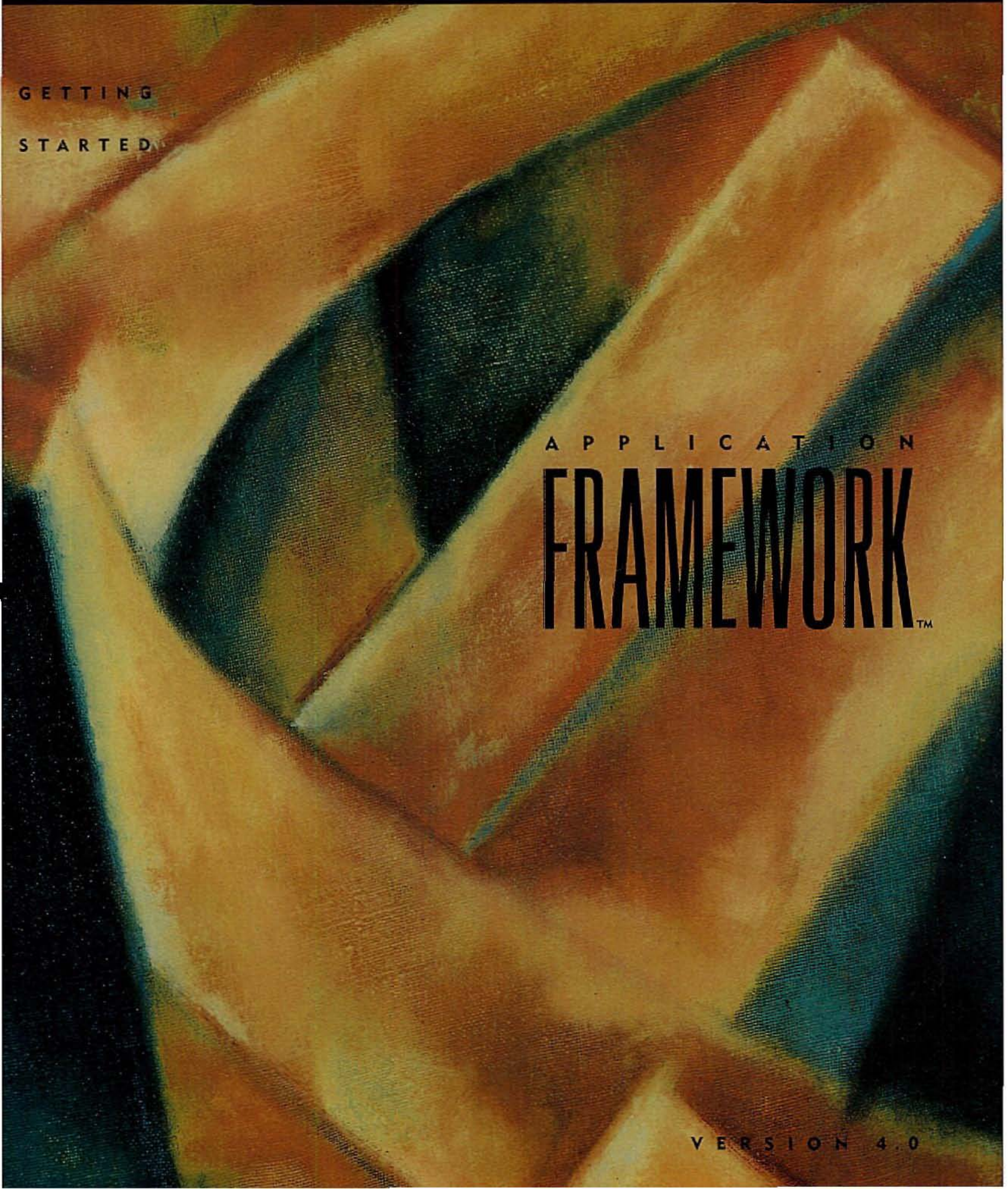
Z i n c

GETTING STARTED

APPLICATION FRAMEWORK™

VERSION 4.0



Getting Started
with Zinc Programming

Zinc® Application Framework™
Version 4.0
Zinc Software Incorporated
Pleasant Grove, Utah

Preface

If you want to learn to program using Zinc, this manual is for you.

This book teaches programmers how to write robust programs using Zinc Application Framework, the advanced object-oriented development environment that runs under nearly every popular operating environment in the world.

Zinc's mission is to help programmers write object-oriented, graphical, event-driven programs that are portable across operating systems, CPU architectures, and languages and locales. Programmers often must deal with issues like writing programs that run under multiple environments, or use unrelated display technology, or that show text and data formatting in multiple languages like English, German, and Japanese. By design, Zinc makes writing these programs far easier. But to achieve this, Zinc had to become different from other programming environments—and this difference means the programmer who is just starting out with Zinc faces the prospect of learning something new.

No other book before this one explained to *novice* Zinc programmers how to write a Zinc program step by step. Though programmers who used other environments found Zinc's reference manuals invaluable for their depth of information, the programmer just starting out with Zinc found it hard to learn the core principles of writing Zinc programs only through following the tutorial. Therefore we designed this book to help the novice Zinc programmer get up to speed with maximum speed and efficiency—and with a minimum of intimidation.

Since this book was designed to help the programmer who is just starting out with Zinc, we will occasionally cover a subject in less detail than more expert programmers would prefer. We encourage Zinc masters to look to our *Reference Manual* for more detail.

Things we've left out altogether are in-depth discussions about object-oriented programming, programming in C++, and operating systems. Although we don't expect you to be an expert C++ programmer, we do expect that you have some knowledge and understanding of object orientation and C++ before you start this tutorial.

To teach you the conceptual framework of writing Zinc programs, we start out with one of the smallest programs possible. After, we introduce more complicated, though still easy to understand, example programs, designed to teach specific Zinc features and benefits. This approach offers an opportunity to understand how every line of code works and fits together—and why Zinc is a wonderful choice for writing applications with graphical interfaces that run under multiple operating environments, languages, and locales.

While you learn how to write Zinc programs, you'll also learn some key principles important to how Zinc accomplishes its mission of portability. Each chapter will emphasize one of these key principles to keep you focused on learning that principle well. Later on, you can generalize these principles to help you write any program with Zinc.

By the end of this book, you will know enough about Zinc to use it on your own. But you'll probably want to refer back to this manual from time to time to refresh your memory about how to accomplish a specific task

What is Zinc?

Zinc is an application framework that programmers use to write object-oriented, graphical, event-driven programs that are portable across operating systems, CPU architectures, and languages and locales.

But more than a mere set of tools, Zinc is also an *architecture*, or a coherent structure that follows a set of design principles. Zinc discusses these design principles in detail in the first part of this manual. Briefly, however, in its classes and member functions, Zinc uses specific design principles of event-driven architecture, object orientation, portability, and flexibility. Zinc programs that use these classes and follow these principles benefit by how easily they port to different operating systems and CPU architectures, and how flexibly they adapt to different languages and locales.

As we learn to write Zinc programs, Zinc's intuitive design will stand out more and more—indeed, you will be able to anticipate how features of Zinc will work without having used them. This quality is what makes Zinc attractive to so many programmers around the world.

What you need to write Zinc programs

Writing Zinc programs means purchasing a Zinc Engine and a Key for the target operating environment. You will also be required to have a supported compiler for that environment.

System requirements

DOS text and DOS graphics. To write Zinc programs for DOS in real mode, you need a Zinc Engine and DOS Key; a C++ compiler for DOS such as the Borland C++, Microsoft C++, or Symantec C++ compiler; DOS 3.1 or later; and a Microsoft mouse–compatible driver. To write Zinc programs for DOS Text and DOS Graphics in protected mode, you need the above as well as a DOS extender SDK. See the **READ.ME** file for a list of currently supported DOS extenders. Most "real-world" applications will require a DOS extender.

Windows. Zinc Engine and Windows Key; a C++ compiler for Windows such as the Borland C++, Microsoft C++, or Symantec C++ compiler; and Windows 3.0 or later. To develop applications for Windows NT you need a Zinc Engine and Windows Key; and a C++ compiler for Windows NT, such as the Borland C++, Microsoft C++, or Watcom C++ compiler.

OS/2. Zinc Engine and OS/2 Key, a C++ compiler for OS/2 such as the Borland C++, IBM C++, or Watcom C++ compiler, and OS/2 2.0 or later.

Macintosh. Zinc Engine and Macintosh Key, a C++ compiler for the Macintosh such as the Symantec C++ compiler, and Macintosh System 7 or later.

OSF/Motif. Zinc Engine and Motif Key, a C++ compiler compatible with AT&T's cfront version 2.1, and OSF/Motif 1.1or later running on X11R4 or later. You may need to change some source code to use the Motif Key on hardware platforms that are not directly supported by Zinc. Though Zinc makes no claim that Zinc programs written for a version of OSF/Motif not directly supported by Zinc will work properly, doing so should be straightforward.

Unix Curses. Zinc Engine and Curses Key, and a C++ compiler compatible with AT&T's cfront version 2.1.

NEXTSTEP. Zinc Engine and NEXTSTEP Key; and NEXTSTEP 3.2 User and Developer editions or later, which come with the required compiler.

The manuals

Programmer's Reference

The *Programmer's Reference* is comprised of two volumes.

The *Programmer's Reference Volume 1* contains descriptions of Zinc Application Framework support classes, the calling conventions used to invoke the class member functions, short code samples using the class member functions, and information about other related classes or example programs. Support objects are those objects that are not window objects.

The *Programmer's Reference Volume 2* contains descriptions of Zinc Application Framework window object classes, the calling conventions used to invoke the class member functions, short code samples using the class member functions, and information about other related classes or example programs.

Some miscellaneous information is presented in the Appendices of *Programmer's Reference Volume 2*. This section (Appendices A through I) contains support definitions, system event definitions, logical event definitions, class identifications, storage information, internationalization information, and some hardware issues.

Getting Started

Getting Started contains a general overview of Zinc's architecture in addition to a series of tutorials designed to help us learn how to write Zinc programs.

If you're a Zinc novice, or if you're a beginning or intermediate C++ programmer, you should probably begin at the beginning of this book and learn what the pieces of Zinc are and how they fit together. If you've already learned about Zinc, or if you have extensive experience with C++, you may want to start with "Section Two—Zinc Programming," which teaches you how to write many different Zinc applications.

Zinc Designer

Zinc Designer contains an overview of the principles of Zinc's interactive interface design tool, in addition to feature-by-feature explanations of Zinc Designer's functionality.

Technical support

Zinc Software Incorporated offers a comprehensive technical support program to registered users, so be sure to complete and return the registration card. Currently, Zinc registered users are eligible for the following support services at no charge:

Limited warranty. The terms of your limited warranty are explained in the Zinc Application Framework End User Software License Agreement.

Telephone support. If you need assistance beyond what the Zinc manuals or your reseller can provide, you can call +1 801 785 8998 between 8:00 a.m. and 5:00 p.m. Mountain Time, or +44 (0)181 855 9918 between 9:00 a.m. and 5:00 p.m. London Time, or +81 (052) 733 4301 between 9:00 a.m. and 5:00 p.m. Japan Time to speak with one of our technical support representatives. Technical support is closed during the noon hour and on weekends and holidays. Please have the following information ready before you call:

- Your Zinc version number, serial number, and registered name
- Your hardware and operating system configuration
- Your compiler and version number

Electronic support. If you want to send messages to Zinc's technical support representatives, download software maintenance releases (requires passwords), exchange ideas with other programmers, or download user contributions, you can use the following electronic support services:

- Zinc Fax. In North America, call +1 801 785 8996. In Europe, call +44 (0)181 316 7778. In Asia, call +81 (052) 733 4328. If you need to send more than one page of code, don't use the Fax.
- Zinc BBS. In North America, call +1 801 785 8997 with 300-9600 baud (V.32bis), 8 data bits, no parity, 1 stop bit or +1 801 785 8995 with 300-9600 baud (HST dual standard), 8 data bits, no parity, 1 stop bit. In Europe, call +44 (0)181 317 2310 with 300-9600 baud (HST dual standard), 8 data bits, no parity, 1 stop bit. In Asia, call +81 (052) 733 4359 (HST dual standard), 8 data bits, no parity, 1 stop bit. Zinc's BBS is accessible 24 hours a day.
- Internet. Zinc's Internet connection is accessible 24 hours a day.

  Email: tech@zinc.com

  Anonymous ftp: ftp.zinc.com

  Web server: http://www.zinc.com/
- CompuServe—Zinc's CompuServe forum is accessible 24 hours a day.

  GO ZINC

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

Zinc's technical support program is subject to change without notice.

Conventions used in this book

This manual uses the following conventions:

TABLE 1. Conventions

Italics	identify arguments, variables, and pointers in function and method prototypes.
Bold	identifies file and directory names, and Zinc class and member function names.
`Constant width text`	identifies programming examples and command line or shell output.
`c:`	is the command line DOS prompt, which you can access from inside Windows

section one

Zinc concepts

Getting Started with Zinc Programming 2

Chapter 1

Installing Zinc

This chapter explains how to install Zinc for all its supported operating environments. Refer to this chapter for instructions on how to ensure Zinc's components are installed correctly. Also, refer to the appropriate section for your operating environment for installation instructions.

Safety first. . .

Before actually installing Zinc Application Framework, back up your distribution disks.

Key Concepts

how to install Zinc in your operating environment

DOS, Windows, and OS/2

Installing Zinc on a DOS, Windows, or OS/2 system takes five steps:

Confirm license agreement

1. *Confirm license agreement.*

   To install Zinc Application Framework, read and accept the Zinc Application Framework End User Software License Agreement and the Source Code License Addendum. The license agreement is found at the beginning of this manual. To confirm and proceed with the installation, select "yes." Otherwise, select "no" and the installation will abort.

Run the installer program

2. *Run the installer program.*

   The install program is a DOS executable, and should be run from DOS or a DOS window in OS/2, Windows, or Windows NT.

   Insert into your floppy drive the first Zinc Engine diskette. Then run the Zinc 4.0 Installer on the diskette by typing the following:

   ```
   A:INSTALL
   ```

Select a drive and subdirectory

3. *Select a drive and an installation subdirectory.*

   Select the hard drive on which to install Zinc. Then, on that hard drive, select an installation subdirectory. Press <Enter> to accept the default directory, **\ZINC**, or type in a new directory and press <Enter>.

Select the package option

4. *Choose Zinc engine and key(s).*

   The following is a list of diskette packages you need to use Zinc on your computer.

   Required

   - Zinc Engine

   Optional

   - DOS Key
   - Windows Key
   - OS/2 Key

Install Zinc

5. *Install Zinc.*

   The program installs Zinc from the distribution floppies to your hard drive and displays its progress on the screen. Periodically, it will prompt you for a new disk. Remove the current disk from the drive, insert the appropriate new disk, and press any key to continue the installation.

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

Macintosh

On a Macintosh computer, the install process takes six steps:

Confirm license agreement

1. *Confirm license agreement.*

   To install Zinc Application Framework, read and accept the Zinc Application Framework End User Software License Agreement and the Source Code License Addendum. The license agreement is found at the beginning of this manual. To confirm and proceed with the installation, select "yes." Otherwise, select "no" and the installation will abort.

Run the installer program

2. *Run the installer program.*

   Insert into your floppy drive the first Zinc Macintosh Key diskette. Then run the Zinc 4.0 Installer icon on the diskette. After reading the **README** file, select "Continue."

Choose an installation

3. *Choose an installation.*

   Choosing "Install" installs the entire Zinc Application Framework package. If you choose the default Zinc installation, skip to the next step.

   Choosing "Custom" allows you to specify only those components of Zinc Application Framework you wish to install. You can select the entire Zinc Application Framework by choosing "Zinc Application Framework 4.0 (All)," or you can select from a range of options by clicking on the first option in the range, and then, while holding the <Shift> key on the keyboard, clicking on the last option in the range—this will select all options between them. Then choose "Install" after you have selected the desired components.

