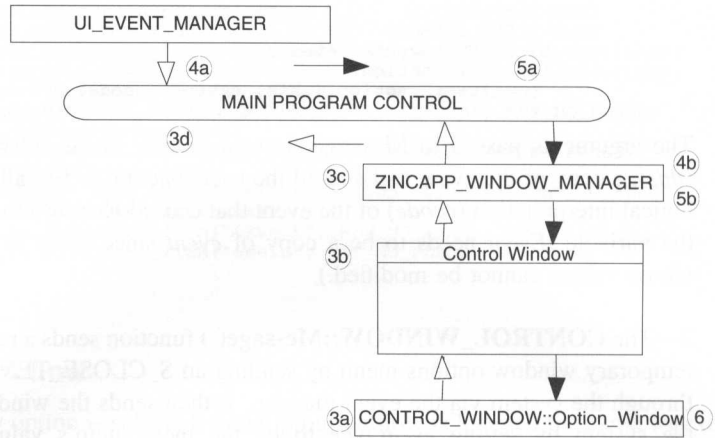
2—The **CONTROL_WINDOW::Message()** function sends a request to remove the temporary window options menu by sending an **S_CLOSE_TEMPORARY** message through the system via the event manger. It then sends the window request through the system by setting *event.type* to be the menu item's value (i.e., 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 **UI_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 window options 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, 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 **Option-Window()** 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 `Option_Window()` 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_COMBO_BOX_WINDOW:
            object = WindowComboBox();
            break;

        case MSG_DATE_WINDOW:
            object = WindowDate();
            break;

        case MSG_GENERIC_WINDOW:
            object = WindowGeneric();
            break;

        case MSG_ICON_WINDOW:
            object = WindowIcon();
            break;

        case MSG_LIST_WINDOW:
            object = WindowList();
            break;

        case MSG_MENU_WINDOW:
            object = WindowMenu();
            break;

        case MSG_NUMBER_WINDOW:
            object = WindowNumber();
            break;

        case MSG_STRING_WINDOW:
            object = WindowString();
            break;

        case MSG_TEXT_WINDOW:
            object = WindowText();
            break;

        case MSG_TIME_WINDOW:
            object = WindowTime();
            break;

        case MSG_BUTTON_WINDOW:
            object = WindowButton();
            break;

        case MSG_TOOL_BAR_WINDOW:
            object = WindowToolBar();
            break;
    }
}

```

```

    case MSG_MDI_WINDOW:
        object = WindowMDI();
        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`),
- system button (`UIW_SYSTEM_BUTTON`), and
- the title bar (`UIW_TITLE`).

In this function, the window is created by calling the constructors for each individual window objects. The operations performed in this function are equivalent to 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 an MDI child window as well as some minimized windows.

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

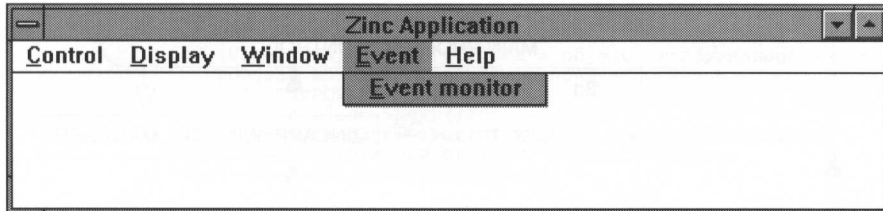
Text—This window shows a full-window implementation of a `UIW_TEXT` object and an associated vertical scroll bar.

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:



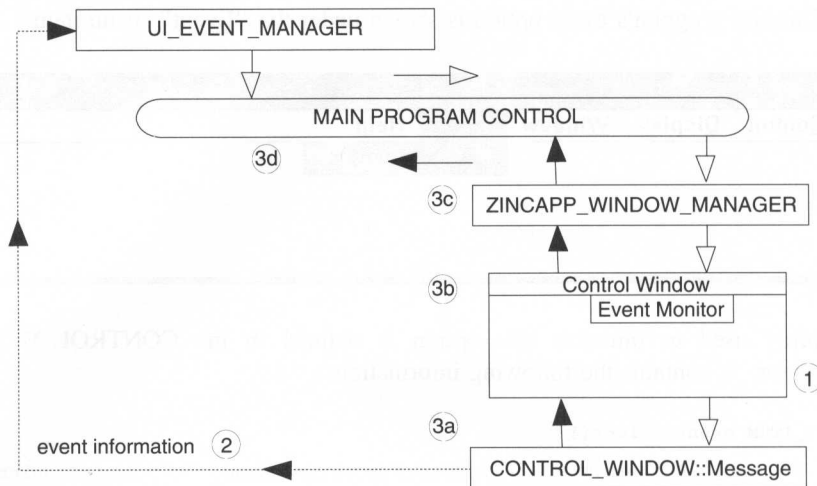
The array used to initialize this option is defined in the CONTROL_WINDOW constructor. It contains the following information:

```
UI_ITEM eventOptions[] =
{
    { MSG_EVENT_MONITOR,    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)
    + new UIW_PULL_DOWN_ITEM("&Control objects", WNF_NO_FLAGS,
      controlObjectItems)
    + new UIW_PULL_DOWN_ITEM("&Input objects", WNF_NO_FLAGS,
      inputObjectItems)
    + new UIW_PULL_DOWN_ITEM("&Selection objects", WNF_NO_FLAGS,
      selectionObjectItems)
  + 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 `OptionsEvent()` 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 = event;
        if (userFunction)
            (*userFunction)(this, tEvent, ccode);
}
  
```

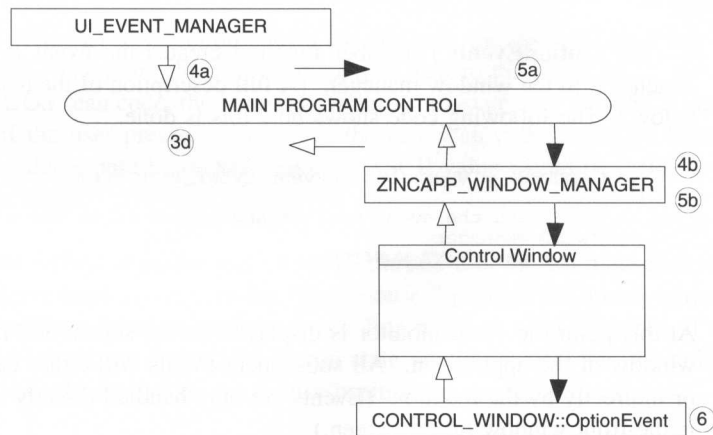
The arguments passed to `Message()` are a pointer to the selected event option (*this*), a copy of the event that caused the user function to be called (*tEvent*), and the logical interpretation (*ccode*) of the event that caused `Event()` to be called. (NOTE: the variable *tEvent* needs to be a copy 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 manger. 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 `UI_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 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 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.)

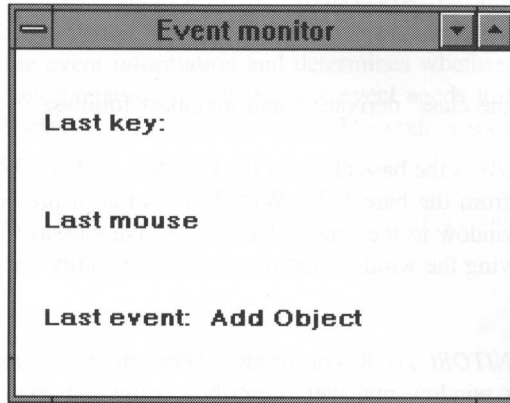
Monitoring library events

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.



This window has three sections:

Last key—The last key section shows the end-user the last keyboard key that has been pressed. There are three items displayed next to the “Last key:” prompt: the key’s raw DOS scan code, the current keyboard shift-state, and the ASCII value. For example, if the user pressed <Ctrl+F1>, the raw scan code would be 0x5E00, the shift-state value would be 0x0004, and the ASCII value would be 0 (the low 8 bits of the scan code).

Last mouse—The last mouse section shows the end-user the last mouse event. There are three items displayed next to the “Last mouse:” prompt: the mouse button status, its screen column position, and its screen line position. For example, if the user pressed the left-mouse button, the button state would be 1100 and the column and line values would depend on the mouse position on the screen.

Last event—The last event section shows the interpreted value of the last event. This can be any of the Zinc Interface Library 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 defined 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:
    UIW_STRING *keyboard[3];
};
```

```

    UI_EVENT kEvent;
    UIW_STRING *mouse[3];
    UI_EVENT mEvent;
    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* discusses 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 are 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, then 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 our window (if we are the 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 last three are described where the *keyboard*, *mouse* and *system* variables are described. 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 (tEvent->type == E_KEY)
    .
    .

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

    // 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). 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 (see the “Last mouse” description above). These variables work just like the keyboard variables *keyboard* and *kEvent* except 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 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;
    }
}

```

- *system* and *sEvent* contain information about the last interpreted event that was returned by the window object (see the “Last event” description above). These

variables work just like the mouse variables *mouse* and *mEvent* except 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" },
        { MSG_GENERIC_WINDOW, "Generic Window" },
        .
        .
        { 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 MSG_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:

- *public 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:

```

main()
{
    .
    .
    // Initialize the ZincApp window manager and add the control window.
    ZINCPP_WINDOW_MANAGER *windowManager =
        new ZINCAAPP_WINDOW_MANAGER(display, &eventManager,
        ZINCAAPP_WINDOW_MANAGER::ExitFunction);
    *windowManager
        + new CONTROL_WINDOW;
}

```

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

```

EVENT_TYPE ZINCAAPP_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)
        {
            UI_EVENT tEvent(event.type, 0xFFFF);
            tEvent.data = (void *)&event;
            object->Event(tEvent);
        }

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

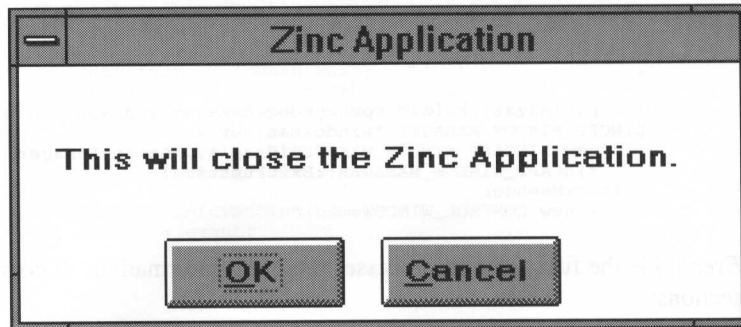
```

The first section calls `UI_WINDOW_MANAGER::Event` so 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` 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.

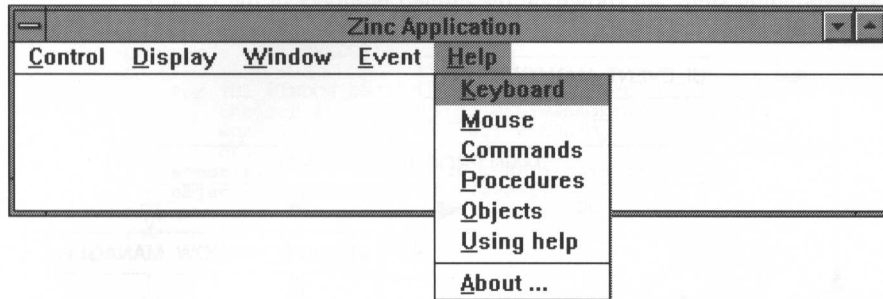
A picture of this window is shown below:



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:

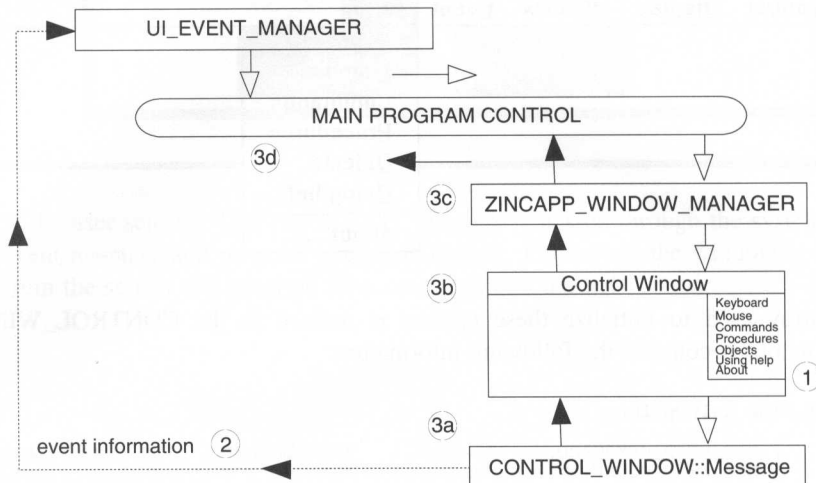


The array used to initialize these options is defined in the CONTROL_WINDOW constructor. It contains the following information:

```
UI_ITEM helpOptions[] =
{
    { MSG_HELP_KEYBOARD,    Message,    "&Keyboard",    MNIF_NO_FLAGS },
    { MSG_HELP_MOUSE,      Message,    "&Mouse",      MNIF_NO_FLAGS },
    { MSG_HELP_COMMANDS,   Message,    "&Commands",   MNIF_NO_FLAGS },
    { MSG_HELP_PROCEDURES, Message,    "&Procedures", MNIF_NO_FLAGS },
    { MSG_HELP_OBJECTS,    Message,    "&Objects",    MNIF_NO_FLAGS },
    { MSG_HELP_HELP,       Message,    "&Using help", MNIF_NO_FLAGS },
    { 0, 0, 0, 0 },
    { MSG_HELP_ZINCAPP,    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)
+ new UIW_PULL_DOWN_ITEM("&Control objects", WNF_NO_FLAGS,
controlObjectItems)
+ new UIW_PULL_DOWN_ITEM("&Input objects", WNF_NO_FLAGS,
inputObjectItems)
+ new UIW_PULL_DOWN_ITEM("&Selection objects", WNF_NO_FLAGS,
selectionObjectItems))
+ 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 **OptionsHelp()** 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 = event;
        if (userFunction)
            (*userFunction)(this, tEvent, ccode);
}
```

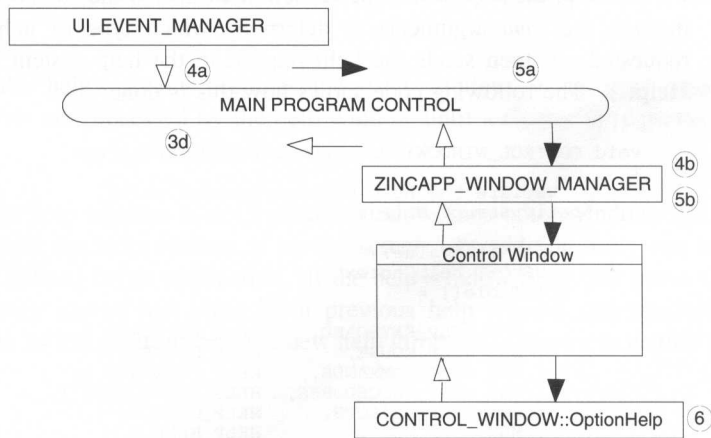
The arguments passed to **Message()** are a pointer to the selected help option (*this*) and a copy of the event that caused the user function to be called (*tEvent*). (**NOTE:** the variable *tEvent* needs to be a copy of *event* since *event* is a constant variable whose values cannot be modified.)

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 manger. 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) and then by sending another 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 UI_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 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);
}
```

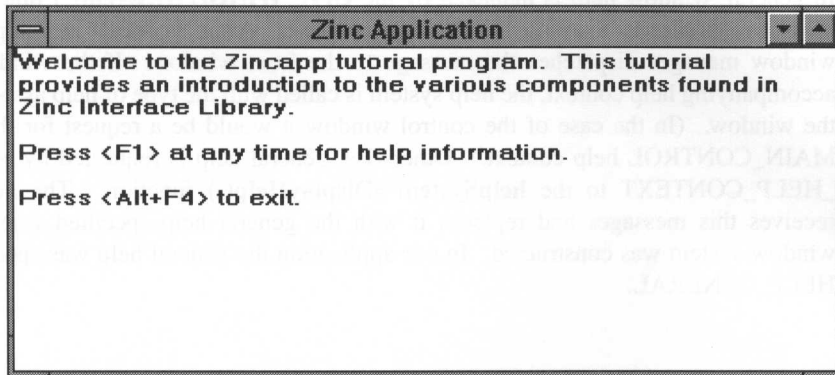
6—The **OptionHelp()** member function evaluates the item's value (passed down through the *item* argument) to determine which type of help context has been requested. It then sends the help request to the help system by calling **DisplayHelp()**. The following code shows how this is done:

```
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 `UIW_WINDOW::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 messages and replaces it with the general help specified when the help window system was constructed. In our application the general help was specified to be `HELP_GENERAL`.

```

main()
{
    .
    .
    // 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);
}

```

SECTION IV DERIVED CLASSES

SECTION IV
DERIVED CLASSES

... multiply by m ...
... times greater than the ...
... N1527 ...
... the amount ...
... view system ...
... HELP OFF ...

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 Interface Library
- the type of information needed to initialize the `UI_DEVICE` base class

The source code associated with this program is located in `\ZINCVTUTOR\MACRO`. It contains the following files:

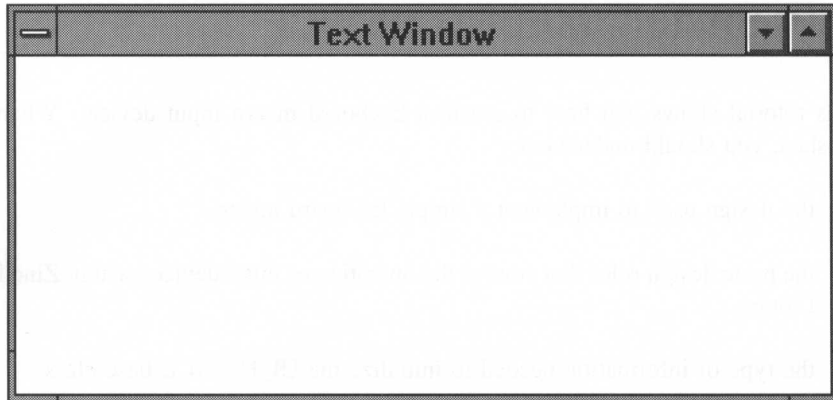
MACRO.CPP—This file contains the macro device member functions `MACRO_HANDLER::Event()` and `MACRO_HANDLER::Poll()`, as well as the main program loop (`main()` or `WinMain()`).

MACRO.HPP—This file contains the macro device class definition.

BORLAND.MAK, **MICROSOFT.MAK**, and **ZORTECH.MAK**—These are the makefiles associated with the macro program. You can compile the DOS version, **MACRO.EXE**, by typing `make -fborland.mak macro.exe` at the command line prompt. To make the Windows version, **WMACRO.EXE**, type `make -fborland.mak wmacro.exe` at the command line. (**NOTE:** When compiling with either Microsoft or Zortech, substitute the name of that compiler as the name of the make file on the command line.)

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 to typing normal text, 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 for DOS and Microsoft Windows environments:

```
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;
class MACRO_HANDLER : public UI_DEVICE
{
public:
    struct MACRO_PAIR
    {
        RAW_CODE rawCode;
        char *macro;
    };

    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 Interface Library 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 shown below:

```
MACRO_PAIR macroTable[] =
{
    { F5, "Macro #1." },
    { F6, "Macro #2." },
    { F7, "Macro #3." },
    { F8, "Macro #4." },
    { 0, NULL }
};
```

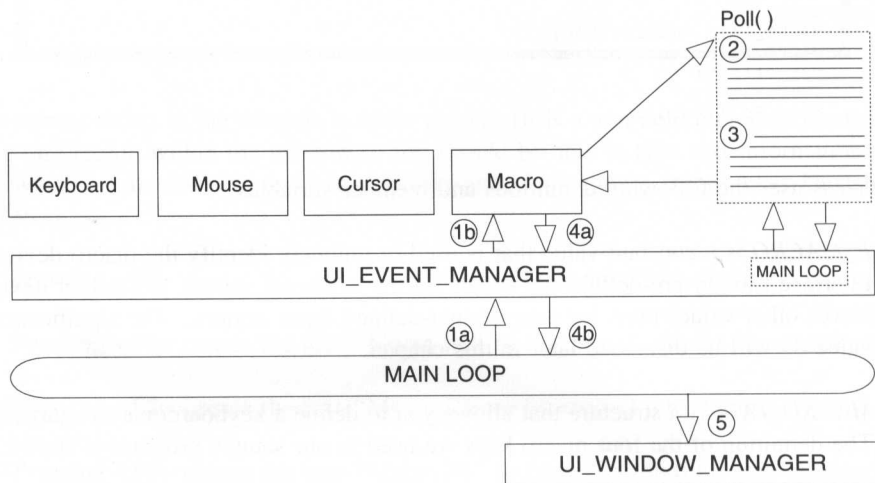
The entry { 0, NULL } is used as an end-of-array indicator. In addition, the use of F5, F6, F7 and F8 in the array above requires us to define a constant value called *USE_RAW_KEYS*. This definition allows us to have access to the raw DOS scan codes defined in *UI_EVT.HPP*.

- *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;
    int emptyQueue = eventManager->Get(event,
        Q_NO_POLL | Q_NO_BLOCK | Q_NO_DESTROY | Q_BEGIN);
```

```

// See if the event is a macro key. (step 2)
if (state == D_OFF && !emptyQueue && event.type == E_KEY)
{
    .
    .
}

// Put macro information into the input queue. (step 3)
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 `eventManager->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 just want to “check” the input queue to see if something is available. Since `Q_NO_BLOCK` is set, the status of the input queue will be returned by the event manager’s function. (The return value is 0 if there is currently input information in the queue, or it is negative if there is no information.)

Next, we do not want to destroy the contents of the queue since we are only looking for special keyboard values. 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 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 keys. 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);
}

```

```

// See if the event is a macro key. (step 2)
if (state == D_OFF && !emptyQueue && event.type == E_KEY)
{
    for (int i = 0; macroTable[i].rawCode; 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 3)
if (state == D_ON && emptyQueue)
{
    .
    .
}
}

```

3—The third 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);

    // See if the event is a macro key. (step 2)
    if (state == D_OFF && !emptyQueue && event.type == E_KEY)
    {
        .
        .
    }
}

```

```

// Put macro information into the input queue.                (step 3)
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;
    }
}
}
}

```

4—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.