Specify an installation folder

4. *Specify where to install Zinc Application Framework.*

   The Symantec project manager requires that you install Zinc in the same folder that contains the Symantec THINK Project Manager. If you wish to install Zinc Application Framework into a folder with a name other than the default, enter the new name in the field provided.

Install Zinc

5. *Begin installation of Zinc.*

   Choose "Save" to begin installing the files.

   When installation is complete, you may install Zinc in another location, or you may simply quit.

Make aliases

6. *Make aliases.*

   The Symantec compiler needs to know how to locate Zinc files when compiling. Make aliases of the **SCCPP700 Include** folder, located in the Include folder, and of the **SCCPP700 Library** folder, in the **Library** folder. Move these aliases to the Aliases folder within the Symantec C++ folder.

   Installation is now complete. You may wish to precompile Zinc's header files, which will speed up compile time considerably. To do so, refer to the file **MAC.TXT**, included in the **Read Me Files** folder in the Zinc directory now installed on your hard drive.

OSF/Motif and Unix Curses

Installing Zinc on an OSF/Motif or Curses system, takes three steps:

Confirm license agreement

1. *Confirm license agreement.*

   To install Zinc Application Framework, read and accept the Zinc Application Framework End User Software License Agreement and the Source Code License Addendum, found at the beginning of this manual. To confirm and proceed with the installation, select "yes." Otherwise, select "no" and the installation will abort.

Extract Zinc

2. *Extract Zinc from distribution media.*

   Copy the Zinc distribution to your system by following the appropriate instructions in one of the sections below. The examples below will place the Zinc distribution in **/usr/local/Zinc**.

   a. Installing from tape. To install Zinc from a tape, change directory to the installation directory. Use the **tar** command to extract the contents of the tape. For example, use **tar xv** or **tar xvf TAPENAME**, where **TAPE-NAME** is the name of the tape drive on your system, such as **/dev/rmt/1m**.

   b. Extracting from DOS floppy or DOS BBS. To install Zinc from the DOS file **ZAF4xMTF.TZ**, mount the DOS floppy or use a communications software package to retrieve the file, then move or copy the DOS file into the installation directory.

   Use **zcat** and **tar** to uncompress and unarchive the distribution files:

```
localhost> cat zaf36mtf.tz | zcat | tar xvf -
```

   If you purchased the Zinc Unicode key, uncompress and unarchive the distribution files this way:

```
localhost> cat zaf36uni.tz | zcat | tar xvf -
```

   c. Extracting from a file. If you received the file **zaf.motif.4.x.tar.Z** over the Internet, move the file to the location that you want to contain the Zinc directory tree, such as **/usr/local**.

   Use **zcat** and **tar** to uncompress and unarchive the distribution files:

```
localhost> zcat zaf.motif.3.6.tar.Z | tar xvf -
```

   If you purchased the Zinc Unicode key:

```
zcat zaf.unicode.3.6.tar.Z | tar xvf -
```

Run the installation script

3. *Run the installation script.*

   Once you have extracted Zinc from the distribution media, run the installation script called **INSTALL**.

```
localhost> ./INSTALL
```

   The script will detect whether you've installed the OSF/Motif or Curses keys. If you have, the script will ask you which you would like to use.

   INSTALL also asks questions about what type of system you have, and then it will show you the default configuration for your system type. You can change any parameters necessary. **INSTALL** then configures all the makefiles in the Zinc tree. If the C++ compiler on your system needs to have C++ source file names to end with something besides **.cpp**, such as **.C**, **.cc**, or **.cxx**, **INSTALL** changes all the source files in the Zinc tree.

NEXTSTEP

Installing Zinc on a NeXT computer or on a PC running NEXTSTEP takes three steps:

Confirm license agreement

1. *Confirm license agreement.*

   To install Zinc Application Framework, read and accept the Zinc Application Framework End User Software License Agreement and the Source Code License Addendum, found at the beginning of this manual.

Extract Zinc

2. *Extract Zinc from distribution media.*

   a. Extracting from floppy. To extract Zinc from a floppy, insert the floppy into your computer and mount it in the Workspace. Click on the floppy icon in the Workspace Manager, and drag the **Zinc.pkg** icon from the floppy to a directory in which you have write permissions.

   b. Extracting from DOS BBS. To install Zinc from the DOS file **ZINC.NXT**, use a communications software package to retrieve the file, then move or copy the DOS file into the Zinc installation directory. Then rename the DOS file to **Zinc.pkg.compressed**. Last, open the Tools Inspector panel, and select **Uncompress**.

   c. Extracting from a file. If you received **Zinc.pkg.compressed** over the Internet, move the file to the Zinc installation directory. Then open the Tools Inspector panel, and select **Uncompress**.

Load the package

3. *Load the package.*

   Double-click on the **Zinc.pkg** icon to launch the NEXTSTEP Installer. The Installer will then ask you to specify an installation directory. Choose an installation directory such as **/LocalDeveloper/Zinc** or **/usr/local/lib**. When the Installer prompts you, remove the floppy in the computer and replace it with the next one.

Finished!

Now that you've reached the end of this chapter, you're finished installing Zinc. Now you're ready to learn the details of Zinc's architecture—what the pieces of Zinc are, and how they fit together.

Chapter 2

Introduction to Zinc

In the early days of the Industrial Revolution, pinmaking was a slow, excruciating process. Each pinmaker, responsible for the entire construction of each pin, would fashion its head, its shaft, and finally sharpen the pin from a solid sliver of metal. Pinmaking was so inefficient, a group of twenty talented pinmakers might produce no more than twenty pins per week. Understandably, pins were expensive.

Then came the development of interchangeable parts, and the craft of pinmaking became radically more efficient. Teams of pinmakers specialized in creating pin components—some would create the heads, some the shafts, and still others would put them together into a finished product. Because each pinmaker could benefit from the work of others, pin production soared and its costs plummeted.

Key Concepts

what Zinc is

what Zinc's components are

how Zinc benefits us

In the early days of the Information Revolution, programming, like pinmaking, was also a slow process. Like pinmakers carving pins whole from solid slivers of metal, each programmer was responsible for writing his entire program. A programmer would first design the program according to a specification, create the program's procedures from scratch, and finally test and debug those procedures in a long and drawn-out process. Programming was so inefficient, a group of twenty talented programmers might take five years to produce a robust mission-critical program. Understandably, programs, like pins in the Industrial Revolution, were expensive.

With the development of object-oriented programming, analogous to the development of interchangeable parts in the Industrial Revolution, the craft of programming became radically more efficient. Teams of programmers specialized in creating parts of programs. Some wrote file storage objects, some event handling objects, still others concatenated the objects into working programs. Because these programmers could concatenate objects into working programs without knowing how the objects worked, they often would write object-oriented programs in a fraction of the time.

Procedural programs are difficult to maintain, difficult to port to different operating environments, and difficult to enhance with new features. This is what Zinc calls “the procedural dilemma.” Caught in the procedural dilemma, procedural programmers struggle valiantly to incorporate new features into their programs. Often they give up, and rewrite their programs from scratch when incorporating new features.

Object-oriented programming helps programmers avoid the procedural dilemma by offering interchangeable software components. Object-oriented programmers realize dramatic improvements in productivity and reliability, and consequently the costs of developing and maintaining object-oriented programs plummets.

An object-oriented solution

Zinc helps programmers write object-oriented programs, in turn helping us solve the procedural dilemma.

Zinc gives us a robust library of C++ classes that we can access in our applications. This library includes classes that handle events, manage windows, display help and error messages, and write to the displays. Further, Zinc's library includes user interface objects like windows, buttons, controls, lists, menus, tool bars, strings—all native to every environment Zinc supports. Zinc's architecture is open and extensible by design, allowing us to create custom versions of Zinc objects with behaviors that precisely meet our needs. With Zinc's modularity we won't find ourselves painted into a corner.

Zinc also features an intuitive interface design tool, Zinc Designer. Because Zinc Designer is tightly integrated with the Zinc class library, from within the Designer we have direct access to all of the library's features, including event handling and window management infrastructure, and Zinc interface objects. Further, our interfaces run under any environment Zinc supports with a look and feel native to the environment.

In addition to Zinc Designer and Zinc's robust and comprehensive class library, Zinc lets us write applications to run under multiple operating environments with one set of source code, which makes porting trivial. For example, with one set of source code, we can port our Zinc applications to DOS text and DOS graphics in real and protected modes, Microsoft Windows, OS/2, Macintosh, OSF/Motif, Unix Curses, and NEXTSTEP. Further, one set of source code makes maintenance easier, letting us spend our development resources on developing new products, not on trying to juggle several versions of the same product.

Zinc also helps us write programs that we can internationalize easily. If we're writing programs that need to run in multiple languages like English, German, and Japanese, and that need to display data in formats specific to certain countries, money and dates, for example, Zinc does much of the work for us.

Transition to C++

We might question the need to learn the new features of C++, and more importantly, object-oriented programming in general. But as we learn our way around Zinc, we'll find many compelling reasons to use Zinc and object-oriented programming techniques.

The transition to object-oriented programming is nontrivial—but because Zinc has an elegant and consistent architecture, Zinc's a great place to start. Designed from the ground up for helping programmers write object-oriented programs that have graphical user interfaces, respond to events, and support multiple operating environments and languages, we'll find writing object-oriented programs in Zinc will become intuitive and natural.

However, to complete the Zinc tutorials, we recommend at least a working knowledge of object-oriented programming concepts as well as differences between ANSI C and C++. To successfully complete the tutorials, for example, you will need to understand basic principles of object-oriented programming like classes, inheritance, polymorphism; as well as basic features of C++ like constructors and destructors, member functions, virtual functions, and function and operator overloading.

The benefits of Zinc

Writing object-oriented Zinc applications offers us several benefits over writing the same application procedurally. Some of those benefits are—

Consistency. Because of its object-oriented nature, Zinc eliminates developing and maintaining multiple versions of source code for multiple platforms. With Zinc we can focus our efforts on developing, maintaining, and enhancing one set of source code, and let Zinc interact at a low level with the operating environment and display so we don't have to. Through abstraction, Zinc insulates us from the complexities of the operating environment without restricting our access to environment specific features, like Microsoft Windows messages or the raw scan codes from the keyboard.

Ease-of-use. Instead of generating source code which is difficult to optimize and is not object oriented, Zinc Designer saves our user interface as platform-independent resources.

Reusability. Not only are Zinc's base classes reusable, but any object or class that we create can become a part of our tool kit. We save time by using classes that have previously been tested and debugged. After all, "the line of code we didn't have to write is the line of code that won't break."

Extensibility. Because Zinc is object oriented from the ground up, we benefit from a powerful feature of OOP—inheritance. Rather than developing an object from scratch, we can use Zinc's base classes with their existing member functions and data to derive new classes. For example, we can create a new input device like a digitizer by deriving our own class from Zinc's device class. With inheritance we can stand on the shoulders of giants by creating only the unique characteristics of the new class and reusing the characteristics of the old class.

Maintenance. Object-oriented applications are much easier to maintain than structured programs. With object-oriented encapsulation, C++ keeps relevant data and functions together and allows us to modify an object without affecting other parts of the application.

Flexibility. Wherever possible, Zinc has chosen to give the programmer more flexibility, rather than more rules. This means that Zinc, like C++ itself, gives us more freedom to write code, and less worries about conforming to arbitrary Zinc standards.

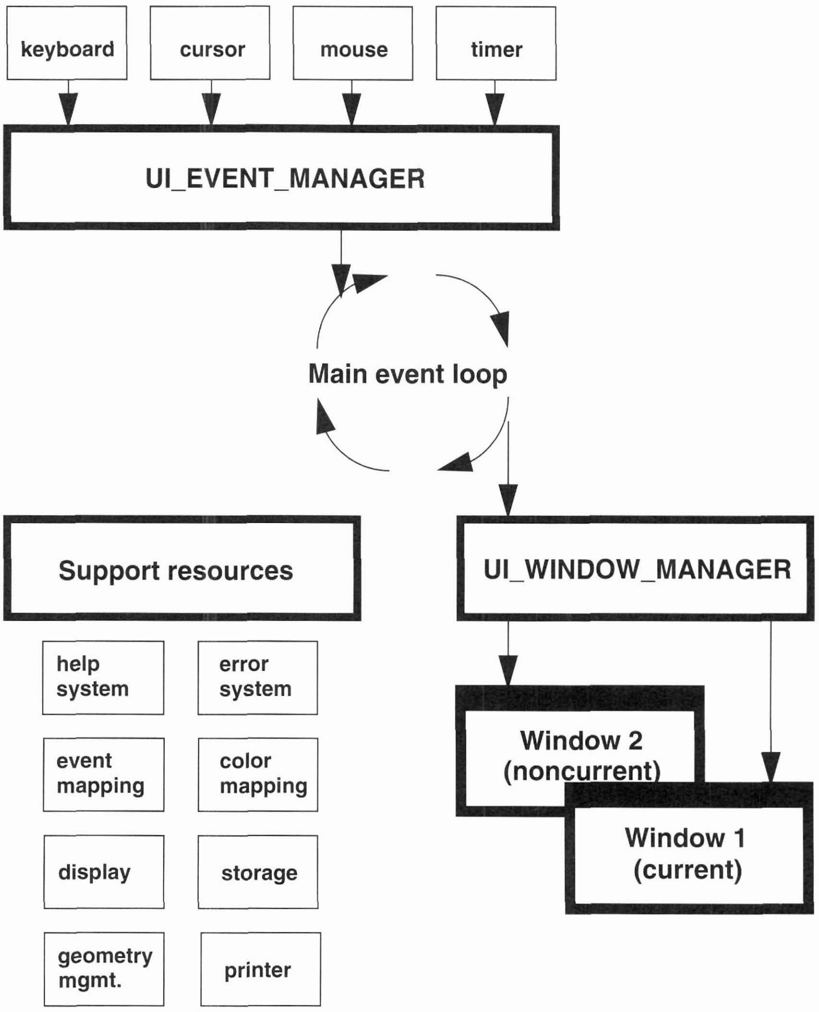
Globalization. Zinc is the only environment where programmers can write Zinc programs for all other popular languages and locales. Zinc uses the ISO8859-1 character set, which defines 8-bit characters, by default, but also provides support for the Unicode 16-bit character set using the Unicode Key. Zinc maps strings between these character sets and the native character set of the target operating systems. Additionally, Zinc also allows programmers to save language and locale information in a single file, and separate the information for applications that use different languages in the same locale, and different locales with the same language.

Zinc: an application framework

At the highest level of its architecture, Zinc consists of components that handle specific tasks; these components make up what Zinc calls the Zinc Application Framework, which is an infrastructure for helping us write event-driven, object-oriented, global programs faster than we could otherwise.

For example, one Zinc component is an infrastructure for retrieving events and routing them to the part of our program that knows how to respond to those events. Another component is an infrastructure for managing those parts of our application that respond to events, as well as managing how windows behave on screen and how they respond to user input.

This diagram displays the basic Zinc components and how they work together in an infrastructure. Study this diagram until you know how to recreate it without looking at the book, and you'll have a much easier time of understanding Zinc as we continue with this discussion.



Here's a description of all these components and what they do:

Input devices. The keyboard, mouse, cursor, timer, or any other devices that generate events.

Event Manager. Handles the flow of events and system messages throughout the application. Certain operating systems sometimes will pass events to a window object, bypassing the Event Manager.

Window Manager. Controls the behavior of windows. Certain operating systems sometimes will pass events to a window object, bypassing the Window Manager.

Display. The display of the computer running the Zinc application. Generalized by Zinc as an abstract class, from which programmers derive displays specific to particular display libraries.

Help system. Displays help information at run time.

Error system. Displays error information when a user enters inappropriate data.

Event mapping. Mapping of raw input, such as mouse clicks and keystrokes, to logical system events such as sizing, moving, and redrawing.

Color mapping. Mapping of colors in a specific operating system to Zinc colors.

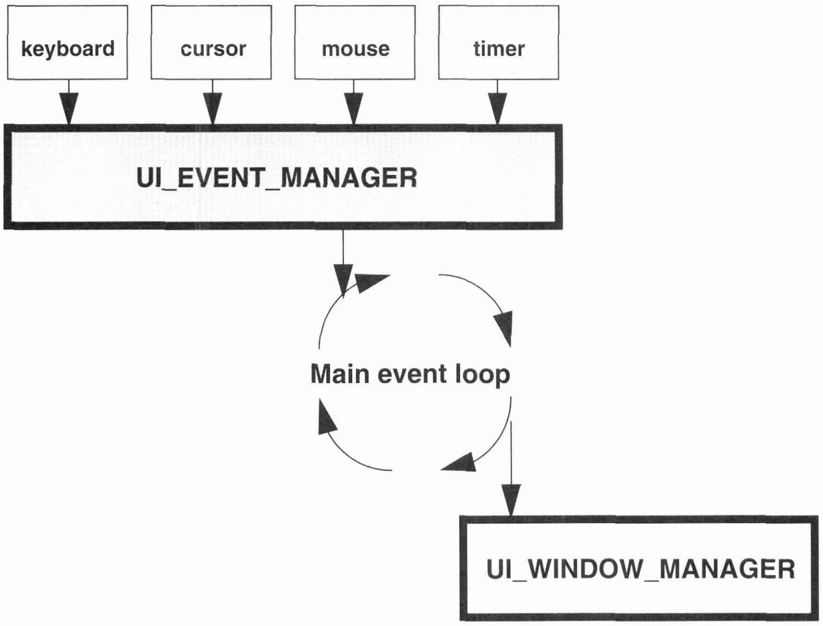
Storage. Reads and writes objects to and from disk.

Geometry management. Allows the programmer to specify rules that dictate how objects should be positioned and sized in specific situations.

Printer support. Allows the application to send output to a printer, either by performing a screen dump or by using the display primitives to draw an image.

The Event Manager

The Event Manager is Zinc's infrastructure for handling events and system messages. It accepts events from common *input devices* such as the keyboard and mouse and it stores event information in the *event queue*. The Event Manager also handles custom input devices we write ourselves like a digitizer or a scanner device.



The event queue stores events until the event loop can pass the event to the appropriate window or window object. The event queue can buffer as many as 100 events by default, but this can be easily changed by the programmer. The Event Manager deals with these events one at a time until the event queue is empty. At run time, the event loop immediately takes events from the queue and passes them to the appropriate window or window objects.

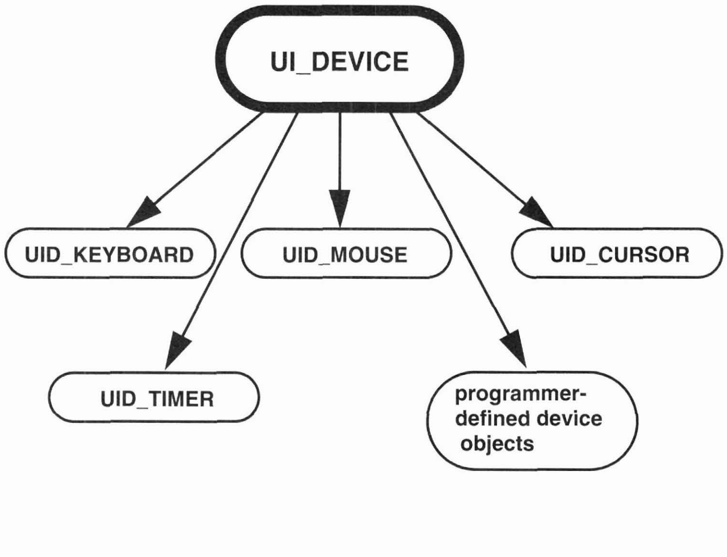
For example, in DOS, when the user presses Alt <F4>, the **UID_KEYBOARD** device, which is a specific Zinc class that handles keyboard events, receives the keystroke event and puts it into the event queue. As the event loop repeats its cycle, it passes the event to the Window Manager, which then passes it to the appropriate window or window object. The event loop repeats until the user gives the application the "quit" event.

UI_DEVICE and abstract classes

We've skimmed over how the Event Manager works with input devices to retrieve events. Now we'll explain how Zinc's devices work. Most compiler libraries have a set of functions to get input information from the keyboard, functions like **getch( )**, **getchar( )**. However, most of these libraries include neither functions to handle information from other devices like the mouse, nor functions to handle multiple input devices. Zinc provides seamless support for multiple, diverse devices, which is what makes Zinc such a flexible event-driven environment.

Zinc handles keyboard and mouse input in classes called **UID_KEYBOARD** and **UID_MOUSE**. By responding to events, which are information that comes from input devices, a Zinc program can follow the user's orders and call member functions, change data, even load new languages and locales "on the fly."

All input devices inherit from the base class **UI_DEVICE**, which is an *abstract class*. An abstract class defines the basic behavior for a type of object, but typically leaves specific implementation details to a derived class. An instance of an abstract class cannot be created; a class must be derived from it and an instance of that class created. If we were to create our own input device, we would derive it from **UI_DEVICE** and add our own functionality.



Event mapping

Many user interface libraries convert raw input information to logical information when the input device sends information. For example, a mouse device may define the left mouse button click as the *L_SELECT* operation. The programmer must then decipher the *L_SELECT* operation in the context of his or her program's operations, a task that many programmers find cumbersome.

Zinc takes a different approach to event mapping. Zinc receives raw events from input devices at run time and interprets them in the context of the object and the type of operation being performed. This means the programmer doesn't have to write as much code, freeing him or her up to focus on writing the program's core functionality.

Here's how Zinc's event mapping works. Imagine running a hypothetical application that has a main window and a text field. Here's a description of how Zinc would map the events generated in DOS when clicking the left mouse button or pressing the <F2> key on the keyboard:

1. The input device, **UID_KEYBOARD** or **UID_MOUSE**, receives the event and places the keyboard or mouse information in the event queue.
2. The Window Manager passes the event to the current window.
3. The window passes the event to the current window object.
4. The **UIW_TEXT** window object evaluates both the keyboard and mouse events as the *L_BEGIN_MARK* operation.
5. Finally, the results of the *L_BEGIN_MARK* operation return to the window and then to the Window Manager.

Benefits of logical event mapping

Here's how logical event mapping benefits us. First, each object interprets the event according to how the object operates, eliminating the need for us to write code to tie events to window objects. The **UIW_TEXT** object views both events as an *L_BEGIN_MARK* operation. However, if the mouse click returned unprocessed to the Window Manager, it would interpret it as an *L_BEGIN_SELECT* operation, while the <F2> key, which is unknown by the Window Manager, would remain unprocessed.

Another benefit from logical event mapping gives us the ability to create additional input devices that generate their own raw event information. This way, we can define logical event mapping for Zinc but still receive all the raw event information generated by the new input device. Still another benefit is that we can easily redefine key mapping without changing Zinc's source code, allowing us to customize our programs without interfering with how Zinc operates.

But the most important benefit from logical event mapping is portability. Because Zinc allows each object to behave differently, an object has the flexibility to behave differently under different operating environments. We can assign behavior on an object-by-object basis, a manageable task, in contrast to forcing an object to reevaluate its behavior in contexts of different operating environments.

The Window Manager

So what happens after the Event Manager gathers events in the event queue? How does the program know how to respond to those events? The answer lies in the Window Manager, which determines how windows and window objects behave.

Just as good tourists do what the Romans do when in Rome, Zinc ensures that when we're writing programs that run under multiple operating environments, the programs behave as though they're native to that environment. In fact, Zinc creates native applications for each environment it supports. One reason Zinc does this is to ensure each program responds to events in the manner native to that operating system, making programming simpler.

"Top down" and "bottom up"

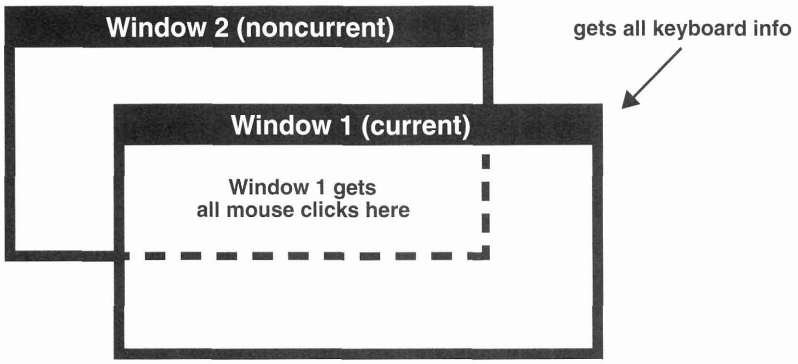
An operating environment with no native ability to handle events means that if we want to write an event-driven program for that environment, we'll have to bring our own event handling infrastructure with us. Zinc's native event handling model is a "top-down" model because events trickle down from the top, the main event loop, through the Window Manager and the current window, to the current window object. Each of these objects gets a chance to determine if it should respond to the event; if an object doesn't or can't, it merely passes it down the hierarchy.

Some operating environments that can handle events use what is called a "bottom up" event handling model in which the event goes directly to the lowest level current object. In this case, events follow a more complicated route. Here's a brief description. When we write a Zinc program that runs under Microsoft Windows, an example of an operating environment that uses the bottom-up model of handling events, Zinc relies on the native methods of Windows for its windows and window objects to respond to system events.

For example, if a user clicks on an object, the operating system gets the event and places it on its own event queue. When Zinc gets the event from the event queue and starts to process it, instead of routing the native message through the Zinc hierarchy, it sends it to the operating system and lets it process it like any other native event. Because Windows is a bottom-up environment, the event goes directly to the window that was clicked on, the "bottom" object. If that object does not handle the event, it may choose to pass it back up the hierarchy so that a higher level object can process it—hence the term "bottom up."

Window position and priority

The Window Manager maintains a list of windows and minimized windows. The Window Manager determines the position and priority of windows on the screen and channels the events to the proper windows.



For example, if Window 1 overlapped Window 2, the Window Manager would route all keyboard information to Window 1, since it is the topmost window—the current window. In addition, any mouse events that overlapped Window 1 or the area intersected by Window 1 and Window 2 will be sent to Window 1 for processing. If a mouse event overlaps the area occupied only by Window 2, however, that event would go to Window 2.

All windows and window objects derive either directly or indirectly from the **UI_WINDOW_OBJECT** base class. This means that all Zinc windows and window objects share certain behaviors and characteristics, notably the ability to appear native to the operating environment under which we compile our programs.

Native objects, not emulated

Zinc doesn't emulate the look and feel of a native object, as do some other application frameworks. Rather, Zinc uses native objects—no emulation needed. Windows and window objects native to each environment are faster because the Zinc program doesn't have to draw the objects or process system events that the native object already processes. When we write programs in Zinc for multiple environments, our programs are indistinguishable from other programs written specifically for those environments.

If you've ever used a program that runs under multiple environments, and that program uses windows and window objects different from those that you're used to, you'll understand the frustration of users who feel that the programmer didn't care enough to write that program specifically for them. Because all windows and window objects are native, it's easier for us to write applications.

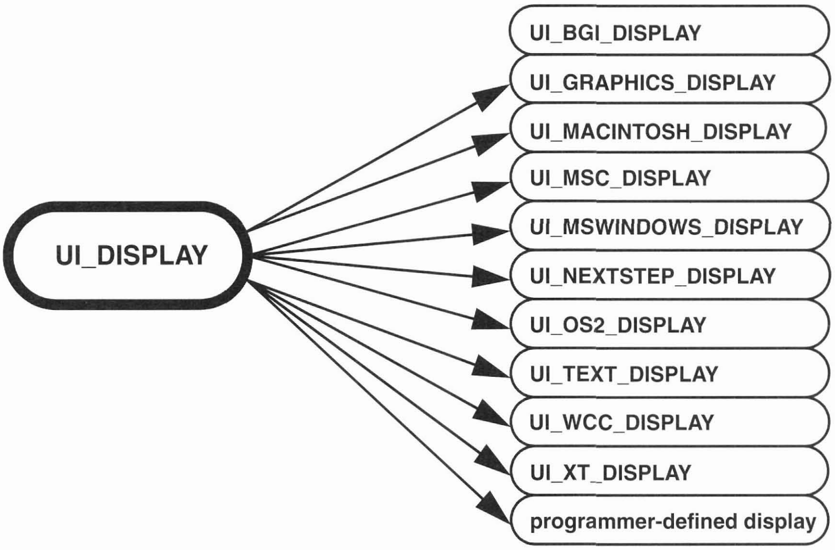
The display

Since Zinc programs support multiple operating environments, Zinc has created some infrastructure for making those programs easier to write in an intuitive and simple way.

Zinc's infrastructure is an abstract class called **UI_DISPLAY**. We will never use a display of the **UI_DISPLAY** class; rather, we will use a display derived from **UI_DISPLAY**, but with behaviors defined for a specific type of display, such as a Borland BGI display, an OS/2 display, or a NEXTSTEP display. As an abstract class, **UI_DISPLAY** defines some functions that a display object should perform, but it leaves how those functions should be performed up to the specific displays.

This object-oriented approach to handling displays gives us an attractive benefit. We can run our graphical application under all the environments Zinc supports with one set of source code, merely deriving a display specific to our own. Further, because all displays derive from **UI_DISPLAY**, they all have the same interface, making it less work to understand how to access different displays.

Here's both a representation of the **UI_DISPLAY** class hierarchy and a list of all the classes derived from the **UI_DISPLAY** base class:



The help and error systems

Most robust applications have some sort of help system to give users information about features while running the program. Zinc makes it easier for us to write such a help system with a class called, appropriately, **UI_HELP_SYSTEM**. This class uses Zinc windows to display help information, ensuring that no matter which applications we want our program to support, we'll only have to write the help information once.

Zinc initially does *not* make us include the **UI_HELP_SYSTEM** class; if we don't want the help system class linked into our programs, we don't have to use it. Zinc gives us the choice to decide whether or not we include a help system, putting us in control of how we write our own applications.

As with help systems, most robust applications have some sort of error system that tell us when we've made a mistake while running the program. Zinc's error system is a class called, appropriately, **UI_ERROR_SYSTEM**. This class uses Zinc windows to display error information. Again, as with the help system, this means no matter which environments we want our program to support, we'll only have to interface with one error system.

Zinc initially does *not* make us include the **UI_ERROR_SYSTEM** class; if we don't want the error system modules linked into our programs, we don't have to use it. Zinc gives us the choice to decide whether or not we include an error system, putting us in control of how we write our own applications.

Storage and retrieval

We've seen how Zinc gives us quite a bit of infrastructure for handling much of what goes on under the hood of an object-oriented, event-driven, graphical application. In addition to all the other infrastructure Zinc gives us, we can use Zinc's ability to save and load data to and from disk. Zinc uses an advanced method for saving and loading data to disk called persistent object technology. Persistence isn't unique to Zinc, but Zinc's flavor of persistence allows us to store and retrieve C++ objects to and from disk as platform-independent resources through low-level file management routines as well as persistent object technology.

Zinc uses its own storage and retrieval classes in Zinc Designer. When we interactively create and modify windows and window objects using Zinc Designer, we're using the same storage and retrieval classes we'd use without Zinc Designer.

Globalization

We've covered almost all of Zinc's infrastructure for writing object-oriented, event-driven, graphical applications. But we still need to discuss how Zinc makes it easy for us to write applications that run in different languages and display localized information about dates, money, and so forth.

The obstacles to reaching the global market

True globalization is a complex process. If we were to write a program and deploy it on desktops in North America, Europe, and the Pacific Rim, among other things, we would have to enable our program to be compatible with complex permutations of languages and locales, eight- and 16-bit character sets, incompatible hardware and display technologies, and a plethora of input methods.