5—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 event manager's list of devices. Input devices are arranged in the device list in ascending *type* order. Thus, the order of our 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 we don'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 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_DEVICE::Poll()** function was described in the conceptual operation part of this chapter. In general, **Poll()** functions should be used:

1—to feed information to or get information from the event manager’s input queue. The keyboard, mouse, and Microsoft Windows message devices all have poll routines that feed information into the input queue.

2—when an object needs to be called 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_DEVICE::Event()` function is defined as an in-line stub.

```
class MACRO_HANDLER : public UI_DEVICE
{
public:
    EVENT_TYPE int Event(const UI_EVENT &event);
```

This routine must be declared by the macro device since the base `UI_DEVICE` class declares it 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 allows us to not disable 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 `UI_WINDOW_OBJECT::pasteBuffer` and `UI_WINDOW_OBJECT::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. The class variable `macroTable` would need to be modified to

support the addition and deletion of macro pairs, but it shouldn't be too hard to implement.

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 Interface Library and the way in which input devices operate within the library.

...the first step is to create a new object of the class `Zinc`...

...the second step is to call the `init` method...

...the third step is to call the `start` method...

...the fourth step is to call the `stop` method...

...the fifth step is to call the `destroy` method...

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 Interface Library
- how to derive a new window object from the `UI_WINDOW_OBJECT` base class
- how to implement a new window object in Microsoft Windows.

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

HELPBAR.CPP—This file contains the static functions `SetHelp()` and `ActionFunction()` as well as the main program loop (`main()` or `WinMain()`).

HLPBAR.CPP—This file contains the `HELP_BAR` class source code.

HLPBAR.HPP—This file contains the `HELP_BAR` class definition.

BORLAND.MAK, MICROSOFT.MAK, and ZORTECH.MAK—These are the makefiles associated with the help bar program. You can compile the DOS version, **HELPBAR.EXE**, by typing `make -fborland.mak helpbar.exe` at the command line prompt. To make the Windows version, **WHELPBAR.EXE**, type `make -fborland.mak whelpbar.exe` at the command line. (**NOTE:** When compiling with either Microsoft or Zortech, substitute the name of that compiler as the name of the make file on the command line.)

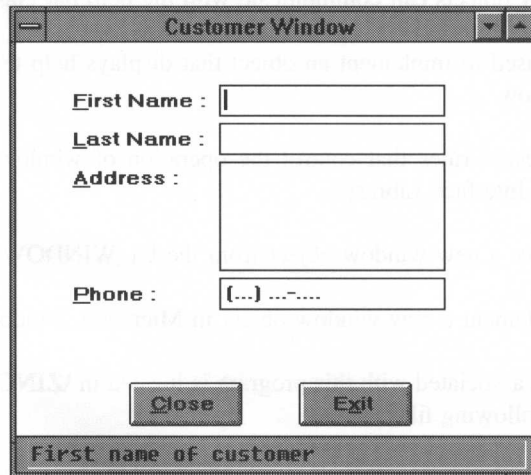
WHELPBAR.DEF—This is the definition file for the help bar program. It is used to set constants associated with the executable program (Windows only).

WHELPBAR.RC—This is the resource file for the help bar program. It is used to add resources to the executable program (Windows only).

Program execution

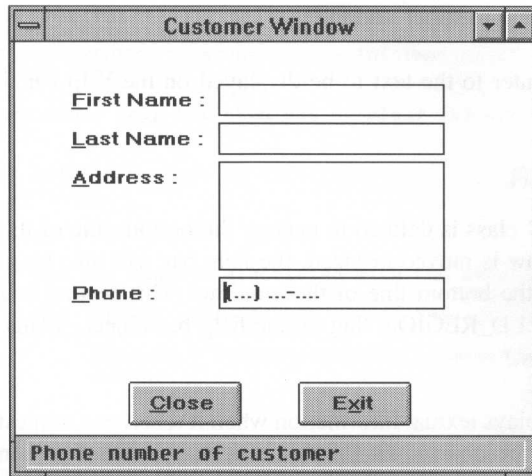
The operation of help bar objects can be seen by compiling and running the application **HELPBAR.EXE** in DOS or **WHELPBAR.EXE** in Windows.

Two copies of the following window should appear on the screen:



The image shows a standard Windows-style dialog box titled "Customer Window". It contains four input fields: "First Name" (a single-line text box), "Last Name" (a single-line text box), "Address" (a multi-line text area), and "Phone" (a single-line text box with a mask "[...] ..-...."). Below the input fields are two buttons labeled "Close" and "Exit". At the bottom of the dialog box, there is a help bar containing the text "First name of customer".

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 Number" 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 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;
};
```

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:

```
// 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.
static 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. Although *helpBar* is a pointer to a `UI_WINDOW_OBJECT`, the **Information()** function associated with the help bar will be called since this function was declared as virtual.

Event function—DOS

A DOS version of the `HELP_BAR::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 mode. Since some of the low-level messages are environment-specific, one event function is created for DOS and another for Windows. Other than this exception, the source code for the help bar is portable across DOS and Windows. 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.

```
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_REGION region = true;
if (FlagSet(woFlags, WOF_BORDER))
    DrawBorder(screenID, region, FALSE, ccode);
```



```

UI_PALETTE *palette = LogicalPalette(ccode, ID_BUTTON);
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_REDISELAY;

```

Event function—Windows

A Windows version of the **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, one **Event()** is created for DOS and another for Windows. Other than this exception, the source code for the help bar is portable across DOS and Windows. 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 this message 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 wParam,
WORD lParam, LONG lParam)
{
    HELP_BAR *object = (HELP_BAR *)GetWindowLong(hWnd, _helpbarOffset);
    return (object->Event(UI_EVENT(E_MSWINDOWS, hWnd, wParam, lParam,
    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).

WM_PAINT—This is a Windows message similar to the Zinc messages **S_DISPLAY_ACTIVE** and **S_DISPLAY_INACTIVE**. The graphical help bar field is similar, in appearance, to a depressed button. The following code shows how the help bar is drawn. (**Note:** The majority of this code uses Windows' calls.)

```
if ((cocode == S_REDISPLAY && screenID) ||
    (event.type == E_MSWINDOWS && message == WM_PAINT))
{
    if (cocode == 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);
EndPoint(screenID, &ps);
}

```

Information function

Each derived window object should have an associated virtual **Information()** function. The purpose of the **Information()** function is to handle information requests. **HELP_BAR::Information()** function handles two requests:

GET_TEXT—This request returns the text information associated with the help bar. If this request is used, *data* (passed as a value parameter) must be a pointer to a programmer-defined string large enough to contain the help bar text information.

```

case GET_TEXT:
    if (!data)
        return (text);
    *(char **)data = text;
    break;

```

SET_TEXT—This request sets the text information associated with the help bar. If this request is used, *data* (passed as a value parameter) must be a pointer to a programmer-defined string containing the help bar text information. When the text is copied into the help bar's *text*, a **S_REDISPLAY** message is sent to the help bar to display the new text.

```

case SET_TEXT:
    if (text)
        delete text;
    text = data ? strdup((char *)data) : NULL;
    HELP_BAR::Event(UI_EVENT(S_REDISPLAY));
    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 window objects within Zinc Interface Library.
- the type of information needed to initialize the base `UIW_WINDOW` base classes.

The source code associated with this program is located in the `\ZINCVTUTOR\VLIST`. It contains the following files:

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

```
VIRTUAL_LIST::VIRTUAL_LIST( )  
VIRTUAL_LIST::~VIRTUAL_LIST( )  
VIRTUAL_ELEMENT::Event( )  
VIRTUAL_LIST::Event( )  
VIRTUAL_LIST::LoadNext( )  
VIRTUAL_LIST::Seek( )
```

In addition, this file contains the main program loop (**main()** or **WinMain()**).

VLIST.DAT—This file contains 100 records that are dynamically read from disk when needed by the virtual list.

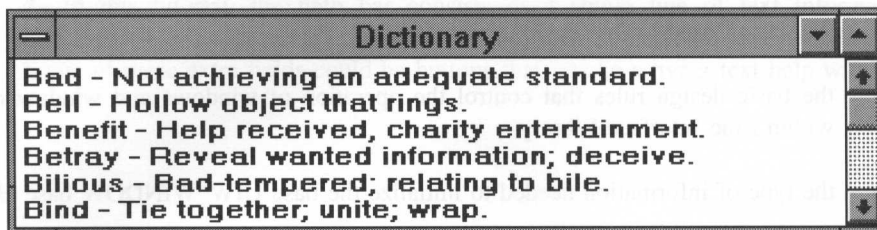
VLIST.HPP—This file contains the virtual list and the element class definitions.

BORLAND.MAK, **MICROSOFT.MAK**, and **ZORTECH.MAK**—These are the makefiles associated with the virtual list program. You can compile the DOS version, **VLIST.EXE**, by typing `make -fborland.mak vlist.exe` at the command line prompt. To make the Windows version, **WVLIST.EXE**, type `make -fborland.mak wvlist.exe` at the command line. (**NOTE:** When compiling with either Microsoft or Zortech, substitute the name of that compiler as the name of the make file on the command line.)

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 current object in the window is the virtual list. 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:

<u>Action</u>	<u>Key</u>	<u>Description</u>
<i>First element</i>	<Ctrl+Home>	Moves to the first database element.
<i>Last element</i>	<Ctrl+End>	Moves to the last database element.
<i>Previous element</i>	<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.
<i>Next element</i>	<Enter> <Gray Enter> <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.

<i>Page-up</i>	<PgUp> <Gray PgUp>	Moves up one page in the database.
<i>Page-down</i>	<PgDn> <Gray PgDn>	Moves down one page in the database.

In addition, the left mouse button can be used to select an object, or to scroll the window's list information.

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

Class definitions

The virtual list you see in the 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 within the window. 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:

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

// Create the virtual list then attach it to the window manager.
windowManager
    + &(*UIW_WINDOW::Generic(5, 5, 30, 12, "Virtual List")
      + &(*new VIRTUAL_LIST("vlist", 12)
        + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_VERTICAL,
          WOF_NON_FIELD_REGION));
```

The definition for the `VIRTUAL_ELEMENT` class is given below:

```
class EXPORT VIRTUAL_ELEMENT : public UIW_STRING
{
    void DataSet(int column, int line, int width, int height);
    void DataSet(const VIRTUAL_ELEMENT *element);
    virtual int Event(const UI_EVENT &event);
    VIRTUAL_ELEMENT *Next(void);
    VIRTUAL_ELEMENT *Previous(void);

private:
    friend class VIRTUAL_LIST;

    int recordNumber;

    VIRTUAL_ELEMENT(int left, int top, int width, int 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 100 records, *recordNumber* for the last database record would be 99.

The VIRTUAL_LIST class definition is:

```
class EXPORT VIRTUAL_LIST : public UIW_VT_LIST
{
public:
    VIRTUAL_LIST(const char *fileName, int aRecordLength);
    ~VIRTUAL_LIST(void);
    VIRTUAL_ELEMENT *Current(void);
    virtual int Event(const UI_EVENT &event);
    VIRTUAL_ELEMENT *First(void);
    VIRTUAL_ELEMENT *Last(void);
    void LoadFirst(VIRTUAL_ELEMENT *element);
    void LoadLast(VIRTUAL_ELEMENT *element);
    void LoadNext(VIRTUAL_ELEMENT *element);
    void LoadPrevious(VIRTUAL_ELEMENT *element);
    void Seek(int recordNumber);