Zinc takes more of the burden off our shoulders than any other application framework. Using Zinc's optional Unicode key ensures that our programs can detect their language and locales at run time, use both 8-and 16-bit fonts as appropriate, run on nearly all popular hardware combinations, and work with nearly all popular input methods.

ISO 8859-1 and Unicode

We're not obligated to use Unicode to deploy our Zinc applications in most areas of the world; Zinc programs automatically use the eight-bit ISO 8859-1 character set, which contains most international characters. This means the base Zinc Engine and Keys let us reach much of the world's software market right out of the box. However, if we must deploy a Zinc application in a nation that uses a 16-bit font, like most Asian countries, Zinc gives us the option to use Unicode, an international standard for character sets. Unicode contains every character from every modern language, giving Zinc a single, comprehensive standard for displaying characters.

For example, if we wrote a Zinc application and intended to distribute the executable in the United States and Japan, we'd translate the interface text into Japanese, and then use Zinc's Unicode characters to represent the Japanese text on our interface. We'd do the same thing if we wanted to translate our interface to any other language—Unicode contains any characters we'd need. Using Unicode to represent character sets makes programming easier because we only need to deal with one standard.

Language and locale

Another reason running Zinc applications in different languages and locales is easier is that Zinc gives us the ability to store different languages and locales in the same interface file. If we write our Zinc application with

English and Japanese interfaces, we don't have to juggle two different interface files; Zinc can store it for us in one place, giving us fewer components to worry about.

Zinc keeps certain globalization information separate from interface text, however; this information concerns the *locale,* or region of the world where our program will run. Part of translating a program is displaying locale information in a format that differs from country to country—date information, decimals, and currency symbols in certain window objects, for example. When we translate our program we merely specify to Zinc which locale to use; it's as easy as that. In fact, Zinc automatically detects what language and locale the environment is using, and will automatically adapt to the environment's needs.

One dramatic benefit of separating language from locale is our program's ability to use multiple languages within one distribution region. For example, if we wrote a Zinc application for both English-speaking and French-speaking Canadians, we'd still have to translate our interface into English and French, but we'd only have to specify one locale—Canada. Another benefit of separating language from locale is our program's new ability to use different data formatting in the same language. For example, if we wanted to write an application for Spanish speakers in Mexico, we'd still have to specify Mexican locale information, but we could merely translate our interface into Spanish. Again, Zinc gives us flexibility in how we write our programs, leaving the design decisions up to us.

Delta storage

Another reason running Zinc applications in different languages and locales is easier is that Zinc gives us the ability to store only the differences between languages and locales in what Zinc calls *delta storage.* If we write our Zinc application with English and Japanese interfaces, Zinc doesn't have to duplicate both interfaces, translated text and all; Zinc merely stores the differences between the interfaces, decreasing our program size and increasing its performance. Without delta storage, users would have to dedicate a larger amount of disk space to their applications.

Geometry management

Programming graphical user interfaces opens up the problem of how interface objects should relate to each other visually—this is called geometry management. Though some user interface design tools provide some rudimentary rules for how those relationships should work, Zinc takes geometry management to the next level. Zinc's geometry management allows the programmer to specify sophisticated rules that dictate how objects should be positioned and sized in specific screen resolutions, on every platform Zinc supports.

Printer support

Part of the difficulty of writing crossplatform programs is determining how to print. Zinc's printer object allows our programs to perform a screen dump, or to print an image using Zinc's display primitives—bitmaps, ellipses, lines, polygons, rectangles, as well as text. Zinc's printer support formats text across an entire page, providing page breaks as necessary. Further, Zinc provides the ability to print an environment's default printer, as well as to a PostScript file. And in DOS, which has no printer support, Zinc supports the popular PCL format.

Conclusion

Zinc allows us to write programs easily ported to other operating environments, languages, and locales. Zinc's library includes native interface objects like windows, buttons, controls, lists, menus, tool bars, and strings in every environment Zinc supports, ensuring high performance and acceptance by users. Zinc includes infrastructure that handles events, manages windows, displays help and error messages, and manages the visual relationships of interface objects, leaving us to concentrate on writing programs rather than reinventing the wheel. In the next chapter we're going to discuss Zinc's windows and window objects.

Chapter 3

Window Objects

In the last chapter, we discussed how Zinc helps programmers write object-oriented applications, Zinc's underlying infrastructure, and the types of Zinc objects we can use in our applications. In this chapter, we'll discuss Zinc's window object classes. We'll discuss each window object, what it does, and how it works.

Most Zinc windows share basic window objects; they have borders, titles, maximize buttons, minimize buttons, and system buttons. In another example of how Zinc helps us write efficient programs, Zinc doesn't make us include these basic window objects with every Zinc window we instantiate. Instead, we add to our windows the objects we want, instead of deleting objects we may not want.

Key Concepts

the different types of window objects

how window objects work

Zinc's window objects

Basic window objects

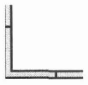
Below is a typical Zinc window and its basic window objects, in addition to the code we'd need to write to instantiate them under any operating environment Zinc supports. Notice it doesn't take much code to instantiate this window and its basic objects.

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

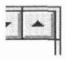
Although some operating environments don't have some of these basic window objects—for example, NEXTSTEP windows don't support maximize buttons—we can use these and all other window objects for any operating environment Zinc supports. If we use a maximize button in a Zinc program that runs under NEXTSTEP, the NEXTSTEP window simply will not display the maximize button.

Here's a list of the window objects we used in the above code, and the classes which they comprise.

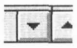
Border. The **UIW_BORDER** class. In graphics mode, the border is a three-dimensional shaded region drawn around the window; in text mode, the border is a shadow.



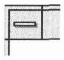
Maximize button. The **UIW_MAXIMIZE_BUTTON** class. Located on the top right side of the window. Changes the size of its parent window to occupy the entire screen display.



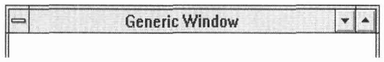
Minimize button. The **UIW_MINIMIZE_BUTTON** class. Usually located at the top right corner of the window. When pushed, it reduces the window to an icon.



System button. The **UIW_SYSTEM_BUTTON** class. When pushed, selects window or system specific commands associated with the window object, such as size, move, maximize, minimize, and close. If the system button has options, a pop-up menu appears on the screen.



Title bar. The **UIW_TITLE** class. Displays text to identify the window.



Now that we've seen the different types of basic window objects we've used in our code snippet, let's take a look at some more complicated window objects we can use in our applications.

Buttons

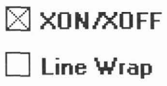
The simplest of the more complicated window objects is a button. A button is a rectangular region of the screen that displays information and performs an operation when pushed.

At its most general level, a basic button display information in the form of text. But in Zinc, we can also use more complicated buttons—bitmapped buttons, check boxes, and radio buttons, all of which look and act differently from basic buttons. Though these more complicated buttons look and act differently, they all derive from **UIW_BUTTON** and share the same behavior—they display information and perform operations. In other words, despite their more complicated behavior, they're all still buttons.

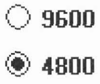
Below is an instance of **UIW_BUTTON**, the most basic button object in Zinc's library.

Bitmapped button. Displays a bitmap rather than, or in addition to, text. Bitmapped buttons used in text mode will not display graphics.

Check box. Check boxes in a window, a group, or a list box are members of the same group. Multiple checkboxes from a group may be selected at any time.



Radio button. Radio buttons in a window, a group, or a list box are members of the same group. Only one radio button from a particular group may be selected at any time.



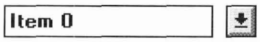
UIW_GROUP is a Zinc class for grouping Zinc objects together on screen. Once we instantiate a group object, we add to the object the desired radio buttons and check boxes. Unless we're using only one radio button or check-box, we use the **UIW_GROUP** class to group our window objects together.

Combo boxes

Another more complicated Zinc window object is the combo box. Implemented as the **UIW_COMBO_BOX** class, the combo box is a one-line string field with a button object attached, that, when clicked, displays a list of items from which we can choose.

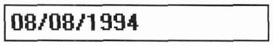
Many operating environments include the combo box, the purpose of which is to give the user multiple ways to select an option. When using a combo box, a user can select options with the mouse, or he can type the option he wants into the string field using the keyboard.

Here's how the combo box works. Consider a program that contains a list of selections. When the user pushes the button attached to the string field, a list that contains those selections appears on the screen. When the user clicks on the selection he wants, the item is copied into the string field, and then the list disappears. Alternatively, the user can type the selection into the string field directly, bypassing the pop-up list and saving time. Here's a **UIW_COMBO_BOX** object:


Item 0

Dates

When we write Zinc programs that display date information or gather date information from a user, we use objects of class **UIW_DATE**. These objects display date information and allow the user to enter and modify date information in different formats. Below is a **UIW_DATE** object:


08/08/1994

The default behavior of a Zinc date object is to display the date in a format native to the language and locale under which the program's running. However, by passing to the constructor certain styles, we can override any language or localization information.

Here's a list of all the different styles Zinc's date class supports, and a sample of how dates look using these styles.

TABLE 2. **Date styles**

Long month	Displays the entire name of the month as an ASCII string value.	`3-28-1990`
		`12-04-1980`
		`1-3-2003`

Dash	Separates each date variable with a dash.	`3-28-1990`
		`12-04-1980`
		`1-3-2003`

Day of week.	Displays the day-of-week as an ASCII string value.	`Monday May 4, 1992`
		`Friday Dec. 5, 1980`
		`Sunday Jan. 4, 2003`

European format.	Displays the date in the European format of *day/month/year.*	`28/3/1990`
		`4 December, 1980`
		`3 Jan., 2003`

Japanese format.	Displays the date in the Japanese format of *year/month/day.*	`1990/3/28`
		`1980 December 4`
		`2003 Jan. 3`

TABLE 2. **Date styles**

Military format.	Displays the date in the format *day month year*, where *month* is either a three-letter abbreviated word, and, if the *DTF_SHORT_YEAR* or *DTF_SHORT_MONTH* flags are set, *year* is a two-digit year value. If those flags aren't set, *month* is spelled out, and *year* is a four-digit value. May be overridden with other date styles.	`4 Jul 91`
		`4 July 1991`

Short day of week.	Displays shortened day-of-week value with the date.	`Mon. May 4, 1992`
		`Fri. Dec. 5, 1980`
		`Sun. January 4, 2003`

Short month.	Displays a shortened alpha-numeric month value with the date.	`Mar. 28, 1990`
		`Dec. 4, 1980`
		`Jan. 3, 2003`

Short year.	Displays the year as a two-digit value.	`3/28/90`
		`December 4, '80`
		`Jan. 3, '89`

Slash.	Separates each date value with a slash.	`3/28/90`
		`12/04/1900`
		`1/3/2003`

Uppercase.	Displays the date in uppercase format.	`MARCH 28, 1990`
		`DEC. 4, 1980`
		`SATURDAY JAN 3, 2003`

U.S. format.	Displays the date in the U.S. format of, *month/day/year*, regardless of the default country information.	`March 28, 1990`
		`12/4/1980`
		`Jan 3, 2003`

Zero fill.	Inserts zeroes before the year, month, and day values when their values are less than 10.	`March 08, 1990`
		`12/04/1980`
		`01/03/2003`

Geometry management

Though geometry management isn't a window object, it affects the way window objects display themselves in relationship to their parent windows and other objects.

An object's geometry is its height, width, and location on its parent, and geometry management is a feature that allows the location and size of other objects to determine an object's geometry. For example, we can use Zinc's geometry management to keep a button centered in its parent, regardless of the parent's size.

Icons

An icon is a small window that displays a graphic image that allows the user to recognize information quickly. Zinc's **UIW_ICON** class gives instances of Zinc icons some standard behavior and properties. For example, when we instantiate an icon of the **UIW_ICON** class, we can display it on a window or attach it to the window as a the icon to which the window will minimize.

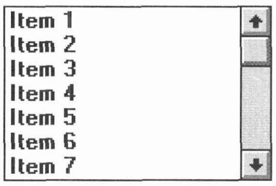
Below is an instance of the **UIW_ICON** class.

Lists

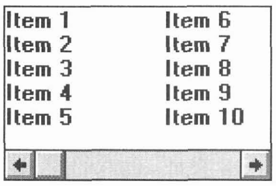
Lists provide a method of giving the user predefined, uneditable selections to choose from. Because the user can choose only the selections that we give him, we can ensure that our program can use those selections as valid input.

To give us a quick way to include lists in our Zinc applications, Zinc provides two list classes, **UIW_VT_LIST** and **UIW_HZ_LIST**, which display selections either in a vertical list with one column, or a horizontal list with one or more columns. The available selections are added to the lists as instances of other Zinc objects, typically strings or buttons.

These are instances of vertical and horizontal list objects:



Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7



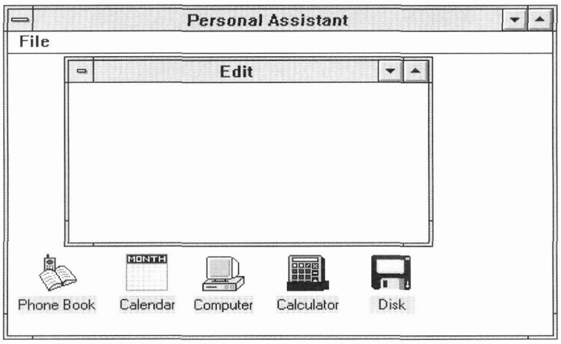
Item 1 Item 6
Item 2 Item 7
Item 3 Item 8
Item 4 Item 9
Item 5 Item 10

MDI windows

The Zinc windows we've seen so far display themselves on screen independently of each other; they can overlap and cover each other, but so far they can't display themselves inside of another window. However, the popular Microsoft Windows environment specifies a type of window called the MDI window, or multiple-document interface window, that displays itself inside another window, and so Zinc created its own MDI window object that we can use to display windows inside other windows.

However, unlike other window objects we've discussed in this chapter, Zinc's MDI window doesn't derive from its own class. A Zinc MDI window is a normal Zinc window, but with a flag that tell the window to become an MDI window. To instantiate an MDI parent and child window with Zinc, we instantiate two windows, the first an MDI parent, and the second an MDI child. However, we create these windows with the flag *WOAF_MDI_OBJECT*; then we simply attach the child to the parent with the overloaded + operator.

Zinc MDI parent windows behave like any other Zinc window; they may be maximized, minimized, moved, or sized within the MDI parent. The only restriction of MDI child windows is that they cannot move outside of their parent—the parent window clips the child at the inside of their parent's border. Below is an MDI parent window that contains an MDI child window and several minimized MDI child windows.

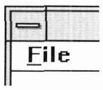


Menus

In describing Zinc's more complicated window objects, we've discussed how some of Zinc's objects present selections to the user. Now we're going to discuss what Zinc lets us do with menus. What sets menus apart from lists and combo boxes? The crucial difference between menus, lists, and combo bars is that menus provide an intuitive way to find functions associated with a specific window.

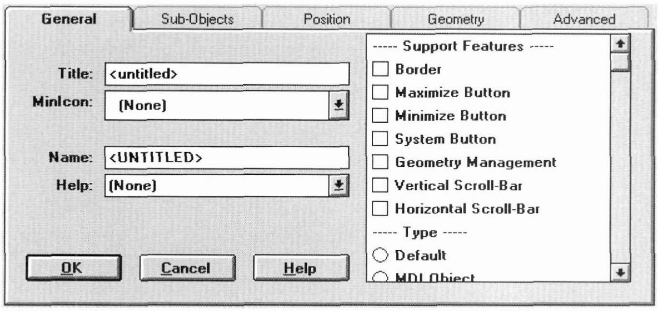
In Zinc, menus of four components: pull-down menus and items, and pop-up menus and items. The pull-down menu is the first level in the selection process. Below is a typical window with a pull-down menu object that stretches across the window below the title bar.

The pull-down menu consists of a pull-down item labelled **File**. This pull-down item lists the types of functions that the user can access while this window is active; because this pull-down item groups similar functions together, the user can find a function without sorting through the pull-down items. When the user clicks on a pull-down item, the pull-down menu displays a pop-up menu that lists those similar functions as pop-up items. Then in only a few seconds, with only one mouse click and some mouse movement, the user can merely click on a pop-up item and access that function. Here is a menu object:



Notebook

The Zinc notebook class, **UIW_NOTEBOOK**, offers an intuitive interface for navigating around groups of related objects. An instance of a notebook object has tabs like a notebook in the real world—except the notebook object "turns" to the page when the user clicks on it. Here's an instance of a notebook object, taken from Zinc Designer:



Title: <untitled>
MinIcon: (None)
Name: <UNTITLED>
Help: (None)

----- Support Features -----
☐ Border
☐ Maximize Button
☐ Minimize Button
☐ System Button
☐ Geometry Management
☐ Vertical Scroll-Bar
☐ Horizontal Scroll-Bar
----- Type -----
○ Default
○ MDI Object

Numbers

Zinc gives us several classes for when we want our programs to display or gather numeric information. Zinc supports three types of number fields with the **UIW_BIGNUM**, **UIW_INTEGER**, and **UIW_REAL** classes.

The **UIW_BIGNUM** class displays numbers with up to 30 digits to the left of the decimal point and eight digits to the right, by default. It also formats numbers using percent signs, commas, and decimal places. The **UIW_INTEGER** class displays numbers using the integer data type. The **UIW_REAL** class displays real numbers and numbers in scientific notation using double-precision, floating-point numbers. When an instance of the **UIW_REAL** class displays numbers that are too long for the field, it uses scientific notation so the user can view the entire number.

These are the display and entry styles we can use with the **UIW_BIGNUM** class, in addition to examples of how these styles look.

TABLE 3. **UIW_BIGNUM styles**

Decimal.	Shows the number with a decimal point at a fixed location.	`10,000.00`
		`43.45`
		`$149.95.`

Currency.	Shows the number with the country-specific currency symbol.	`$10,000.00`
		`DM100`
		`£195`

Credit.	Shows the number with the appropriate credit symbols whenever the number is negative.	`(1000)`
		`(23040)`
		`(759)`

Commas.	Shows the number with commas in the appropriate positions.	`$10,000.00`
		`45,000`
		`1,195`

Percent.	Shows the number followed by a percentage symbol.	`100%`
		`4.5%`
		`10%`

Scroll bar

Scroll bars allow the user to scroll an object or its information using the mouse. Both horizontal and vertical scroll bars can be created. A scroll bar is created using the **UIW_SCROLL_BAR** class.

Slider

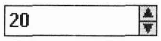
A slider is similar to a scroll bar, except that it doesn't control another object; instead it's a standalone object. A slider lets us select a setting from a range of values; it displays the current value in a range of values. A slider is created using the **UIW_SCROLL_BAR** class by setting the *SBF_SLIDER* flag in the constructor.

Spin control

Many people who have worked with electronic equipment have used a dial to quickly flip through a range of information. A dial gives us the ability to test many values to find quickly the one we want without wasting a lot of time. Zinc's spin control class, **UIW_SPIN_CONTROL**, is the window object equivalent of a dial that lets users flip through a range of values to find the one that works best.

A spin control instance displays the object's current value in a field, while two buttons allow the user to increment or decrement that value. Our spin control objects can use many Zinc window object classes, such as **UIW_BIGNUM**, **UIW_DATE**, **UIW_TIME**, and so forth, to contain that value. When instantiating a spin control object, we can tell the object to increment or decrement its value by certain amounts that we specify.

Below is an instance of a Zinc spin control object:

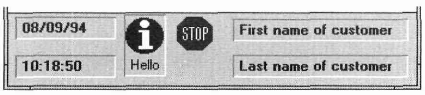


Status bar

Often, programs provide information about the status of some of its components—for example, a program might display status information like the current cursor location or the last key pressed. To make it easier for us to display status information in our programs, Zinc gave us a class called **UIW_STATUS_BAR**.

A Zinc status bar displays at the bottom of a window information about the status of information in our program. To display this information we attach time fields, date fields, number fields—anything that contains status information—to the status bar, in the same way that we'd attach a window object to a window.

Below is an instance of a Zinc status bar object:



String fields

A string is a set of characters upon which we can perform certain operations. In Zinc, a string field is an object that displays or accepts from a user as input a string, with or without special formatting, that takes up only one line in a field. We'll often manipulate strings in our programs, so using an existing Zinc class instead of writing our own will save us a lot of time and work.

Zinc provides two classes for working with string fields, **UIW_STRING** and **UIW_FORMATTED_STRING**. The **UIW_STRING** class allows us to display and to gather from the user string information, whereas the **UIW_FORMATTED_STRING** class does the same thing except it specifies a format for the data that is entered and displays the data in that format. For example, we would create a **UIW_STRING** field to accept the user's name. But we would create a **UIW_FORMATTED_STRING** if we wanted to accept the user's telephone number in the format (801) 785-8900, with parentheses and a dash in the appropriate places.

Below is an instance of a string field object.

When we want to work with a Zinc string field, we pass to the string field object special placeholder characters that represent how to format its encapsulated string information. Though the **UIW_STRING** and **UIW_FORMATTED_STRING** differ in how they format information, both classes share these placeholder characters along with common display styles. Here's a partial list of the display styles these string field classes share:

TABLE 4. **String-field display styles (partial list)**

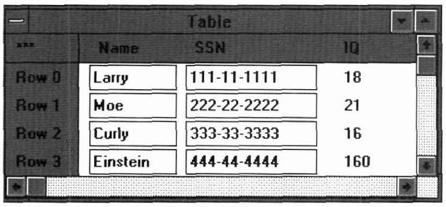
Lowercase	Displays string in lowercase, no matter what its original format.
Uppercase	Displays string in uppercase, no matter what its original format.
Spaces to underscores	Converts all spaces in the field to the underscore character.
Password-style	Doesn't echo characters as the user types in information.
Left justify	Displays string at the leftmost border of the field.

TABLE 4. **String-field display styles (partial list)**

Right justify	Displays string at the rightmost border of the field.
Center justify	Displays string in the center of the field.

Table

A table is used to present lists of information to the user. Often the information is comprised of multiple, related fields. The table can display headers to describe the contents of each row and column of data.



Table

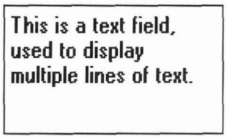
***	Name	SSN	IQ
Row 0	Larry	111-11-1111	18
Row 1	Moe	222-22-2222	21
Row 2	Curly	333-33-3333	16
Row 3	Einstein	444-44-4444	160

Text

Besides working with strings, manipulating text is one of the most common things we'll do in writing graphical applications, so using Zinc's text class will save us a lot of time and work. We can think of Zinc's text class as a multiline string field class, except that we can attach scroll bars to our text objects and that some of the custom display options can't be used with the text class.

Zinc's text class, **UIW_TEXT**, allows us to display and to gather from the user text information; with **UIW_TEXT** we can use many of the custom display styles of the **UIW_STRING** class, in addition to functionality specific to a text object, such as cursor movements. For example, Zinc text objects include the built-in capability for moving to the beginnings and ends of words, lines, and pages, in addition to scrolling up and down pages and wrapping words that extend beyond the boundaries.

Below is an instance of the **UIW_TEXT** class.


This is a text field,
used to display
multiple lines of text.

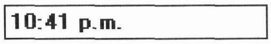
We should use **UIW_TEXT** objects for multiline text information, and use the **UIW_STRING** objects for single line information.

Additionally, when we instantiate a Zinc text object, we can use the *WOF_NON_FIELD_REGION* flag to cause our text to take up all the available space inside the window's border. For example, a help window always contains the basic window objects we discussed at the beginning of this chapter, as well as a **UIW_TEXT** field that dynamically fills the window.

Time

We can use Zinc's time field objects whenever we want to display time information or gather time information from the user. Time field objects, created using the **UIW_TIME** class, display time information and allow the user to enter and modify time information in many different international formats.

Below is an instance of the **UIW_TIME** class.



10:41 p.m.

The default behavior of a Zinc time field object is to display the time in a format native to the language and locale under which the program's running. However, with certain Zinc flags, we can override any language or localization information.

Here are the different styles Zinc's time field class supports, and a sample of how time styles look.

TABLE 5. Time styles

Colon separator.	Separates each time variable with a colon.	`12:00`
		`13:00:00`
		`12:00 a.m.`

Hundredths.	Includes hundredths value in the time, which otherwise is not included.	`1:05:00:00`
		`23:15:05:99`
		`7:45:59:00 a.m.`

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

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

Seconds.	Includes seconds value in the time, which by default is not included.	`8:09:30`
		`14:00:00`
		`3:24:59 p.m.`

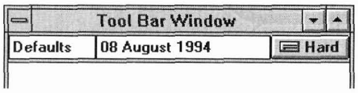
TABLE 5. Time styles

Twelve-hour clock.	Shows time using a 12-hour clock, regardless of the default information.	`12:00 a.m.` `1:00 p.m.` `5:00 p.m.`
Twenty-four hour clock.	Shows time using a 24-hour clock, regardless of the default information.	`12:00` `13:00` `17:00`
Uppercase.	Shows time in an upper-case format.	`12:00 P.M.` `1:00 A.M.` `7:00 P.M.`
Zero fill.	Fills hour, minute and second values with zeroes when times values are less than 10.	`01:10 a.m.` `13:05:03` `01:01 p.m.`

Tool bars

Tool bars display at the top of a window and are used to provide quick access to commonly used features. Tool bars are useful because, like pull-down menus, they provide an intuitive way to access functions associated with a specific window; but in providing a single button for accessing that function, they save mouse clicks and movements and therefore they save time and work. In Zinc, tool bars of the **UIW_TOOL_BAR** class can contain Zinc objects like icons, buttons with bitmaps, strings, combo boxes, and so forth.

Below is an instance of the **UIW_TOOL_BAR** class.



Other programmer-defined window objects

Any window object that comprises and conforms to the operating protocol defined by the **UI_WINDOW_OBJECT** base class.

Editing window objects

Users can edit certain window objects, notably **String**, **Formatted String**, **Text**, **Number**, **Date**, and **Time**. All editable window objects support the following features:

TABLE 6. **Features of editable window objects**

Mark	Marks part of the current field for cutting or copying. Marked regions are shown as shaded regions.
Cut	Cuts the marked contents of the current field and stores it in a paste buffer. This data can later be pasted into any other field, as long as the information is valid for that field type. For example, the text "400" could be pasted into a numeric, string or text field, but not in a check box.
Copy	Copies the marked contents of the current field and stores it in a paste buffer. This data can later be pasted into any other field, as long as the information is valid for that field type.
Paste	Copies the contents of the paste buffer into the current field. Data can be pasted into any field, as long as the information is valid for that field type.

Conclusion

In this chapter, we've learned about Zinc's window object classes, including what they do and how they work. Besides borders, titles, maximize buttons, minimize buttons, and system buttons, which are the most basic window objects in the Zinc library, we can use Zinc window object classes, all operating-environment independent, for accomplishing many things. These things include using dates and times with international formats, using pull-down menus and tool bars, offering selections in vertical and horizontal lists, displaying MDI windows, and manipulating strings and text.

In the next chapter we'll learn about issues of writing Zinc programs for multiple operating environments.

Chapter 4

Writing Multiplatform Programs

In the last chapter, we learned the contents of Zinc's window and window object classes, including what each does and how it works. In this chapter, we'll discuss how Zinc enables us to write programs for multiple operating environments.

Key Concepts

multiplatform application design

special considerations of each environment

About multiplatform programming in Zinc

Single source

Writing a Zinc program for multiple operating environments requires only one set of source code. This is an important benefit of Zinc. Since we only need to write one program for all our operating environments, we don't have to juggle multiple sets of source code, making multiplatform development easier.

Engines and keys

Zinc consists of two parts, the Engine and the Key. With the Engine and the appropriate key, we can compile DOS text, DOS graphics, Windows and Windows NT, OS/2, Macintosh, Motif, Curses, and NEXTSTEP programs from the same set of source code.

The Engine includes all of Zinc's code that is independent of specific operating environments. It also includes collateral such as this manual. The Key includes precompiled libraries for our target operating environment, the source code for the display class, and Zinc Designer.

Look and feel

An important Zinc goal is to allow our programs to look and feel native to the environment for which they were compiled—for example, Zinc wants our DOS applications to look and feel like DOS applications, our OS/2 applications to look and feel like OS/2 applications, and so forth. Zinc wrote its libraries with windows and window objects for each operating environment it supports. This way, we don't have to know the low-level details of each environment, but can still access them directly if we wish. This means our Zinc programs will look and feel native to our target environments because they *are* native—and users will accept our programs without a second thought.

Libraries

Zinc's source code for windows, window objects, and event handling for each operating environment lives in certain library files, named for specific things in each environment. For example, the DOS libraries are called **DOS_ZIL.LIB**. This is the file we must link into the executable if we want to write Zinc applications for DOS. For a complete list of all the library files Zinc includes, consult "Appendix A, Compiler Considerations."

Compiler options

When writing a Zinc program for a target operating environment, pay special attention to the following compiler options:

Application type. If your compiler can compile executables for multiple environments, select the compiler option to create the application as an executable for the target environment.

Memory model. If you are building an application for an operating system that supports multiple memory models, you must use the large memory model since this is the only model Zinc supports in those environments.

Main( )

Ordinary C++ programs call **main( )** as their first function, and Zinc programs are no different. However, in Zinc we can create the **main( )** function in two ways. The first is to create the **Main( )** function in the Zinc class **UI_APPLICATION**. This class provides our programs with a **main( )** or **WinMain( )** function, depending on whether our target environment is DOS or Windows; this class also initializes the display, Event Manager, and Window Manager. The second way is to write the **main( )** function ourselves and initialize the display, Event Manager, and Window Manager by hand.

Zinc designed the **UI_APPLICATION** class to handle much of the work of setting up the infrastructure needed to run a Zinc program under multiple operating environments. This infrastructure includes the display, the Event Manager, and the Window Manager. We recommend that you use **UI_APPLICATION::Main( )** wherever possible to set up that infrastructure.

Event handling

In Zinc, each window object contains an **Event( )** function that processes messages as appropriate for the target operating environment. We can classify event handling into two types: *top down* and *bottom up.*

In top down environments, the Event Manager receives events from input devices such as the keyboard and mouse, which it places in the event queue. Then the main event loop takes each event from the queue and dispatches the event to the Window Manager, which processes the event with its own **Event( )** function, and determines whether or not it can respond to the event. If the Window Manager can, it performs an action and passes control back to the main event loop; but if the Window Manager cannot, it passes the event to the current window, which then processes the event with its own **Event( )** function. If it can, the window performs an action, but if it cannot, it passes it to the current window object, which responds to the event.

In bottom-up environments, the operating environment receives events from input devices such as the keyboard and mouse, and processes the event in a *black box*; inside the black box, the operating environment determines which object the event is supposed to go to. When the system processes the event, it dispatches the event to the current window object, which then determines whether or not it can respond to the event. If the window object can, it performs an action and returns control to the operating system; but if it cannot, it may pass the event to its parent window for processing. Because the events pass from the bottom, the current window object, to the top this type of event handling is called bottom up.

When writing Zinc programs for different operating environments, be sure to take into account how each environment processes events, because if we write a Zinc program to deploy on DOS and Windows, each environment handles events differently than the other. For example, if we write a Zinc program for DOS that traps keyboard events, no matter what window object is current, we might create a window that traps events since all events go through the window. This does not hold true for Windows, so if we run our program under Windows, the window will only trap messages if no other window objects are current. Be sure to take into account event handling for each target operating environment, so that you can write your programs to handle events properly.