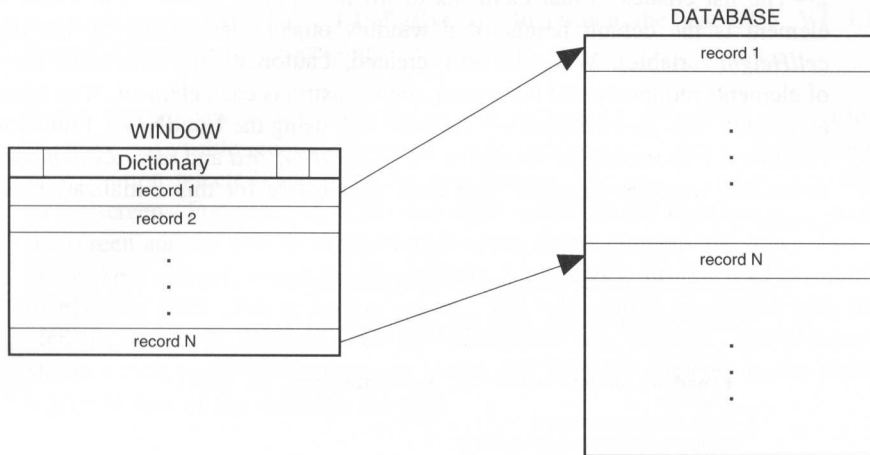
private:
    FILE *file;
    const int recordLength;
    int currentRecord;
    int lastRecord;
};
```

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.
- *currentRecord* tells at what record the file pointer is positioned. For instance, if a read operation were performed, without changing the position of the file pointer, *currentRecord* would be the record that would be read from disk.
- *lastRecord* is the number of the last 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, *lastRecord* is 99 for our database, since we have 100 records.

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_VT_LIST base class and occupies all space within the parent window's border. This is accomplished by setting the WOF_NON_FIELD_REGION flag. Thus, anytime the window is sized, the virtual list occupies all of the space inside the border. The code below shows how the list is initialized and how it inherits the size capability from the base UIW_VT_LIST class.

```

VIRTUAL_LIST::VIRTUAL_LIST(const char *fileName, int aRecordLength) :
    UIW_VT_LIST(0, 0, 0, 0, WOF_NON_FIELD_REGION), currentRecord(0),
    recordLength(aRecordLength)
{
    .
    .
}
EVENT_TYPE VIRTUAL_LIST::Event(const UI_EVENT &event)
{
    // Process virtual list information.
    EVENT_TYPE ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_MATRIX);
    UI_EVENT tEvent = event;
    VIRTUAL_ELEMENT *element = Current();
    VIRTUAL_ELEMENT *tElement;
    switch (ccode)
    {
    case S_CREATE:
    case S_SIZE:
        // Get the original size.
        UIW_VT_LIST::Event(event);
    .
    .
}

```

2—The list creates virtual elements to fill its window space. The height of each element is the default height of a window object (determined by the *display->cellHeight* variable). 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 **LoadNext()** function. This function is responsible for setting the *UIW_STRING::text* and *recordNumber* variables associated with the element. The code responsible for this initialization is shown below:

```

void VIRTUAL_LIST::LoadNext(VIRTUAL_ELEMENT *element)
{
    char *text = new char(recordLength+1);

    // Load the record.
    fread(&element->text[1], recordLength, 1, file);
    text[0] = ' ';
    text[recordLength - 1] = '\0';
    UIW_STRING::DataSet(text);
    element->recordNumber = currentRecord++;
}

EVENT_TYPE VIRTUAL_LIST::Event(const UI_EVENT &event)
{
    // Process virtual list information.
    EVENT_TYPE ccode = UIW_VT_LIST::LogicalEvent(event, ID_VT_LIST);
    UI_EVENT tEvent = event;
    VIRTUAL_ELEMENT *element = Current();
    VIRTUAL_ELEMENT *tElement;
    switch (ccode)
    {
        case S_CREATE:
        case S_SIZE:
            // Get the original size.
            UIW_VT_LIST::Event(event);

            // Calculate the number of elements that will fit on the screen.
            int line = 0;
            int width = true.right - true.left + 1;
            int maxElements = (true.bottom - true.top) /
                display->cellHeight;

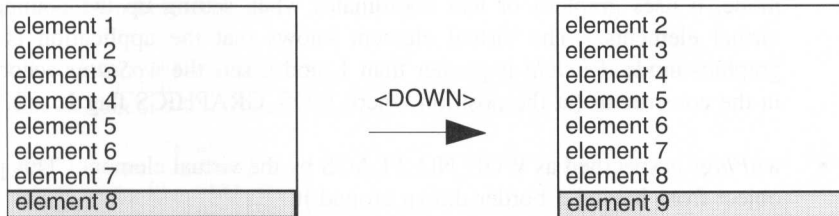
            element = First();
            for (int i = 0; i < maxElements; i++)
            {
                if (!element)
                {
                    element = new VIRTUAL_ELEMENT(0, line, width,
                        display->cellHeight);
                    if (i == 0)
                        Seek(0);
                    LoadNext(element);
                    UI_LIST::Add(element);
                }
                .
                .
            }
    }
}

```

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_VT_LIST::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
{
private:
    friend class VIRTUAL_LIST;

    VIRTUAL_ELEMENT(int left, int top, int width, int height);
    .
    .
}
```

The class constructor initializes information as follows:

- *left*, *top*, *width*, and *height* give the height 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 on the screen.
int line = 0;
int width = true.right - true.left + 1;
int maxElements = (true.bottom - true.top) / display->cellHeight;
.
.
.
element = new VIRTUAL_ELEMENT(0, line, width, display->cellHeight);
.
.
.
```

Since the virtual list has already been set up to display in either graphics or text mode, it uses graphics or text coordinates when setting up the boundaries of the virtual elements. The virtual element knows that the application is running in graphics mode if *height* is greater than 1, and it sets the *woStatus* accordingly later in the constructor (at the position where `WOS_GRAPHICS` flag is set).

- *woFlags* is specified as `WOF_NO_FLAGS` by the virtual element. This prevents the object from having a border drawn around it.
- *woAdvancedFlags* is specified as `WOAF_NO_FLAGS` by the virtual element. This tells the controlling window that the object occupies normal space within the parent window.

All other member variables are set by the base `UIW_STRING` constructor and by the parent window (`VIRTUAL_LIST`) when the window is attached to the window manager.

The two overloaded `DataSet()` functions are used by `VIRTUAL_LIST` to set the information contained within the list. The first overloaded function takes four integer arguments. This overloaded function is used when a new virtual list element is created. The four arguments give the relative position of the object within the virtual list. The second overloaded function resets the `UIW_STRING::text` and *recordNumber* information associated with the element. This function is used by the virtual list to shift record information from one position on the screen to another (in the scrolling operations). Both of these functions are in-line, and both are presented below:

```

void DataSet(int column, int line, int width, int height)
{ relative.left = column, relative.right = column + width - 1,
  relative.top = line, relative.bottom = line + height - 1;
  if (height > 1) woStatus |= WOS_GRAPHICS;
  else woStatus &= ~WOS_GRAPHICS; }
void DataSet(const VIRTUAL_ELEMENT *element)
{ UIW_STRING::DataSet(element->text); recordNumber =
  element->recordNumber; }

```

VIRTUAL_LIST

The virtual list class is derived from the base class UIW_VT_LIST. 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 aRecordLength) :
  UIW_VT_LIST(0, 0, 0, 0, WOF_NON_FIELD_REGION),
  currentRecord(0), recordLength(aRecordLength)
{
  // Open the database.
  file = fopen(fileName, "rb");
  fseek(file, 0L, SEEK_END);
  lastRecord = (int)(ftell(file) / recordLength) - 1;
  fseek(file, 0L, SEEK_SET);
}

```

Base class initialization

The base UIW_VT_LIST 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. The boundary arguments are all zero since we are setting the WOF_NON_FIELD_REGION flag. This flag ensures that we will get the remaining space within the parent window's border.

In addition to the boundary information, the base UIW_VT_LIST 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_VT_LIST : public UIW_WINDOW
{
public:

```

```

UIW_VT_LIST(int left, int top, int width, int height,
int (*compareFunction)(void *element1, void *element2) = NULL,
WNF_FLAGS wnFlags = WNF_NO_WRAP, WOF_FLAGS woFlags = WOF_BORDER,
WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS);
.
.
};