Executable naming conventions

So you can easily identify the environment for which you've compiled your executable, Zinc maintains the following naming conventions for executables:

TABLE 7. **Naming conventions for executables**

Environment	Convention
DOS	hello1
DOS 16-bit	hello116
DOS 32-bit	hello132
Macintosh	Hello1
NEXTSTEP	hello1
OS/2	ohello1
OSF/Motif	hello1
Windows 3.x	whello1
Windows NT	nhello1

Shipping applications

Be sure to include the following run-time files when you ship your finished applications:

- **.DAT** files (generated by Zinc Designer) required by your applications.
- **I18N.DAT** required by globalized applications.
- **UNICODE.FNT** required by double-byte (Unicode) applications running in DOS graphics mode.

YOU MAY NOT INCORPORATE INTO YOUR APPLICATION OR DISTRIBUTE AS PART OF YOUR APPLICATION ANY PORTION OF ZINC DESIGNER WITHOUT THE EXPRESS WRITTEN PERMISSION OF ZINC.

DOS

Look and feel

In DOS, a Zinc application follows IBM's SAA/CUA specification for the display and input devices. Using Zinc libraries, we can compile Zinc programs that run in DOS text and graphics, in both real and protected modes.

DOS libraries

The DOS version of Zinc has been compiled into a single library file called **DOS_ZIL.LIB**. When creating a DOS application, we must link **DOS_ZIL.LIB,** and, if our program is designed to run in DOS graphics mode, the appropriate graphics display class library as well, into the **.EXE** file.

Compiler options

When creating a DOS application, select the following compiler options:

DOS program. If your compiler can compile executables for other environments in addition to DOS, select the compiler option to create the application as a DOS executable program.

Large model. Set the compiler to the large memory model. Since Zinc only uses the large memory model, we must ship all our applications with the large memory model.

See “Appendix A—Compiler Considerations” for more information regarding compiler-specific options.

main( )

Ordinary C++ programs begin with calling **main( )** as the first function. Zinc-based applications for DOS are no different. We may create the **main( )** function in our DOS programs by using the **UI_APPLICATION** class, which contains a **main( )** function, and also initializes the display, Event Manager, and Window Manager. Or we may create our own **main( )** function and initialize the display, Event Manager, and Window Manager by hand.

Windows

Look and feel

In Windows, a Zinc application is an actual Windows application built with actual Windows objects. When writing Zinc programs, we have full access to the Windows API and Windows resources, including writing Win32 applications that run under the Win32s extensions for Windows 3.1 and Windows NT.

Windows libraries

The Windows version of Zinc has been compiled into a single library file called **WIN_ZIL.LIB**, and a Windows NT library file called **WNT-_ZIL.LIB**. When creating a Windows application, we must link **WIN-_ZIL.LIB**, or, if we're compiling a program for Windows NT, **WNT_ZIL.LIB**, into the **.EXE** file.

Compiler options

When creating a Windows application, be sure to select the following compiler options:

Windows application. If your compiler can compile applications for other environments in addition to Windows or Windows NT, select the compiler option to compile the program into a Windows or Windows NT executable.

Large model. Set the compiler option to compile using the large memory model. Since Zinc ships only with the large memory model, all Windows programs must also use the large memory model.

See "Appendix A—Compiler Considerations" for more information regarding compiler-specific options.

WinMain( )

Ordinary C++ programs begin with **main( )** as the first function. However, when writing Zinc programs for Windows or Windows NT, we create instead a function called **WinMain( )**, which Windows uses to begin executing an application. Here is the definition of **WinMain( )**:

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

We can create the **WinMain( )** function two ways. The first is to use **UI_APPLICATION::Main( )**, which contains the **WinMain( )** function, and also initializes the display, Event Manager, and Window Manager. Zinc recommends using **UI_APPLICATION::Main( )** to promote portability between operating environments and to ease program design.

The second way is to create the **WinMain( )** function in our program and initialize the display, Event Manager, and Window Manager by hand. The following code sample demonstrates this technique:

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

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

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

  return (0);
}
```

OS/2

Look and feel

In OS/2, a Zinc program is an actual OS/2 application built with actual OS/2 objects. When writing an OS/2 application, we have full access to the OS/2 API and OS/2 resources.

OS/2 library

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

main( )

Ordinary C++ programs begin with calling **main( )** as the first function. Zinc-based applications for OS/2 are no different. We may create the **main( )** function in our OS/2 programs by using the **UI_APPLICATION** class, which contains a **main( )** function, and also initializes the display, Event Manager, and Window Manager. Or we may create our own **main( )** function and initialize the display, Event Manager, and Window Manager by hand.

Macintosh

Look and feel

In Macintosh, a Zinc program is an actual Macintosh application built with actual Macintosh objects. When writing a Macintosh application, we have full access to the Macintosh Toolbox and Macintosh resources.

Macintosh libraries

The Macintosh version of Zinc has been compiled into several library files, listed in the table below.

TABLE 8. **Macintosh library files**

Library	*Function*
Mac_ZIL1	Global code and international classes
Mac_ZIL2	Device classes
Mac_ZIL3	Window Manager and window object classes
Mac_ZIL4	**UIW_WINDOW** classes
Mac_ZIL5	Window support classes

TABLE 8. Macintosh library files

Library	*Function*
Mac_ZIL6	Date and time classes
Mac_ZIL7	String and text classes
Mac_ZIL8	Window-derived classes
Mac_ZIL9	Button-derived and table classes
Mac_ZIL10	Number classes
UI_JumpTables	Global object table
UI_Application	**UI_APPLICATION** code
ZIL_Storage	Storage code
Mac_ZIL.rsrc	Macintosh-specific code

When creating a Macintosh application, we must link some or all of these libraries into the **.EXE** file, depending on what functionality we need in our application. For example, if we wanted to use the **UI_APPLICATION** class in our Macintosh program, we would include the library **UI_Application**.

main( )

Ordinary C++ programs begin with calling **main( )** as the first function. Zinc-based applications for Macintosh are no different. We may create the **main( )** function in our Macintosh programs by using the **UI_APPLICATION** class, which contains a **main( )** function, and also initializes the display, Event Manager, and Window Manager. Or we may create our own **main( )** function and initialize them by hand.

OSF/Motif

Look and feel

In OSF/Motif, a Zinc application is an actual OSF/Motif application built with actual OSF/Motif widgets. When writing Zinc programs, we have full access to the OSF/Motif toolkit, Xt Intrinsics, X Library, and all X resources.

OSF/Motif libraries

The OSF/Motif version of Zinc has been compiled into a single library file called **lib_mtf_zil.a**. When writing an OSF/Motif program, we must link **lib_mtf_zil.a**, as well as **libXm.{a, so, sl}**, **libXt**, **libX11**, and the Xm library, into the executable file. We may have to change some source code to use the OSF/Motif Key on hardware platforms not directly supported by Zinc. See the **README** file for a list of currently supported hardware platforms.

main( )

Ordinary C++ programs begin with calling **main( )** as the first function. Zinc-based applications for OSF/Motif are no different. However, the **main( )** function for OSF/Motif does require the standard *argc* and *argv* parameters. When the OSF/Motif display is created, these parameters are passed to the Xt Intrinsic initialization routines, which allow Zinc applications to use X command-line options, such as other displays, colors, fonts, and so forth.

There are two ways to implement the **main( )** function in our OSF/Motif programs. The first is to use the **UI_APPLICATION** class, which provides the **main( )** function, and also initializes the display, Event Manager, and Window Manager. Or we may create our own **main( )** function and initialize the display, Event Manager, and Window Manager by hand.

Shipping applications

In addition to the files specified at the beginning of this chapter, be sure to include this additional run-time file when you ship your finished Motif applications:

- **ZincApp.ad**, which provides your Zinc applications with defaults.

Curses

Look and feel

In Curses, a Zinc application uses the Curses library to perform terminal screen I/O.

Curses libraries

The Unix Curses version of Zinc has been compiled into a single library file called **lib_crs_zil.a**. When writing a Curses program, we must link **lib_crs_zil.a** into the executable file. We may have to change some source code to use the Curses Key on hardware platforms not directly supported by Zinc. See the **README** file for a list of currently supported hardware platforms.

main( )

Ordinary C++ programs begin with calling **main( )** as the first function. Zinc applications for Curses are no different.

There are two ways to implement the **main( )** function in our Unix Curses programs. The first is to use the **UI_APPLICATION** class, which provides the **main( )** function, and also initializes the display, Event Manager, and Window Manager. Or we may create our own **main( )** function and initialize the display, Event Manager, and Window Manager by hand.

NEXTSTEP

Look and feel

In NEXTSTEP, a Zinc program is an actual NEXTSTEP application built with actual NEXTSTEP objects. When writing a NEXTSTEP application, we have full access to NEXTSTEP and its resources, with the exceptions of drag and drop and object linking.

NEXTSTEP library

The NEXTSTEP version of Zinc has been compiled into a single library file called **lib_nxt_zil.a**. When creating a NEXTSTEP application, we must link **lib_nxt_zil.a** into the executable.

main( )

Ordinary C++ programs begin with calling **main( )** as the first function. Zinc-based applications for NEXTSTEP are no different. We may create the **main( )** function in our NEXTSTEP programs by using the **UI_APPLICATION** class, which contains a **main( )** function, and also initializes the display, Event Manager, and Window Manager. Or we may create our own **main( )** function and initialize the display, Event Manager, and Window Manager by hand.

Event handling

A Zinc window object running under NEXTSTEP contains an **Event( )** function that processes messages using NEXTSTEP responder methods such as **-mouseDown**, as well as delegate methods for classes such as **Window**.

Conclusion

In this chapter, we discussed how Zinc enables us to write programs for multiple operating environments. Since different operating environments require different **main( )** functions, writing programs for multiple operating environments can be eased with **UI_APPLICATION::Main( )**. Each operating environment requires that we use certain libraries, and that we take into account differences in event handling between environments.

In the next chapter, we'll discuss event handling in greater detail and explain more how top-down and bottom-up event handling works.

Chapter 5

Event Flow and Mapping

In the last chapter, we discussed how Zinc enables us to write programs for multiple operating environments. In this chapter, we'll discuss how events flow through the system and how Zinc maps events. As stated earlier, Zinc programs are event driven, which means that at their core they contain a main event loop which spins in the background, catching events and dispatching them to the appropriate places. In Zinc, each window object contains an **Event( )** function that handles events as appropriate for the target operating environment. And each environment may handle events in a top-down or bottom-up manner. What follows is a discussion of how this works.

Key Concepts

top-down and bottom-up event handling

event map tables

palette mapping

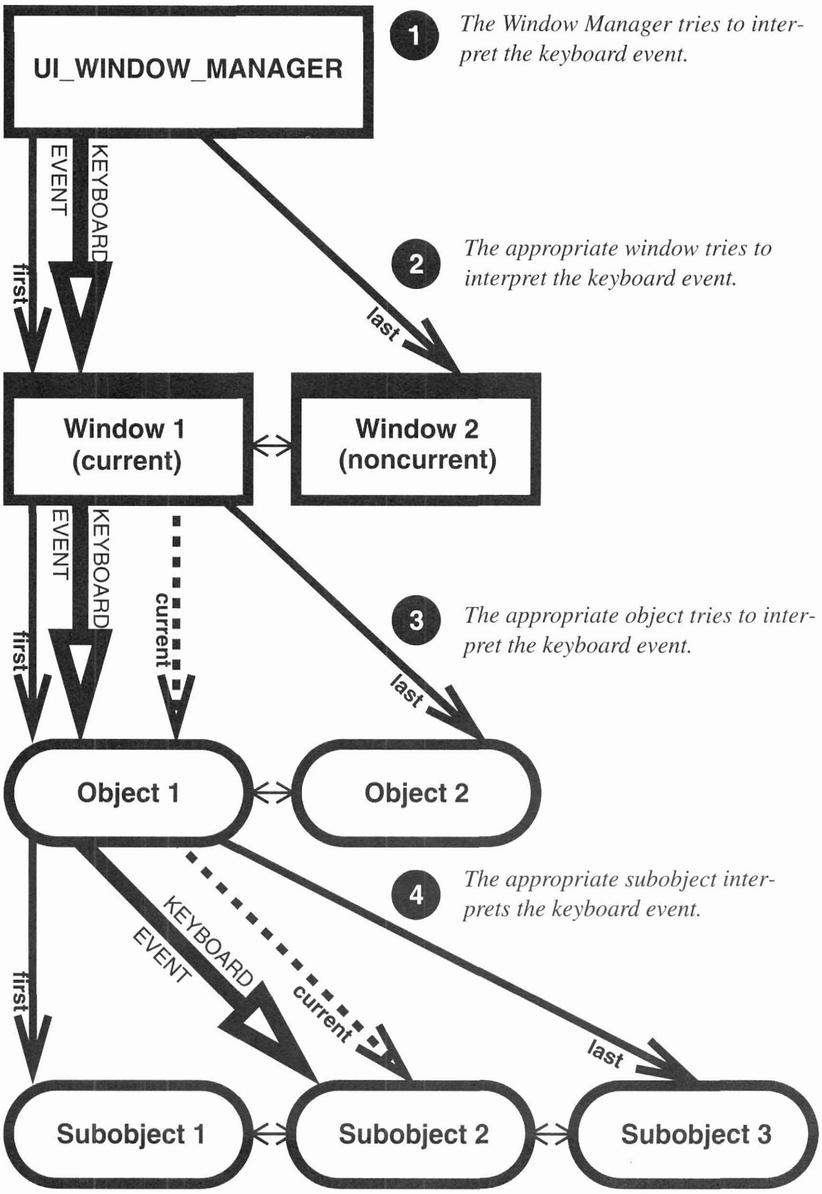
Top down

In top down environments, the Event Manager receives events from input devices such as the keyboard, mouse, and perhaps the operating environment, and places the events in the event queue. The main event loop takes the event from the queue and sends it to the Window Manager, which processes it with its own **Event( )** function. Then the object determines whether or not it can respond to the event. If it can, it performs an action and returns control to the main event loop; but if it cannot, it passes the event to the current window, which processes the event with its own **Event( )** function. If it can, the window performs an action, but if it cannot, it passes it to the current window object, which responds to the event. This continues until an object processes the event or when the event comes to an object with no children.

The following diagram represents event flow in a program running under a top down environment. The program contains two windows; the current window contains a **UIW_GROUP** object, which in turn contains several checkbox objects, while the noncurrent window contains no objects. The Window Manager, the current window, and the current object maintain three pointers, *first*, *current*, and *last*, which are the first object, the current subobject, and the last object below each.

Here's how events flow when the user presses a key.

1. First, the keyboard press sends an event to the Window Manager, which tries to interpret the event and fails. It then passes the event to the current window.
2. The window tries to interpret the event and fails. It then passes the event to the **UIW_GROUP** object since it is the current object on the window.
3. The **UIW_GROUP** object tries to interpret the event and fails. It then passes the event to the checkbox object that is current in the group.
4. The checkbox tries to interpret the event by looking in the event map table. If the event maps into an event it can process, it does so.
5. Then the subobject returns a control code (not shown) indicating whether or not it processed the event.



1 *The Window Manager tries to interpret the keyboard event.*

2 *The appropriate window tries to interpret the keyboard event.*

3 *The appropriate object tries to interpret the keyboard event.*

4 *The appropriate subobject interprets the keyboard event.*

Bottom up

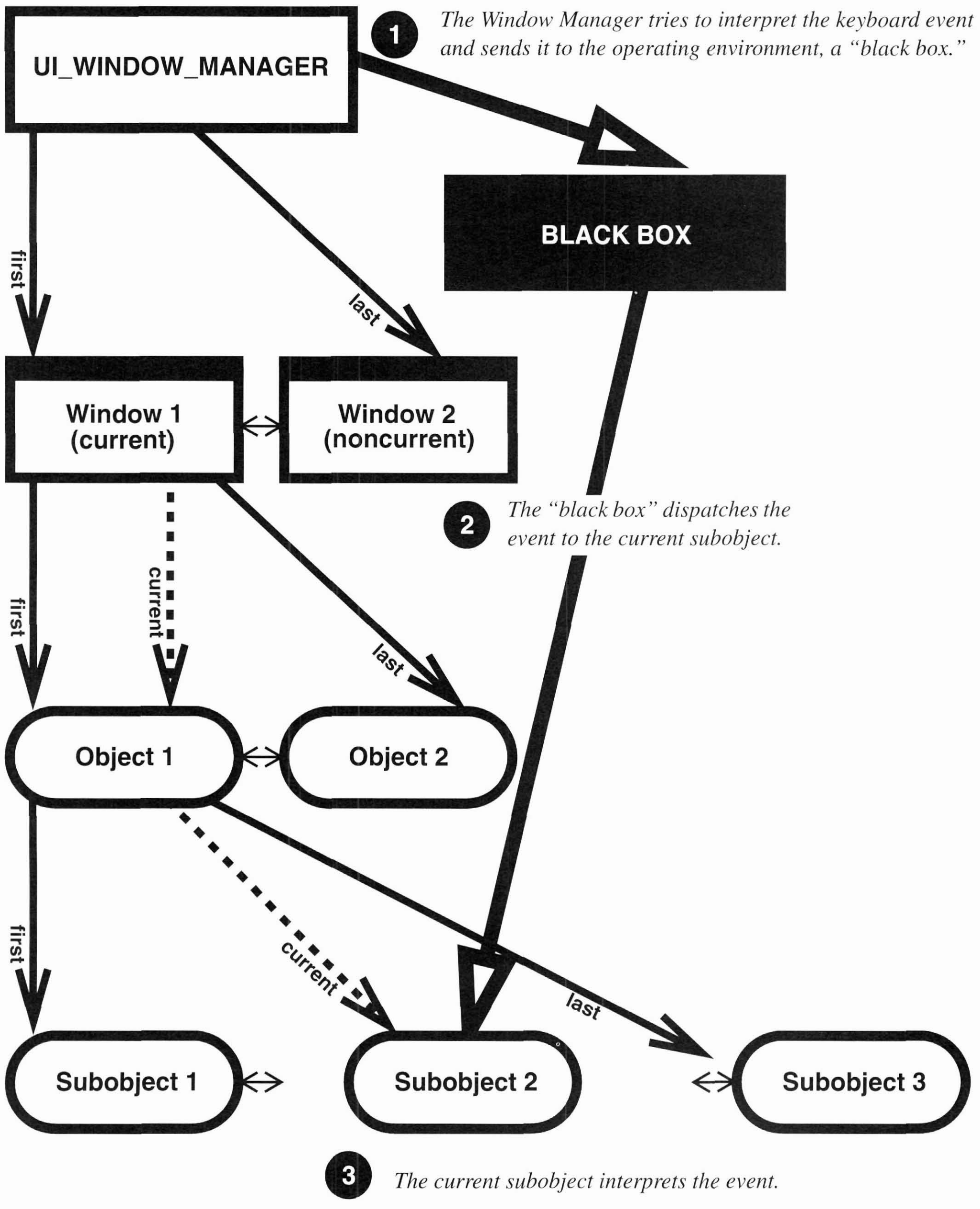
In bottom up environments, although the operating environment processes events from input devices such as the keyboard and mouse that are related to the system, the Event Manager still receives the input first, turns it into a Zinc event, and hands it to the Window Manager.

If the event isn't a native event, the event will flow from top to bottom as it does in a top down environment. But when it is a native event, the Window Manager hands the event to the system. When the system processes the event, it sends it to the current low level window object, which determines whether it will respond to the event. If it responds to the event, it returns control to the operating system; but if it doesn't, it may pass the event to its parent window, which then may process the event with its own **Event( )** function. Because the events pass from the current window object on the bottom, to the top, this type of event handling is called bottom up.

The following diagram represents event flow in a program running under a bottom up environment. Again, the program contains two windows; the current window contains a **UIW_GROUP** object, which in turn contains several checkbox objects, while the noncurrent window contains no objects.

Here's how events flow when the user presses a key.

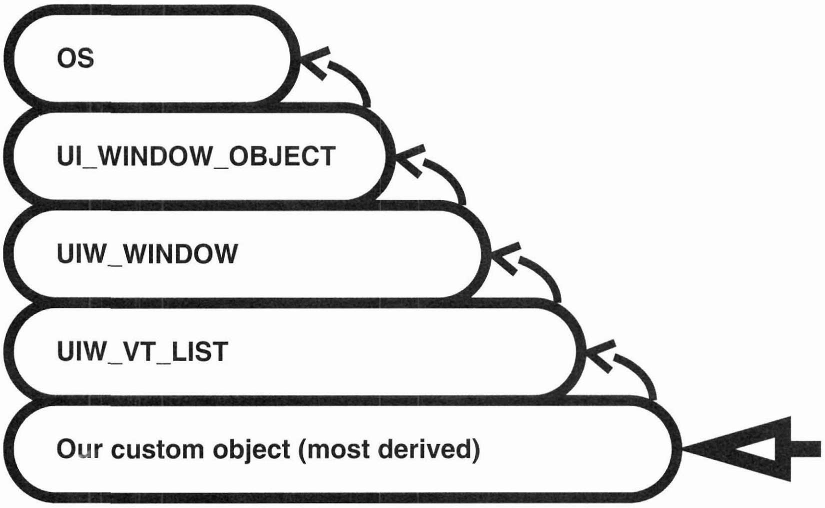
1. First, the key press causes the Event Manager to send an event to the Window Manager, which tries to interpret the event and fails. It then passes the event to the operating environment.
2. The operating environment, in a black box, sends the object to the current subobject.
3. The checkbox tries to interpret the event by looking in the event map table. If the event maps into an event it can process, it does so.


UI_WINDOW_MANAGER
1
The Window Manager tries to interpret the keyboard event and sends it to the operating environment, a "black box."
BLACK BOX
first
last
Window 1 (current)
Window 2 (noncurrent)
2
The "black box" dispatches the event to the current subobject.
first
current
last
Object 1
Object 2
first
current
last
Subobject 1
Subobject 2
Subobject 3
3
The current subobject interprets the event.

When writing Zinc programs for different operating environments, be sure to take into account how each environment processes events, because if we write a Zinc program to deploy on DOS and Windows, both environments handle events different from each other. For example, if we write a Zinc program for DOS that traps keyboard events, no matter what window object is current, the window itself gets events. This does not hold true for Windows, so if we run our program under Windows, the window will only process events if the operating environment thinks it ought to get them. Be sure to take into account event handling for each target operating environment, so that you can wrtite your programs to handle events properly.

Event processing

Here's how events get processed in Zinc. When either the Window Manager or the black box dispatches an event to an object, C++ ensures that it gets sent to the most derived object. Notice in the following diagram that the most derived object, the one that we derive from an existing Zinc object, receives the event first. If our object's **Event( )** function can't process the event, it should send it to the next most derived object, and so on. The benefit is that we can extend Zinc—because we know that our object can receive an event before any other predefined object, we can add custom functionality to our objects that override Zinc functionality.



Here's what should happens when derived objects process events:

1. Our custom object receives an event for processing.
2. If the most derived object, our custom object, cannot process that event, it passes it up to the next most derived object, **UIW_VT_LIST**, for processing.
3. If **UIW_VT_LIST** cannot process that event, it passes it up to the next most derived object, **UIW_WINDOW**, for processing.
4. If **UIW_WINDOW** cannot process that event, it passes it up to the next most derived object, **UI_WINDOW_OBJECT**, for processing.
5. If **UI_WINDOW_OBJECT** cannot process that event, it passes it up to the operating system for processing (if applicable), or returns a control code indicating it could not process the event.

Event map table

In Zinc, event map tables list important events that input devices can send, and how Zinc objects interpret those events. Zinc's event mapping conforms to the key assignments of each operating environment's specifications. For example, a Zinc application running under Windows would conform to IBM's Common User Access Panel Design and User Interaction specification. And a Zinc application running under NEXTSTEP would conform to NeXT's user interface guidelines.

Here's how map tables work. The following portions of *eventMapTable*, which is a static table accessed by **UI_WINDOW_OBJECT::***eventMapTable*, define how a window object interprets events generated by the keyboard and mouse:

```
static UI_EVENT_MAP eventMapTable[] =
{
  { ID_WINDOW_OBJECT,L_NEXT,E_KEY,TAB },
  { ID_WINDOW_OBJECT,L_PREVIOUS,E_KEY,BACKTAB },
  { ID_WINDOW_OBJECT,L_SELECT,E_KEY,ENTER },
  ...
  { ID_WINDOW_OBJECT,L_CONTINUE_SELECT,E_MOUSE,M_LEFT },
  ...
  // End of array.
  { ID_END, 0, 0, 0 }
};
```

An event map table entry is composed of the identification for the type of object, the logical event, the device type that produced the message, and the raw scan code of the event. In our example, a window object will process an *L_NEXT* message when a user presses the <Tab> key.

Not only does Zinc's event mapping allow different devices to generate the same logical message, but it also allows different objects to interpret the same event in different ways. This is a strong benefit to programming in Zinc. Because each object can respond differently to events, we don't have to write code to decipher how each object should behave in context of our program; we need only tell the object to perform a method appropriate to how it operates.

For example, the following portion of an event map table defines how a string object will interpret events.

```
{ ID_STRING,L_BEGIN_MARK,E_MOUSE,M_LEFT | M_LEFT_CHANGE},
{ ID_STRING,L_CONTINUE_MARK,E_MOUSE,M_LEFT},
{ ID_STRING,L_END_MARK,E_MOUSE,M_LEFT_CHANGE},
```

A string interprets a click on the left mouse button as a mark operation instead of a select operation. If a string object couldn't respond differently, it would have to override the select operation in order to set the mark operation, causing us to write more code than necessary.

Event mapping algorithm

When the object receives an event, the mapping algorithm walks through the map table and searches for the best match according to the object's and the device's identification, the raw scan code, and the input modifier, usually the keyboard shift state, associated with the event. For example, if the user presses the left mouse button while the cursor is positioned in a string object, the application will scan the map table until the best possible match is found, shown below:

```
{ ID_STRING,L_BEGIN_MARK,E_MOUSE,M_LEFT | M_LEFT_CHANGE }
```

As a result, the mark operation will begin within the string object. When the application interprets the *L_END_MARK* logical message, the mark operation will be finished.

Palette mapping

Zinc uses palette mapping to provide a way for objects to paint themselves when in different states. Palette mapping takes the state of an object and gives it a palette to use to paint itself.

UI_PALETTE, the Zinc palette class, is the set of colors an object uses when drawing itself; the colors it uses depends on the mode of the display, such as color text mode, color graphics mode, mono text mode, mono graphics mode, and so forth. An object gets a palette when it draws itself. We can describe a palette in terms of its graphics mode and foreground and background color; for example, a palette may contain a red foreground and a blue background for color graphics mode.

UI_PALETTE_MAP contains an object ID, such as *ID_WINDOW_OBJECT, ID_WINDOW*, or *ID_LIST_ITEM*; a *logicalPalette*, such as *PM_ACTIVE*, *PM_SELECTED*, *PM_CURRENT*, *PM_ANY*; and the corresponding **UI_PALETTE**.

A palette map table is a lookup table that is an array of **UI_PALETTE_MAP**s.

When a *WOS_OWNERDRAW* object should draw itself, Zinc calls its **DrawItem( )** function. The control code, passed to the **DrawItem( )** function, tells the object why it should draw—for example, it may receive an *S_CURRENT* control code. The object uses the control code when calling the Zinc **LogicalPalette( )** function, which will look at the control code and the current state of the object in *woStatus*, such as active, current, inactive, selected, and so forth. **LogicalPalette( )** will use the control code and current status to come up with a logical palette, determined by ORing together *PM_* flags.

LogicalPalette( ) will call **UI_PALETTE_MAP::MapPalette( )**, passing in the object's palette map table, the LogicalPalette determined above, and five IDs, which are found in *windowID*. **MapPalette( )** searches the palette map table, comparing IDs and the logical palette to find the appropriate **UI_PALETTE**. This **UI_PALETTE** is used when calling the display's drawing functions.

Most graphics libraries have special ways of using colors, and to make it easier for us to let us use the colors we want in our Zinc programs, Zinc provided concepts called palettes, palette maps, and palette map tables. For example, 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. Below is how this works:

1. Call the **MapColor( )** function with two parameters, *palette*, a pointer to a **UI_PALETTE** class, and *foreground*, which tells us whether we want the foreground or background color.

```
COLOR UI_BGI_DISPLAY::MapColor(const UI_PALETTE *palette,
  int foreground)
{
```

2. Next, we determine the type of display our program is running in, and get the appropriate number of colors from the palette.

```
  // 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);
```

Whenever a window object draws information on the screen, it must map the map logical values into Zinc values. To do so, it uses **UI_WINDOW_OBJECT::MapPalette( )** to get the palette from the system. **MapPalette( )** then uses a specified *mapTable* to match the Zinc value to a system palette. Zinc uses three predefined map tables for palettes called *normalPaletteMapTable*, *helpPaletteMapTable,* and *errorPaletteMapTable*. All window objects use *normalPaletteMapTable*, the **UI_HELP_SYSTEM** window uses *helpPaletteMapTable*, and the **UI_ERROR_SYSTEM** window uses the *errorPaletteMapTable*.

Conclusion

In this chapter, we've discussed how events flow through the system, and how Zinc maps events and palettes. In the next chapter, we'll learn about Zinc's library classes, and how they provide a kind of periodic table of objects with which we can build new objects.

Chapter 6

Library Classes

In the last chapter, we discussed how events flow and how Zinc maps events. In this chapter, we'll learn about what Zinc calls its library classes. Library classes are the molecules and elements that make up Zinc programs.

Some of Zinc's library classes contain properties and behaviors that are so basic they cannot be reduced—these are the Zinc elements. Others, however, are comprised of other Zinc library classes—these are the Zinc molecules that combine Zinc elements to create entirely new properties and behaviors. For example, lists and list elements are the smallest units of Zinc that contain

Key Concepts

- base classes
- region lists
- display classes

its own properties and behaviors, whereas the Event Manager and Window Manager consist of lists and list elements. Here's a table that describes Zinc's library classes.

TABLE 9. Zinc's library classes

Base classes	Lists and list elements. Most Zinc components are made up of these base classes
Event Manager	Input devices, the Event Manager, and their support classes
Window Manager	All Zinc window objects, the Window Manager, and their support classes
Help system	Context-sensitive help displayed in a Zinc window
Error system	Run-time errors displayed in a modal dialog box
Screen display	Low-level screen functions, which include managing screen regions

Base classes—Zinc's periodic table

Zinc contains two base classes: **UI_ELEMENT** and **UI_LIST**. Zinc calls **UI_ELEMENT** and **UI_LIST** base classes because they do not derive from other classes. In fact, we can think of Zinc's base classes like a periodic table of objects that consists of two elements. Below is the definition of these two classes and their public and protected members:

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

class EXPORT UI_LIST
{
  friend class EXPORT UI_LIST_BLOCK;
public:
  int (*compareFunction)(void *element1, void *element2);
  UI_LIST(int (*_compareFunction)(void *element1, void
    *element2) = NULL);
  virtual ~UI_LIST(void);
  UI_ELEMENT *Add(UI_ELEMENT *newElement);
  UI_ELEMENT *Add(UI_ELEMENT *element, UI_ELEMENT *newElement);
  int Count(void);
  UI_ELEMENT *Current(void);
  virtual void Destroy(void);
  UI_ELEMENT *First(void);
  UI_ELEMENT *Get(int index);
  UI_ELEMENT *Get(int (*findFunction)(void *element1, void *matchData),
    void *matchData);
  int Index(UI_ELEMENT const *element);
  UI_ELEMENT *Last(void);
  void SetCurrent(UI_ELEMENT *element);
  void Sort(void);
  UI_ELEMENT *Subtract(UI_ELEMENT *element);
  UI_LIST &operator+(UI_ELEMENT *element);
  UI_LIST &operator-(UI_ELEMENT *element);
protected:
  UI_ELEMENT *first, *last, *current;
};
```

The Event Manager has two main classes: **UI_DEVICE** and **UI_EVENT_-MANAGER**. The **UI_DEVICE** class derives from **UI_ELEMENT** and is used to define the operation of input devices. Its derivation from **UI_ELEMENT** allows other classes to be grouped together, in the form of a list. Since the **UI_EVENT_MANAGER** class derives from **UI_LIST**, it is able to maintain a list of all attached devices. This derivation also allows the Event Manager to control the operation and flow of event information from the input devices.