UIW_VT_LIST::UIW_VT_LIST(int left, int top, int width, int height,
int (*compareFunction)(void *element1, void *element2),
WNF_FLAGS _wnFlags, WOF_FLAGS _woFlags, WOAF_FLAGS _woAdvancedFlags)
: UIW_WINDOW(left, top, width, height, _woFlags, _woAdvancedFlags)
{
.
.
}

```

- the *UIW_WINDOW* class is initialized to contain no window objects. (In our class the objects will be *VIRTUAL_ELEMENT* objects.)

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 open 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.
- *currentRecord* is set to 0, since we will begin reading from the front of the file.
- *lastRecord* is determined by computing the end-of-file position, then by dividing that number by the size of each record. Then, 1 is subtracted from the number since we begin record counts at 0.

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. When a subsequent display message is passed to the virtual list's **Event()** function, the message is passed to the **UIW_WINDOW::Event()** member function for processing. This causes the list element's 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. This element is presented to the screen by calling the **element->Event()** function with the message **S_REDISPLAY**.

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. This element is presented to the screen by calling the **element->Event()** function with the message **S_REDISPLAY**.

L_PGUP, **L_PGDN**, **L_FIRST**, and **L_LAST** 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::LoadNext()** function is the only function that actually performs read operations from disk. All other load functions first call the **VIRTUAL_LIST::Seek()** function, then call **VIRTUAL_LIST::LoadNext()**.

The Seek function

The **VIRTUAL_LIST::Seek()** function allows us to move to a different position in the database. It contains some optimization so that we don't have to seek from the beginning of the file every time we want to read information from disk.

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 **LoadNext()** 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.

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 Interface Library.

CHAPTER 16 – CUSTOMIZED DISPLAYS

This tutorial tells you 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. 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 Interface Library.
- the type of information needed to initialize the base `UI_DISPLAY` class.

The source code associated with this program is located in the `\ZINCVTUTOR\DISPLAY`. It contains the following files:

BGIDSP.CPP—This file contains the BGI class constructor, destructor, and associated display member functions.

TEST.CPP—This file contains a graphics test program.

BORLAND.MAK—These are the makefiles associated with the display program. You can compile **TEST.EXE**, by typing `make -fborland.mak test.exe` at the command line prompt.

NOTE: The `UI_BGI_DISPLAY` class requires the use of **GRAPHICS.LIB** and the associated BGI file (e.g., **EGAVGA.BGI**, **CGA.BGI**, **HERC.BGI**). These files are not provided on any of the Zinc Interface Library distribution diskettes—they are provided with the Borland compiler. If you do not have the Borland compiler, it is still recommended you read this tutorial so that you understand the theory and implementation details of Zinc Interface Library 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 Interface Library is to call **display->Rectangle()**. The actual details of drawing a rectangle are left to the device dependent **Rectangle()** function. We do not need to know whether we are running under Borland's BGI graphics, Zortech's Flash Graphics, Microsoft's MSC graphics, or even in text mode.

There are three key aspects to the implementation of display classes. The first is the abstract definition of the base UI_DISPLAY 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 as pure virtual by the base UI_DISPLAY class:

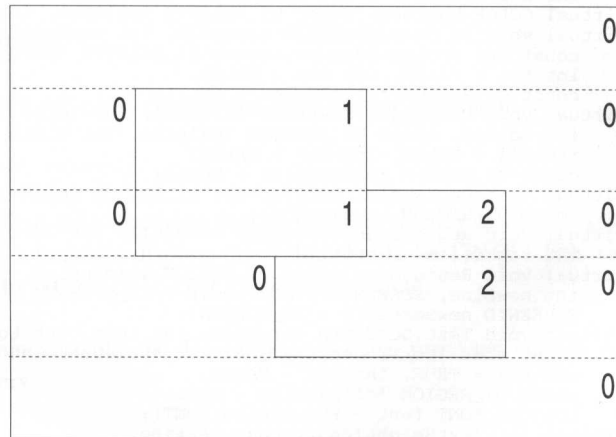
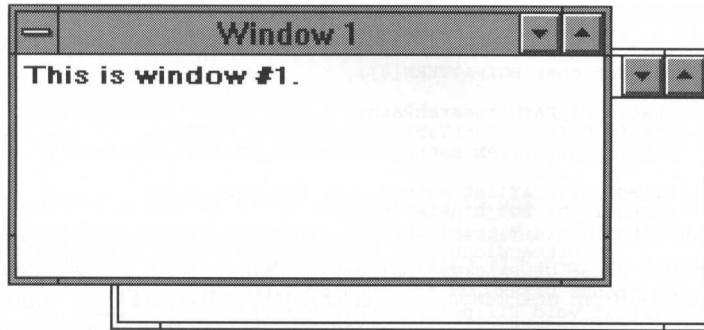
```
class EXPORT UI_DISPLAY
{
public:
    virtual void Rectangle(SCREENID screenID, int left, int top,
        int right, int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL) = 0;
```

This declaration requires each derived display class to have an implementation for the **Rectangle()** function.

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.

<u>Display mode</u>	<u>Right</u>	<u>Bottom</u>
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. For example, if the following two windows were attached to the screen, the display would contain several rectangular regions with different identifications:



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 Interface Library, 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);
    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 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;

    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 Interface Library 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.

#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 base `UI_DISPLAY` class 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 `TRUE`. This value is used by the base display to set the `UI_DISPLAY::isText` variable.

In addition to this variable the `UI_DISPLAY` class provides default settings for the following members:

- *installed* is a flag that tells whether the display has been installed. This member is set to `FALSE` by the base `UI_DISPLAY` class. You should set this variable to be `TRUE` if the graphics display installs correctly.
- *isMono* is a flag that tells whether the display is operating in monochrome mode.
- *columns* and *lines* are both set to 0. They must be set when you determine the type of display available.
- *preSpace* denotes the size (in pixels) of the white space between the top border of a string field and the tallest character.
- *postSpace* denotes the size (in pixels) of the white space between the bottom border of a string field and the lowest character.

- *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.)
- *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.)
- *backgroundPalette* is a pointer to the background color palette.
- *xorPalette* is a pointer to the XOR color palette.
- *colorMap* is a pointer to the normal color palette.

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 = registerbgifont(SmallFont);
    if (BGIFont.font >= 0)
    {
        BGIFont.charSize = 0;
        BGIFont.maxWidth = 10;
        BGIFont.maxHeight = 11;
        UI_BGI_DISPLAY::fontTable[FNT_SMALL_FONT] = BGIFont;
    }
    BGIFont.font = registerbgifont(DialogFont);
    if (BGIFont.font >= 0)
    {
        BGIFont.charSize = 0;
        BGIFont.maxWidth = 11;
        BGIFont.maxHeight = 11;
        UI_BGI_DISPLAY::fontTable[FNT_DIALOG_FONT] = BGIFont;
    }
    BGIFont.font = registerbgifont(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;
lines = getmaxy() + 1;
maxColors = getmaxcolor() + 1;

// Fill the screen according to the specified palette.
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;
}

```

(Step 4)

(Step 5)

The main steps in this initialization are:

- 1—The first step is to register the system, dialog, and small fonts that were linked in. 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 **ZIL.LIB**.)
- 2—The second step requires that we find out what type of display can be created. In the Borland graphics library this is done by calling **detectgraph()**. The *driver* and *mode* arguments allow the programmer to override this default detection.
- 3—The third step required to set up the Borland graphics package (not required by Zortech's graphics library) is that we find the associated graphics driver. The **UI_PATH** class object is used as a searching area where the driver may be found. 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 searches 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. All we need to do is call `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; dRegion = dRegion->Next())
        if (xor || screenID == ID_DIRECT ||
            (screenID == dRegion->screenID &&
             dRegion->region.Overlap(region, tRegion)))
        {
            if (xor || 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 (xor || 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 object should be clipped. The clip region is useful within Zinc Interface Library 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* variable.

2—In the conceptual discussion of display classes we saw how each display keeps track of reserved areas on the screen. The second step of each paint routine is used to determine what 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, no graphics libraries 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.

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.

5—In order to make the screen drawing faster, the **VirtualGet()** and **VirtualPut()** functions have been added. They allow an entire section of the display to be updated at once and not line-by-line. (See “UI_BGI_DISPLAY” in the *Programmer’s Reference* for more details.)

Confused? It is really quite simple in theory, it’s just the implementation that gets messy. You will find each graphics library has its own way of doing things and that you need to understand the operation of the whole system, even though you are at a very low-level in the library. The main things to remember are to be very consistent in what you do, make sure that you set up the clip regions properly, and be sure that you understand where you really want to paint the image.

Information Member Functions

There are two information functions associated with the display. **TextHeight()** is used to get the maximum height of the current font. If the current font, *logicalFont*, is 0, the Borland **textheight()** function is called. **TextWidth()** is used to get the maximum width of the current font. Its operation is similar to that of **TextHeight()**.

```
int UI_BGI_DISPLAY::TextHeight(const char *string, SCREENID,
    LOGICAL_FONT logicalFont)
{
    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);
    int length = textwidth((char *)string);
    if (strchr(string, '&'))
        length -= textwidth("&");
    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 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 Interface Library. If you want to support additional displays, use this tutorial as a template for your implementation.

SECTION V PERSISTENT OBJECTS

SECTION V
PERSISTENT OBJECTS

CHAPTER 17 – 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 Interface Library.

Webster's New Universal Unabridged Dictionary has the following definitions of persistence:

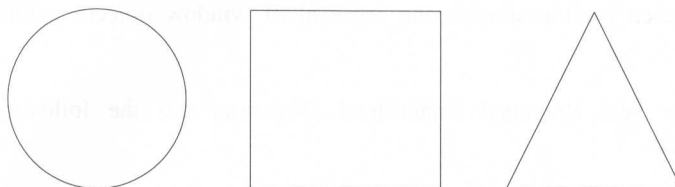
1. the act of persisting; stubborn or enduring continuance, as in a chosen course or purpose.
2. a persistent or lasting quality; resoluteness, tenacity.
3. continuous existence; endurance, as of a headache.
4. the continuance of an effect after the cause which first gave rise to it is removed; as persistence 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
{
    int column, line, radius;
};

struct RECTANGLE
{
    int left, top, right, bottom;
};

struct TRIANGLE
{
    int 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** also contains functions that support Microsoft C/C++, Microsoft Windows, and Zortech's Flash Graphics.

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
{
    int column, line, radius;
};

struct RECTANGLE
{
    int left, top, right, bottom;
};

struct TRIANGLE
{
    int 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, then declaring compiler specific instances of the class.

The code implementation of these concepts is shown below. (The actual code is contained in the file **PERSIST4.CPP**. 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(int _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(int _column, int _line, int _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:
    int column, line, radius;
};
```

```

class RECTANGLE : public GRAPHIC_OBJECT
{
public:
    RECTANGLE(int _left, int _top, int _right, int _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:
    int left, top, right, bottom;
};

class TRIANGLE : public GRAPHIC_OBJECT
{
public:
    TRIANGLE(int column1, int line1, int column2, int line2,
             int column3, int 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:
    int 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 disjoint, 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. 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 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(int _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(int column1, int line1, int column2, int line2,
             int column3, int 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:
    int 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, 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** and the Windows version is contained in **PERSIST8.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 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.

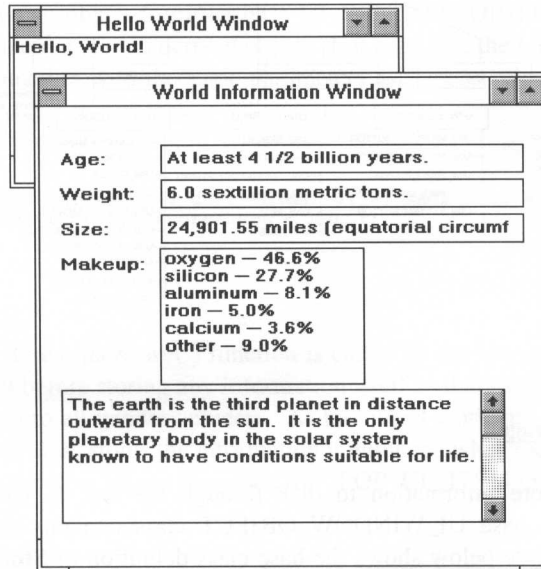
This implementation gives us the total abstraction we wanted at a relatively small inconvenience. Let's review the steps required to implement the simple persistence for graphic objects:

- 1—We defined an abstract `GRAPHIC_OBJECT` class that contains 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 18 – ZINC WINDOW OBJECTS

The previous tutorial should give you a good introduction of how simple persistent objects are implemented. Zinc Interface Library retrieves window objects created by the interactive design tool. For example, the “Hello, World!” tutorial loaded the following two windows from disk:



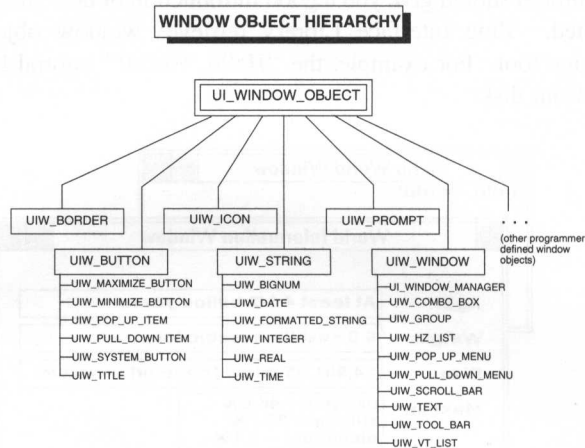
The retrieval of these two objects from disk required only two lines of constructor code. This code is shown below:

```
.  
. .  
// Add two windows to the window manager.  
*windowManager  
+ new UIW_WINDOW("HELLO~HELLO_WORLD_WINDOW")  
+ new UIW_WINDOW("HELLO~WORLD_INFORMATION_WINDOW");
```

Implementation details

The basic design of persistent objects in Zinc Interface Library is centered around four 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 Interface Library:



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, then 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_BUTTON` class object defines this retrieve capability:

```
class UIW_BUTTON : public UI_WINDOW_OBJECT
{
public:
    UIW_BUTTON(const char *name, UI_STORAGE *file = NULL,
               UI_STORAGE_OBJECT *object);
    .
    .
}
```

The inheritance code works in a similar manner to that used by the store operation. For example, here is the inheritance code that loads the `UIW_POP_UP_ITEM` class:

```
UIW_POP_UP_ITEM::UIW_POP_UP_ITEM(const char *name, UI_STORAGE *file,
                                  UI_STORAGE_OBJECT *object): UIW_BUTTON(name, file, object)
{
    .
    .
}

UIW_BUTTON::UIW_BUTTON(const char *name, UI_STORAGE *directory,
                       UI_STORAGE_OBJECT *file): UI_WINDOW_OBJECT(name, directory,
                           file), btStatus(BTS_NO_STATUS), userFunction(NULL), getString(NULL),
                           value(0)
{
    // Read the button information.
    .
    .
}

UI_WINDOW_OBJECT::UI_WINDOW_OBJECT(const char *name, UI_STORAGE *object,
                                     UI_STORAGE_OBJECT *file)
{
    // Read the base object information.
    .
    .
}
```

In this example, the pop-up item first calls the button class, then the button class calls the base class file constructor. The benefit of overloading the constructor of classes, rather than using a “new” and subsequent “load” operation, is that it only requires one call. The constructor can initialize its information based on the disk information, rather than doubling the effort by initializing variables that will later be over-written.

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 to 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 Interface Library 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 **.EXE** 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 VI APPENDIXES

SECTION V
APPENDICES

APPENDIX A – COMPILER CONSIDERATIONS

This appendix explains the initial configuration you must implement in order to compile your applications with Zinc Interface Library.

Borland

Integrated Development Environment (IDE)

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, turn **Word alignment** off, and turn **Unsigned characters** off.
- 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. You must include both your source files and the proper Zinc library.

To compile applications for Windows in the IDE, the following options must be selected within the IDE:

- Select **Options\Application**, then select **Window App**.
- Select **Options\Compiler\Code Generation**, then select the **Large** model.

NOTE: Although you can compile from within the IDE, you must be in Windows in order to actually run a Windows application.

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
```

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 Interface Library:

```
-I.;C:\ZINC\INCLUDE;C:\BORLANDC\INCLUDE  
-L.;C:\ZINC\LIB;C:\BORLANDC\LIB
```

- The following line should also be changed in **TLINK.CFG** to reflect the path of the compiler and Zinc Interface Library:

```
-L.;C:\ZINC\LIB;C:\BORLANDC\LIB
```

- In order to compile Microsoft Windows applications, you must be using Microsoft Windows Version 3.X and Borland C++ Version 3.0 or later. All of the options specified in the tutorial Borland makefiles must be used. In addition, the **-WE** compiler option and **/Twe** link option, which compiles the application as a Windows executable program, must be included.
- The appropriate Zinc **.LIB** files—either **ZIL.LIB** for DOS or **ZILW.LIB** for Windows—should be linked in.

Here is a sample “generic” makefile:

```
# Makefile for GENERIC example program  
# Uses Zinc Interface Library Version 3.00 and Borland C++ 3.00  
  
# 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;C:\BORLANDC\LIB  
# and your TLINK.CFG file to include the Zinc paths, e.g.:  
# -L.;C:\ZINC\LIB;C:\BORLANDC\LIB  
  
# make -fborland.mak generic.exe      (makes the generic example for DOS)  
# make -fborland.mak wgeneric.exe    (makes the generic example for  
#                                   Windows)  
  
## Compiler and linker: (Add -v to CPP_OPTS and /v to LINK_OPTS for debug.)  
CPP=bcc  
LINK=tlink  
CPP_OPTS=-c -ml -O1 -w  
LINK_OPTS=/c /x  
WCPP_OPTS=-c -ml -O1 -WE -w  
WLINK_OPTS=/c /C /Twe /x  
  
## Libraries  
C_OBJS=c01  
C_LIBS=zil graphics emu math1 cl  
WC_OBJS=c0wl  
WC_LIBS=zilw mathwl import cwl
```

```

.cpp.obj:
$(CPP) $(CPP_OPTS) {$< }

.cpp.obw:
$(CPP) $(WCPP_OPTS) -o$*.obw {$< }

generic.exe: generic.obj
$(LINK) $(LINK_OPTS) @&&!
$(C_OBJS)+generic.obj
$*,,$(C_LIBS)
!

wgeneric.exe: generic.obw
$(LINK) $(WLINK_OPTS) @&&!
$(WC_OBJS)+generic.obw
$*,,$(WC_LIBS),wgeneric.def
!
rc wgeneric.rc $<

```

Zortech

Workbench (ZWB)

To compile applications in the ZWB, the following options must be selected within the ZWB:

- Select Compile|Compile Options|Code Generation. Choose the Large Memory Model, select OS Support of DOS or Windows, set Structure Alignment to BYTE, turn CHAR==UCHAR off, turn Integer Only off, and turn SS!=DS on.

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
```

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;C:\ZORTECH\LIB
```

NOTE: Probably the easiest place to change the environment variables is in your **AUTOEXEC.BAT** file.

- In order to compile Microsoft Windows applications, you must be using Microsoft Windows Version 3.X. All of the options specified in the tutorial Zortech makefiles 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—either **ZIL.LIB** for DOS or **ZILW.LIB** for Windows—should be linked in.

Here is a sample “generic” makefile:

```
# Makefile for GENERIC example program
# Uses Zinc Interface Library Version 3.00 and Zortech C++ 3.00

# Be sure to set the LIB and INCLUDE environment variables for Zinc, e.g.:
# set LIB=.;C:\ZINC\LIB;C:\ZTC\LIB
# set INCLUDE=.;C:\ZINC\INCLUDE;C:\ZTC\INCLUDE

# make -fzortech.mak generic.exe      (makes the generic example for DOS)
# make -fzortech.mak wgeneric.exe    (makes the generic example for
#                                   Windows)

## Compiler and linker: (Add -g to CPP_OPTS and /CO to LINK_OPTS for debug.)
CPP=c1
LINK=link
CPP_OPTS=-c -ml -a1 -br
LINK_OPTS=/NOI
WCPP_OPTS=-c -ml -a1 -br -W2
LINK_OPTS=/NOI

## Libraries
C_OBJS=
C_LIBS=zil fg
WC_OBJS=
WC_LIBS=

.cpp.obj:
    $(CPP) $(CPP_OPTS) $<

.cpp.obw:
    $(CPP) $(WCPP_OPTS) -o$*.obw $<

generic.exe: generic.obj
    $(LINK) $(LINK_OPTS) $(C_OBJS)+generic.obj,$*, ,$(C_LIBS),NUL

wgeneric.exe: generic.obw
    $(LINK) $(WLINK_OPTS) $(WC_OBJS)+generic.obw,$*,
```



```
,$(WC_LIBS),wgeneric.def  
rc -k wgeneric.rc $<
```

Microsoft

Programmers Workbench (PWB)

To compile applications in the PWB, the following options must be selected within the PWB:

- Select Options|Language Options|C++. Choose the Large Memory Model.

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 -fmicrosoft.mak
```

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 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;C:\C700\LIB
```

NOTE: Probably the easiest place to change the environment variables is in your **AUTOEXEC.BAT** file.

- In order to compile Microsoft Windows applications, you must be using Microsoft Windows Version 3.x. All of the options specified in the tutorial Microsoft makefiles 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—either **ZIL.LIB** for DOS or **ZILW.LIB** for Windows—should be linked in.

Here is a sample “generic” makefile:

```
# Makefile for GENERIC example program
# Uses Zinc Interface Library Version 3.00 and Microsoft C++ 7.00

# Be sure to set the LIB and INCLUDE environment variables for Zinc, e.g.:
#   set LIB=.;C:\ZINC\LIB;C:\C700\LIB
#   set INCLUDE=.;C:\ZINC\INCLUDE;C:\C700\INCLUDE

#   nmake -fmicrosoft.mak generic.exe      (makes the generic example for DOS)
#   nmake -fmicrosoft.mak wgeneric.exe    (makes the generic example for
#                                           Windows)

## Compiler and linker: (Add -Zi to CPP_OPTS and /CO to LINK_OPTS for
##                      debug.)
CPP=c1
LINK=link
CPP_OPTS=-c -AL -BATCH -Gs
LINK_OPTS=/NOD /NOI /BATCH
WCPP_OPTS=-c -AL -BATCH -Gsw -DWINVER=0x0300
WLINK_OPTS=/NOD /NOI /BATCH

## Libraries
C_OBJS=
C_LIBS=zil llibce graphics oldnames
WC_OBJS=
WC_LIBS=zilw libw llibcew oldnames

.cpp.obj:
    $(CPP) $(CPP_OPTS) $<

.cpp.obw:
    $(CPP) $(WCPP_OPTS) -Fo$*.obw $<

generic.exe: generic.obj
    $(LINK) $(LINK_OPTS) @<<zil.rsp
    $(C_OBJS)+generic.obj
    $*, ,$(C_LIBS),NUL
    <<

wgeneric.exe: generic.obw
    $(LINK) $(WLINK_OPTS) @<<zil.rsp
    $(WC_OBJS)+generic.obw
    $*, ,$(WC_LIBS),wgeneric.def
    <<
    rc -30 -k wgeneric.rc $<
```

APPENDIX B – COMPILED BGI FILES

The Borland compiler converts **.BGI** files to **.OBJ** files so that you can link the graphics driver into your program. This appendix shows how you can use **.OBJ** graphics files in your program. For simplicity we will modify the first “Hello World!” tutorial program. To change the screen initialization, add the following code (shown as the bold regions of the program):

```
// HELLO1.CPP - Initializing the display, event manager and window manager.
// COPYRIGHT (C) 1990-1992. All Rights Reserved.
// Zinc Software Incorporated. Pleasant Grove, Utah USA

#include <ui_win.hpp>
#include <graphics.h>

main()
{
    // Initialize the screen display, trying for graphics mode first.
    // The graphics overlay files are linked into the application program.
    int mode;
    int driver = DETECT;
    detectgraph(&driver, &mode);
    switch(driver)
    {
        case EGA:
        case EGAMONO:
        case VGA:
            registerbgidriver(EGAVGA_driver);
            break;

        case CGA:
            registerbgidriver(CGA_driver);
            break;

        case HERCMONO:
            registerbgidriver(Herc_driver);
            break;
    }

    // Construct the new display.
    UI_DISPLAY *display = new UI_BGI_DISPLAY;
    if (!display->installed)
    {
        delete display;
        display = new UI_TEXT_DISPLAY;
    }

    // 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;
```

```

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

// 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_PROMPT(0, 0, "Hello World!");

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

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

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

// Clean up.
delete windowManager;
delete eventManager;
delete display;
return (0);
}

```

The only change you need to make to your source code are the changes that explicitly call **detectgraph()** and **registerbgidriver()**. All other parts of your code remain the same.

The Borland compiler has a support program called **BGIOBJ.EXE** that converts **.BGI** files to **.OBJ** files. The following steps are required to compile a **.BGI** file:

- 1—Change to the directory that contains the **.BGI** file you want to convert.
- 2—Convert the **.BGI** file by executing the **BGIOBJ** program with the **.BGI** file as the argument (no extension is required) by typing:

```

bgiobj.exe <bgiFileName>

```

After the **.OBJ** file is produced, include it in your program's makefile or project file so that it is linked into the executable program.

For example, the following command line remakes the hello world program with the EGAVGA driver (you should have the **TURBOC.CFG** file set so that the include and library paths for Zinc Interface Library are defined):

```
bcc -ml -P hello1.cpp egavga.obj zil.lib graphics.lib
```

Compiled **.BGI** files are useful because the `UI_BGI_DISPLAY` constructor does not need to search your `PATH` environment to try and find the **.BGI** files. (If no file is found the graphics display is not initialized.) The drawback with compiled **.BGI** files is that you must compile all possible display types and link them into your application. This will cause the executable file to be larger than that produced without the compiled files.

APPENDIX C – 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 Interface Library examples directory. For additional updates and examples, keep in contact with our Bulletin Board Service and the Zinc Software technical support group.

Each **.ZIP** file contains several compressed files with different extensions. An explanation of these extensions follows:

.HPP—A header file to be used with the example code.

.CPP—A file containing example code.

.TXT—A text file to be used with the GENHELP utility in preparing the **.HPP** and the **.DAT** files needed for use with the UI_HELP_SYSTEM.

READ.ME—A text file explaining the functionality of each example.

.MAK—A make file that will compile and link the example program. The syntax to be used in making the executable file is:

```
make -f<make file name> <program name>
```

(NOTE: The make file name is either: **BORLAND.MAK**, **MICROSFT.MAK**, or **ZORTECH.MAK** depending upon which compiler you are using.)

ANALOG

This program displays two constantly updating, sizeable analog clocks to the graphics display. This is accomplished by implementing a multiple inheritance class derived from UI_DEVICE and UIW_WINDOW.

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.

CALC

This program uses a `CALCULATOR` class derived from `UIW_WINDOW` to display a calculator, which consists of a `UIW_BIGNUM` 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.

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 `UIW_VT_LIST` and `UIW_WINDOW`.

CHECKBOX

This program derives a checkbox and radio button class from the `UI_WINDOW_OBJECT` base class and demonstrates the implementation of these classes. The classes `CHECKBOX` and `RADIO_BUTTON` use the `UI_DISPLAY::Bitmap()` function to display themselves in graphics mode and the `UI_WINDOW_OBJECT::Text()` function to display themselves in text mode. The class `RADIO_BUTTON` allows for any number of groups of radio buttons to be created, with only one button in a group being selected at one time.

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`.

COMBOBOX

This program derives a combo box class from the `UIW_WINDOW` class. The `COMBO_BOX` class demonstrates how to pass information between two windows and how to implement the logical association of two classes (eg. `UIW_STRING` with a `UIW_BUTTON`) as one class (e.g., `UI_COMBO_BOX`).

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 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.

DRAW

This program paints information to the screen using the `UI_DISPLAY::Rectangle()` function and the `UI_EVENT_MANAGER` class.

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.

ERROR

This program toggles between using the `UI_ERROR_SYSTEM` and the `UI_ERROR_SYSTEM` when a function key is pressed.

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.

FREESTOR

This program implements a free store exception handler. When the `new()` operator fails to allocate memory, the Freestor can be used to allow the user application to recover gracefully.

GRAPH

This program displays line graphs, bar graphs, and pie graphs inside of several overlapping windows.

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 then 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.

NOTEPAD

This program creates two "note pad" windows, each with a `UIW_STRING`, a `UIW_DATE`, and a `UIW_TEXT` class object attached to it. These windows are attached to the window manager, allowing the end-user to cut and paste text between the windows, etc.

PERIODIC

This program creates a periodic table of elements. `Periodic` implements user functions and uses the Zinc data file.

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.

PIANO

This program uses objects of a class derived from `UIW_BUTTON` to display a piano keyboard in a window. The keys can be selected with the mouse or with the keyboard in order to play music. This program also demonstrates how to assign hot keys to window objects, and the use of the `WOAF_HOT_REGION` flag.

PUZZLE

This program creates a “15’s” puzzle using a class derived from `UIW_WINDOW` and a group of `UIW_BUTTON` class objects. This program demonstrates how to size button objects and move them within a window.

SATELLITE

This program computes the elevation and azimuth of a satellite (for your latitude and longitude). It uses the `UIW_REAL` class.

SERIAL

This program implements a simple RS-232 communications device. It allows two users to “chat” between two machines via a serial connection. Multiple inheritance is used to derive a new window from `UIW_WINDOW` and `UI_DEVICE`. Adding user devices to the event manager is addressed.

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 a device class from the `UI_DEVICE` class. This device class, which is of type `E_DEVICE`, intercepts all events as they occur and outputs their textual representation to an object of the example class `TTY_WINDOW`.

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.

APPENDIX D – 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
class UIW_WINDOW : UI_WINDOW_OBJECT
```

In addition, the following prefixes are used in conjunction with Zinc Interface Library:

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 Interface Library - BORDER.CPP
// COPYRIGHT (C) 1990-1992. 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 Interface Library - BUTTON.CPP
// COPYRIGHT (C) 1990-1992. 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.

```
FORM *cardForm;      // The card form structure.
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.

```
// Release the memory associated with the card form.
DestroyPortal(cardForm->portalNumber);
DestroyForm(cardForm);

// Build and display a menu and allow the user to interact with it.
// When the user selects an option execute the appropriate menu
// action procedure.
switch (option)
{
    .
    .
}
```

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:

```
struct EXPORT UI_POSITION
{
    int column, line;
    .
    .
};

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 are indented one tab more than the first.

APPENDIX E – 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 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
NOTE: After 5:00 p.m. this number provides additional connections to the Zinc BBS.
- 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
(801) 785-8998, 2400 (8,N,1), 5:00 p.m. to 8:00 a.m. Mountain Standard Time
- 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, 2400 (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.

Borland BGI dependencies

Question: How can a program avoid being dependent on the Borland **.BGI** files at run-time?

Answer: The **.BGI** files may be converted to **.OBJ** files and be included in the project. An explanation of the steps required to convert **.BGI** files is given in "Appendix B—Compiled BGI Files" of the *Programmer's Tutorial* manual.

Borland IDE compiling

Question: How can programs written with Zinc Interface Library be compiled and linked with the Borland IDE (Integrated Development Environment)?

Answer: Do the following:

1—In the **Options|Compiler|Code Generation** menu of the IDE:

- Set memory model to **Large**.
- Turn **Word alignment** option off.
- Turn **Unsigned characters** option off.
- Turn **Treat enums as ints** option on.

2—In the **Options|Compiler|Advanced C++ options** menu of the IDE:

- Set C++ member pointers to **Support all cases**.
- Set Virtual base pointers to **Always near**.
- Turn off all the Options.

3—For DOS applications, set the **Graphics library** option to on in the **Options|Linker|Libraries** menu of the IDE.

4—In the **Options|Directories** menu of the IDE, set the **Include directories** to include **ZINC\INCLUDE** and **BORLANDC\INCLUDE** directories. More than one directory can be included in a field by using a semicolon. For example, the following directive includes the Zinc and Borland directories:


```
.;C:ZINC\INCLUDE;C:BORLANDC\INCLUDE;
```

5—Create a project by selecting **Project|Open Project** and entering a project name (for example: **HELLO.PRJ**).

6—Press <Ins> to add items to the project list, and add the source file (ex: **HELLO.CPP**) and the proper Zinc library (i.e., **ZIL.LIB** for DOS large memory model).

7—You should now be able to compile, link and run your program by selecting **RUN**, or by pressing <Ctrl+F9>.

Borland linker warnings

Question: Why are there several linking warning messages of conflicting destructors when linking Zinc Interface Library applications with the Borland linker?

Answer: Anytime a derived class is compiled with debugging turned on, a warning message occurs. These warnings only appear when the debug option is selected. In this case, the warnings can be ignored.

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 |= WOF_NON_CURRENT;

// Clear the auto-clear flag of an object.
object->woFlags &= ~WOF_AUTO_CLEAR;

// Toggle the selected status of an object.
object->woStatus ^= WOF_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 cannot be changed for Windows 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.

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 Zortech Workbench

Compile|Compile Options|Code Generation menu and

"Align"

must be set to Byte and the CHAR == UCHAR option must be off.

DOS extenders

Question: Does Zinc plan to support DOS extenders, and if so, when will this support become available?

Answer: Zinc is developing versions of Zinc Interface Library for popular DOS extenders (e.g., Ergo, PharLap, Rational). This support is currently scheduled for the 2nd quarter of 1992.

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 or a number ID it can be found by using the following code:

```
UI_WINDOW_OBJECT *object = (UI_WINDOW_OBJECT *)  
window->Information(GET_STRINGID_OBJECT, stringID);  
or  
UI_WINDOW_OBJECT *object = (UI_WINDOW_OBJECT *)  
window->Information(GET_NUMBERID_OBJECT, &numberID);
```

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 Interface Library (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 Interface Library that compiled with another. This is especially a problem when using Borland C++ 2.0 instead of Borland C++ 3.0. (Contact Customer Support about support for previous versions of compilers.)

International language

Question: Does Zinc Interface Library 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;
```

Making a window object current

Question: How can a certain window object be changed to be the current object?

Answer: This can be done by leaving an L_BEGIN_SELECT message if the top, left hand corner of the object is visible. (**NOTE:** The L_BEGIN_SELECT message also requires a position or region argument.)

```
eventManager->Put (UI_EVENT (L_BEGIN_SELECT, 0, object->true));
```

Other platforms

Question: Does Zinc plan to support other platforms in addition to Windows 3.X, DOS Graphics, and DOS Text, and if so, when will they become available?

Answer: Zinc is developing versions of Zinc Interface Library for OS/2, UNIX and Apple Macintosh platforms. Support for OS/2 and UNIX is currently scheduled for the 3rd quarter of 1992.

“Out-of-memory” errors

Question: What causes ‘Out of Memory’ errors when compiling Zinc Interface Library programs with Borland’s compiler?

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;
```

This 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 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 re-add it to its parent.

To re-display an entire window, including all of the objects attached to the window, re-add the window to the window manager.

Royalties

Question: If I build an application with Zinc Interface Library, 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, development tool or operating system, or 3) it is not competitive with or used in lieu of Zinc. (Please refer to the Zinc Interface Library End User License Agreement.)

Undetected graphics mode

Question: Why might a Borland compiled 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. When using the `UI_BGI_DISPLAY`, DOS will search the in 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.

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);
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 Interface Library?

Answer: Use the Zinc file conversion program to convert them to resources in a **.DAT** file. They may then be used as persistent objects.

INDEX

.BGI 209-211, 230, 238
.CHR 173
.OBJ 173, 209, 210, 230, 235
_WINDOWS 12, 87, 97, 101

A

abstract 12, 166, 168, 186, 190, 192, 193
accelerator keys 87-89
access 10, 46, 47, 66, 73, 82, 135, 149, 161,
164, 171
Add 13, 16, 19, 21, 28-30, 47, 57, 64, 69, 70,
74, 77, 110, 123, 143, 145, 160, 173, 195,
204, 206, 208-210, 225, 231, 235-237
ANALOG 213

B

backgroundPalette 172, 173
base class 3, 55, 63, 118, 122, 133, 139, 141,
145, 159, 163, 165, 166, 172, 191, 192,
196-198, 214
BBS 229
BIO 29, 213
block comments 224
books 1, 9
border 15, 16, 19, 28, 29, 31, 34, 35, 40, 42,
44, 63-65, 76, 85, 91, 97, 105, 110, 113,
125, 150, 159, 162-164, 171, 175, 210,
220, 222
Borland BGI dependencies 230
Borland compiler 167, 182, 209, 210
Borland IDE 230, 233
Borland linker warnings 231
boundaries 31, 162, 163
bulletin board service 5, 213

C

CALC 214
CALENDAR 214
cascade 93
cell 16, 161, 172, 177
cellHeight 28, 150, 160, 162, 172-174, 177
cellWidth 28, 151, 152, 172-174, 177
Changing object flags 231
Changing the map tables 231
check box 232
CHECKBOX 214
class derivation 55
cleanup 10, 18, 22, 27, 33, 47
CLOCK 214
Closing the current window 232
coding standards 4, 219
color mapping 3, 177
combo box 105, 110, 214
COMBOBOX 214
compiling 10, 59, 60, 69, 70, 82, 133, 145,
146, 155, 156, 230, 235, 236
conceptual design 13, 14, 167
constant names 220
construction 16, 18, 85, 171
constructor 12, 22, 53, 54, 61, 63, 64, 71,
73-76, 84, 89, 91, 97, 105, 113, 118, 122,
125, 139, 141, 161-164, 167, 169, 171,
172, 189, 190, 192, 195, 198, 200, 211,
221, 53
context name 24
context title 25
conversion 177, 239
coordinates 9, 16, 162, 168, 175
creating windows 63
creation 9, 16, 22, 29, 37, 42, 46

D

data file usage 74

- data files 235
- data hiding 53
- database 74, 155-158, 161, 163, 165, 181
- date 4, 26, 105, 109, 110, 213, 215, 216
- default arguments 163
- default information 42
- dependencies 4, 230
- derived 3, 12, 53, 55, 63, 74, 77, 84, 116, 131, 153, 159, 161, 163, 168, 169, 190-193, 197, 213, 214-217, 231, 234, 237
- derived classes 3, 53, 55, 116, 131, 193, 55
- derived object 191, 197
- design.exe 37
- destructors 51, 53, 54, 231, 53
- device (derived)
 - macro 133
- DIRECT 146, 174, 175, 215
- directory 25, 173, 174, 196-198, 203, 204, 210, 213, 215, 220, 230
- display 2-4, 9-14, 18, 23, 24, 26-29, 31, 33, 47, 59, 61, 64, 67, 68, 71, 81, 82, 85, 86, 87-89, 91, 97-103, 106, 108, 113, 116, 122, 123, 125, 128, 134, 140, 146, 148, 150-153, 157, 160-162, 165-177, 182-184, 186, 188, 190, 192, 209-211, 213-217, 224, 226, 233, 237, 238
 - creating new 167
- Display/Mouse remaining active 233
- DRAW 12, 14, 152, 153, 167, 174, 175, 182, 184-191, 193, 215

E

- encapsulation 52, 57, 64, 185, 188
- ERROR 2, 19, 21-27, 31-33, 47, 57, 60, 64-66, 121, 130, 215, 236
- error system 2, 22, 25, 26, 31, 32, 25
- errors 235, 236
- Event 2, 3, 9, 11, 13, 17, 18, 23, 26-28, 47, 48, 59, 61, 65-68, 71, 81-90, 92-95, 98, 99, 100-102, 106-109, 113-128, 130, 133-143, 147, 148, 150-153, 155, 157-162, 164, 165, 209, 210, 215-217,

- 219, 220, 222, 224-227, 231, 232, 236, 238

- Event (function)
 - DOS 150
 - Windows 151
- event driven 3, 83, 84, 90
- event flow
 - DOS 66
 - Windows 67
- event manager 2, 9, 11, 13, 18, 27, 47, 67, 85, 89, 94, 95, 118, 124, 134, 136-141, 143, 209, 217
- event passing 66
- eventMapTable 220
- example programs 4, 5, 213, 219
- executable programs 10, 51
- Exit 19
- exit function 21, 22, 27

F

- field information 44
- field movement 34, 163
- file layout 222
- FILEEDIT 215
- Finding an object in a window 233
- Finding the current window 234
- Finding the parent window 234
- Fix-up overflow errors 235
- Flash Graphics 167, 184, 185
- font 169, 170, 172-174, 176
- fonts 172, 173
- free store exception 215
- friend classes 53

G

Generic 29, 30, 35, 40, 56, 74, 85, 91, 97,
105, 109, 110, 113, 121, 125, 157,
204-206, 208, 225
genhelp.exe 23, 25
getting help 229
GRAPH 216
graphic objects 181-185, 188, 190-193
graphics 3, 9, 11-13, 82, 97, 121, 150, 162,
163, 167, 168, 171, 173-177, 182-190,
192, 193, 204, 208, 209, 211, 213-215,
230, 236, 238
graphics display 3, 12, 171, 174, 211, 213
graphics driver 173, 209, 238
grouping 188

H

height 16, 28, 157, 160-164, 170, 172, 176,
177, 223, 225
Hello World! 2, 7, 9-12, 15-19, 21, 22, 24,
25, 29, 31, 37, 39, 40, 42, 43, 46, 48,
209, 210
help 1-4, 17, 19, 21-25, 29-33, 46, 69, 81-83,
85-90, 92, 98, 100, 106, 108, 113, 116,
125, 126-130, 145-154, 165, 213, 215,
225, 229
help bar 145
help information 23-25, 129, 146, 149
help system 22-25, 31, 128-130, 22
help window 22, 24, 25, 129, 130, 154
hierarchy 11, 67, 196

I

icon 28, 105, 109, 110
IDE 4, 203, 230, 233
include files
 UI_DSP.HPP 10
 UI_EVT.HPP 11
 UI_GEN.HPP 10
 UI_WIN.HPP 11
indentation 224, 225
Information 3, 4, 9-17, 19, 21-26, 29, 30, 35,
40, 42-48, 59, 65, 66, 68, 69, 73, 74, 76,
81, 83, 85-87, 91, 95, 97, 101, 105, 113,
118-123, 125, 129, 133, 134, 136-143,
146-151, 153, 154, 155-157, 159-168,
172, 176, 177, 181, 182, 184, 188, 189,
191, 193, 195-200, 210, 213, 214, 215,
221, 223, 225, 226, 234, 235
inheritance 51, 52, 55, 181, 188, 197, 198,
213, 214, 217, 55
input devices 2, 9, 13, 47, 133, 135, 137, 139,
142, 143
input queue 136-142
isText 28, 150, 171
item array 85

L

library file 232
linker warnings 231
lists 4, 53, 56
Load 3, 75, 77, 160, 165, 186, 187, 189, 190,
192, 198, 199
local variables 57
logical events 121

M

macro device 3, 133, 135-138, 140-143
makefiles 59, 69, 82, 133, 145, 155, 167,
203-207
Making a window current 235
Making a window object current 235
map tables 231
mapping 3, 48, 67, 143, 177
Maximize 16, 19, 28, 29, 35, 64, 85, 91, 97,
105, 110, 113, 121, 125, 210, 19
maximize button 16, 19, 29, 110
menus 111
message passing i, 89
MESSAGES 24, 65, 67, 68, 81, 82, 84-86,
93, 94, 100, 103, 108, 115-118, 122, 127,
130, 150, 151, 152, 216, 231
Microsoft 1, 4, 10-12, 59, 60, 69, 70, 81, 82,
133, 134, 141, 145, 155, 167, 185, 204,
206-208
Microsoft Windows 10-12, 59, 60, 69, 70, 81,
134, 141, 145, 185, 204, 206, 207
Minimize 18, 19
monitor 81, 82, 87, 113, 116-123, 238
monitoring 116, 118, 123
monitoring events 116
Move 18, 19, 28, 34, 35, 118, 156, 165, 217
window 19
multiple inheritance 181, 213, 214, 217
multiple windows 28, 237

N

names 169, 196, 219, 220, 223
New 1, 11-14, 16, 22, 26, 28-31, 33, 35,
38-40, 43, 47, 55, 59, 63-65, 67, 69,
72-76, 85, 87, 91, 92, 97, 98, 101, 102,
105, 106, 109, 110, 113, 116, 118-123,
125, 129, 130, 134, 135, 137, 145, 147,
153, 157, 160-162, 165, 172-174, 181,
185-187, 189-193, 195, 198, 209, 210,
215, 216, 217, 220, 225, 226
New (function) 76

new operator 16, 29, 76
NOTEPAD 216

O

object retrieval 191, 195, 198, 190
object storage 195, 196, 190
object table 47, 69, 76, 81, 199
object-oriented 1-3, 51, 52, 84, 90, 181
object-oriented programming 1
objects (C++) 52
operators 56, 57, 142
out of memory 236
overloaded functions 56
overloaded operators 56, 57, 56

P

paint functions 185
PERIODIC 141, 216
persistence 181, 182, 193, 200
PHONEBK 216
PIANO 216
Poll 133, 135-142, 225
Poll (function)
function 141
polymorphism 56
pop-up item 81, 92, 98, 106, 114, 126, 197,
198
Preventing the modification of objects xi, 236
program design 81
pure virtual 142, 166, 168, 188, 193
pure virtual functions 166, 188
Putting a single object in multiple windows
237
PUZZLE 217
PWB 4, 207

R

radio button 214, 232
Re-displaying objects and windows 237
ReportError 26, 64
Restore 19

S

satellite 217
scope 46, 53, 83, 89, 101, 188, 221
scope resolution operator 53
scopes 221
screen display 2, 9, 11, 26, 82, 140, 173, 190, 192, 209
screenID 150-153, 168, 170, 174-176
serial 217
Size
 window 18, 19
SPY 217
storage 4, 72, 74, 75, 77, 147, 153, 181, 186, 189, 190, 195-199, 216, 220
Store (function) 76
structured 17, 87, 89, 90, 183
system events 121

T

technical support 4, 5, 213, 229

U

UI_DEVICE 3, 118, 133, 135, 139, 140, 142, 213, 214, 217, 224, 225
UI_DSP.HPP 10
UI_EVT.HPP 11
UI_GEN.HPP 10
UI_WIN.HPP 11
UI_WINDOW_MANAGER 14, 16, 24, 26-28, 93, 99, 115, 122, 123, 157, 210
undetected 238
Undetected graphics mode 238
user function 27, 64-66, 74, 92, 99, 107, 114, 126, 148, 149, 214, 215, 232, 234, 238
using classes 54
Using member functions as user functions 238
Using the Q_NO_BLOCK flag 238

V

VALIDATE 26, 217, 238
virtual list 3, 59, 82, 155-163, 165, 166

W

WOF_NON_FIELD_REGION 16, 29, 31, 42, 148, 157, 159, 163, 214, 222, 226
Workbench 205, 207, 233

Z

ZIL_PERSISTENCE 200
Zinc Designer 3, 37, 39, 46, 72, 235, 37
Zinc Interface Library 1-5, 9-14, 18, 19, 21, 22, 25, 29, 48, 59, 67, 69, 77, 83, 87, 89, 90, 111, 117, 133, 135, 143, 145, 155, 166, 167, 169, 170, 175, 177, 181, 195,

196, 200, 203, 204, 206, 208, 211, 213,
219, 222, 223, 230, 231, 233, 235-237,
239
Zortech 2, 4, 10, 12, 59, 69, 82, 133, 145,
155, 167, 173, 184, 185, 205, 206, 213,
233
ZWB 4, 205

Z