The Window Manager has three major classes: **UI_WINDOW_OBJECT**, **UI_WINDOW_MANAGER,** and **UIW_WINDOW**. The **UI_WINDOW-_OBJECT** class derives from **UI_ELEMENT**, and serves as the base class for all window objects, such as buttons, icons, and menu items. Because **UI_WINDOW_OBJECT** derives from **UI_ELEMENT,** we can combine window objects inside a parent window. Similarly, because **UI_WINDOW_MANAGER** derives from **UIW_WINDOW**, it can group window objects in a list.

The **UIW_WINDOW** class is unique because it acts like an element when attached to the Window Manager, and it acts like a list because it contains window objects such as a border, title bar, and so forth. Appropriately, this class derives from both the **UI_ELEMENT** base class through the **UI_WINDOW_OBJECT** class and the **UI_LIST** base class.

We've been discussing two base classes, **UI_ELEMENT** and **UI_LIST**. Technically, however, Zinc has a third base class called **UI_DISPLAY**, which provides to all of Zinc's displays some basic behaviors and draw functions. But we will use this class only when deriving a display class, so we'll spend most of our time talking about the other base classes.

UI_ELEMENT

The **UI_ELEMENT** class defines an element by what it can do, which is point to other elements directly before or after it in a list. It's meaningless to create an instance of **UI_ELEMENT**, because the class merely describes the basics of what elements can do, rather than describing more specialized things, such as collecting input from users or displaying themselves on screen. These things are left to classes such as input devices and window objects that derive from **UI_ELEMENT** and thereby inherit the basic behavior of elements and then add more specialized behavior. We'll explain more of what's going on under the hood in the next chapter when we discuss abstract classes.

The **UI_ELEMENT** class has two member functions, **Previous( )** and **Next( )**, which allow an element to point to the element directly before or after it in a list. Here's an example of how this works. The following code adds three input devices, a keyboard, mouse, and cursor to the Event Manager object, which we'll discuss later in this chapter.

```
eventManager->Add(keyboard);
eventManager->Add(mouse);
eventManager->Add(cursor);
```

If the mouse were the current object, **Previous( )** would return a pointer to the keyboard, whereas a call to **Next( )** would return a pointer to the cursor.

UI_LIST

The **UI_LIST** class defines a list by what it can do, which is contain elements. While you can create an instance of **UI_LIST**, it usually doesn't make much sense because the class merely describes the basics of what lists can do, rather than the more specialized things like receiving and responding to input devices or display a collection of windows and window objects on the screen. These things are left to objects such as the Event Manager and the Window Manager that derive from **UI_LIST**, which inherit the basic behavior of lists and then add some more specialized behavior.

The **UI_LIST** class has four member functions, **First( )**, **Last( )**, **Add( ),** and **Subtract( )**, as well as + and -, which are overloaded operators that allow us to add and delete elements to and from the list without using the corresponding functions. Predictably, the **First( )** and **Last( )** member functions retrieve the first or last element in the list. For example, **First( )** would return a pointer to the keyboard object, and **Last( )** would return a pointer to the cursor.

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

```
eventManager->Add(keyboard);
eventManager->Add(mouse);
eventManager->Add(cursor);
```

or

```
*eventManager
  + keyboard
  + mouse
  + cursor;
```

Event Manager

We introduced the Event Manager in "Introduction to Zinc" on page 11, where we described it as Zinc's infrastructure for handling events and system messages. Now we can elaborate by saying **UI_EVENT_MANAGER**, the main class of the Event Manager portion of Zinc, uses the list functions of **UI_LIST** and adds a *queueBlock* member variable to store events.

Input devices

The Event Manager's **UI_LIST** contains input devices, such as keyboards and mouses, that collect events as the user works with the application. Zinc defines how these input devices work in classes called **UID_KEYBOARD** and **UID_MOUSE**, which derive from **UI_DEVICE**. **UI_DEVICE** is an abstract class that defines the structure of input devices and how they work, but which must be derived from. The **UI_DEVICE** class derives from the **UI_ELEMENT,** which allows us to add input devices to the Event Manager's list of input devices, and contains virtual member functions not present in **UI_ELEMENT** called **Event( )** and **Poll( )**, which control how input devices operate. These functions also allow input devices to place events in the event queue, which we'll discuss later in this section.

We can use the **Event( )** function to send a message to an input device to change its behavior. Zinc applications pass this message in *event.type*. Here are some sample messages we can send to input devices:

D_OFF. Tells the device to stop placing events into the Event Manager's event queue. It will send no further input information until a *D_ON* message is received.

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

DM_WAIT. Changes the mouse pointer to an hourglass. The mouse is the only input device that uses this message.

Where the **Event( )** function controls how input devices operate, the **Poll( )** function allows each device to place events in the Event Manager's event queue. For example, the **UID_KEYBOARD** class uses the **Poll( )** function to check if the user has pressed any keys. If so, the **Poll( )** function places the resulting event in the Event Manager's event queue.

The event queue

Just as the Event Manager derives from **UI_LIST** and adds additional behavior, so does the event queue, a member variable of type **UI_QUEUE_BLOCK**, which we'll discuss in just a moment. The *queue-Block* member variable stores all unprocessed events.

Three major classes make up the event queue: the **UI_EVENT** structure, the **UI_QUEUE_ELEMENT** class, and the **UI_QUEUE_BLOCK** class.

The **UI_EVENT** structure contains the event, the type of which depends on the type of class that generated the message. For example, **UID_KEYBOARD** sets the following event information:

- *_event.type* always contains the value *E_KEY.* This lets all receiving objects know that *event.key* contains any related keyboard information.
- *_event.rawCode* contains the keyboard's raw scan code.
- *_event.modifiers* is a flag field indicating the keyboard shift states.
- *_event.key* contains other keyboard information, such as the shift state and the key's value.

The **UI_QUEUE_ELEMENT** and **UI_QUEUE_BLOCK** classes store event information in a list block. The **UI_QUEUE_ELEMENT** class derives from **UI_ELEMENT** and contains the event information.

The **UI_QUEUE_BLOCK** class derives from **UI_LIST_BLOCK** and stores **UI_QUEUE_ELEMENT** objects. Though it's natural for Zinc to use its own **UI_LIST_BLOCK** class to build the **UI_QUEUE_BLOCK** class, Zinc also gains in performance through using these classes, which allow the event queue to buffer event information before the application processes it. By buffering events in a list block, Zinc doesn't allocate and destroy memory every time it receives or dispatches a message, an slow process, thereby increasing performance.

Window Manager

The class **UI_WINDOW_MANAGER** controls the flow of events to all windows and manages the front to back ordering of windows (called the z-order). **UI_WINDOW_MANAGER** derives from **UIW_WINDOW** and uses a virtual **Event( )** member function to process messages it receives from the main event loop.

Window objects

The **UIW_WINDOW** part of the Window Manager contains a list of active windows, and each window contains a list of its window objects. Since window objects derive from **UI_ELEMENT**, they know how to belong to the list that the Window Manager maintains.

Each Zinc window object derives from the **UI_WINDOW_OBJECT** base class, which defines the structure and behavior of window objects. **UI_WINDOW_OBJECT** derives from the **UI_ELEMENT** base class, adding the necessary functionality to display itself and to process events in an **Event( )** virtual member function.

The **Event( )** function processes logical or system events sent to a window object. Here are some sample messages that window objects can interpret:

S_CREATE. Tells the window object to initialize its internal information, such as its size and position within a parent window. The *S_CREATE* message is sent to all of the window objects associated with a window whenever the window is attached to the Window Manager.

S_DISPLAY_ACTIVE. Tells the window object to display itself in its active state. The complementary message is *S_DISPLAY_INACTIVE.*

L_BEGIN_SELECT. Begins the selection process of a window or window object. For example, if the user presses the left mouse button, the selection of an object is initiated. When the mouse button is released, an *L_END_SELECT* is received, and the selection process is completed.

Event member functions

The *UI_WINDOW_MANAGER::***Event( )** member function sends events it receives from the main event loop to windows. For example, if an application contained two overlapping windows, the Window Manager would automatically route normal event information to the top window, but pass a mouse click to the bottom window if the user clicked the mouse on that window.

The Window Manager and window objects understand three types of events:

Logical Events. Logical events are the logical interpretation of a raw event that was generated by an input device. For example, a window would interpret a mouse click as the logical event *L_BEGIN_SELECT*, or "begin selecting something"; but a text field object would interpret the same mouse click as *L_BEGIN_MARK*, or "begin marking text." Logical events have an L_ prefixand generally should not be sent to an object. They are intended to be interpreted.

System Events. The Window Manager, or window objects as the result of a previous event, generate system events. For example, when a window is added to the Window Manager, the Window Manager sends the window an *S_CREATE* event. System events have an S_ prefix and are intended to be generated and sent directly to objects or placed directly on the event queue.

Environment-specific. The operating system or host environment in which the Zinc application is running generates these events. For example, when running under Windows, Zinc objects understand and interpret *WM_* messages such as *WM_PAINT*, or many other Windows messages. The same holds true for Zinc objects running under other operating environments as well.

Help system

The help system, designed to provide help for both general and specific features of an application, contains one important virtual function, **DisplayHelp( )**:

```
class EXPORT UI_HELP_SYSTEM
{
public:
  ...
  virtual void DisplayHelp(UI_WINDOW_MANAGER *windowManager,
    UI_HELP_CONTEXT helpContext = NO_HELP_CONTEXT);
```

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

Error system

The error system brings up a window to display error information whenever an error is detected. The error system inherits one important virtual function, **ReportError( )**.

The **UI_ERROR_SYSTEM** class uses a **UIW_WINDOW** object or an environment specific error handling mechanism to present error information to the screen.

The error system's **ReportError( )** member function is used to display information about the type of error encountered during an application. This function takes **printf( )** style arguments that are used in the text portion of the window.

Screen displays

Display classes provide common display primitive functionality to the Zinc programmer but handle the output using the low-level graphics or text functions. Each display class derives from the **UI_DISPLAY** base class. Zinc defines the following display classes:

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

UI_GRAPHICS_DISPLAY. A DOS graphics display that uses the GFX graphics libraries by C-Source, included with Zinc, to display information to the screen. **UI_GRAPHICS_DISPLAY** supports CGA, EGA, VGA, SVGA, and Hercules monochrome display adapters running in graphics mode.

UI_MACINTOSH_DISPLAY. Uses the Macintosh's QuickDraw routines to display information on screen.

UI_XT_DISPLAY. Uses the X11 drawing primitives to display information using the X window system. Used by the OSF/Motif and X Keys.

UI_MSC_DISPLAY. Uses the Microsoft MSC graphics routines to display information. Supports CGA, EGA, VGA, SVGA, and Hercules monochrome display adapters in graphics mode.

UI_MSWINDOWS_DISPLAY. Uses the Microsoft Windows GDI graphics routines to display information.

UI_NEXTSTEP_DISPLAY. Uses NEXTSTEP's Display PostScript Window Server to display information.

UI_OS2_DISPLAY. Uses OS/2 GPI graphics routines to display information.

UI_TEXT_DISPLAY. A compiler-independent text display used in DOS and Curses. The **UI_TEXT_DISPLAY** class supports MDA, CGA, EGA, and VGA display adapters in the following text modes:

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

This class supports snow checking on CGA monitors and IBM TopView. In turn, TopView supports Microsoft Windows and Quarterdeck DESQview environments.

UI_WCC_DISPLAY. Uses the Watcom graphics routines to display information. Supports CGA, EGA, VGA, SVGA, and Hercules monochrome display adapters in graphics mode.

Other programmer-defined screen display objects. Any custom display object that derives from or conforms to the UI_DISPLAY base class. Zinc posts third-party display classes supporting other DOS graphics libraries on its BBS that Zinc customers are free to download.

Region lists—DOS and Curses

The DOS and Curses display classes derive from **UI_REGION_LIST**, which contains functionality for keeping track of regions on the screen. When a program places an object on the screen under DOS or Curses, the display class reserves a drawing region for the object. As the program places more objects on the screen, the display class splits up the regions to allow more objects to display themselves without disturbing higher level objects, clipping screen regions according to an object's identification.

Region lists have three main components: a **UI_REGION** structure, **UI_REGION_ELEMENT** objects, and a **UI_REGION_LIST** class. The **UI_REGION** structure contains the actual reserved region. The screen coordinates are defined according to the mode of operation, with the top-left corner at {0, 0}. Here are some sample right-bottom coordinates for a screen, based on the type of display mode:

TABLE 10. BGI display values

Display	*Columns*	*Lines*
Text	80	25
	40	25
	80	43
	80	50
CGA	320	200
MCGA	320	200
EGA	350	480
VGA	640	480

The **UI_REGION_ELEMENT** and **UI_REGION_LIST** classes store the region information in elements, organized in a list. The **UI_REGION_ELEMENT** class derives from **UI_ELEMENT** and contains the actual region information as well as a unique identification:

```
class EXPORT UI_REGION_ELEMENT : public UI_ELEMENT
{
public:
  SCREENID screenID;
  UI_REGION region;
```

When a window is attached to the Window Manager, Zinc assigns it a unique value stored in its *screenID* member variable. In addition, the screen is redefined to contain the window's region. This area is represented by a new **UI_REGION_ELEMENT**, where *screenID* is assigned the same value as the window's screen identification, and *region* is assigned the same area occupied by the window. The *region* variable is used later by display functions to clip the boundaries of an object before any screen painting is performed. For example, if two windows were attached to the screen and information were painted to the background window, the background information would be clipped so that the painted regions would not overlap the front window. Since all operating environments other than DOS and Curses handle clipping internally, their display classes do not derive from **UI_REGION_LIST**. In those environments, *screenID* is the handle assigned to the object by the operating system.

Virtual display functions

Virtual display member functions define an abstract method of drawing information to the screen. For example, all display classes have the **Rectangle( )** member function. In text mode, a rectangle is drawn with either a single or a double line. In graphics mode, however, the same routine draws a single or double pixel rectangle. Virtual display member functions allow us to use drawing functions in all of Zinc's display modes by acquiring at run time basic information such as the display's resolution, boundaries, and so forth.

Conclusion

In this chapter, we learned about Zinc's library classes, the basic elements that combine to make up other classes. In the next chapter, we'll learn about how Zinc puts the advanced features of C++ to work across the entire application framework.

Chapter 7

Zinc and C++

In the last chapter, we discussed how Zinc's library classes combine to make up other classes. In this chapter, we'll examine how Zinc uses C++ features to define classes, instantiate and destroy objects, and work with member variables and overloaded functions. We'll also learn how Zinc uses C++'s virtual functions to help objects respond to the right events.

Note that this chapter is *not* a substitute for learning C++, and that Zinc depends heavily on the features of the language for many of its own features. This chapter gives its best results if we are already familiar with C++.

Key Concepts

instantiating and destroying objects

member variables and scope

member functions, overloaded functions and operators

Class definitions

How to design classes

When Zinc's architects wrote the library classes in C++, they followed some explicit rules to make programming in Zinc logical and efficient. Here they are—if we follow them, too, we'll find understanding our code later on will be easier.

1. Precede all C++ class definitions with the reserved word **class**; the environment-specific identifier, **ZIL_EXPORT_CLASS**; and one of the Zinc prefixes **UI_**, **UID_**, **UIW_**, and **ZAF_**.

   The reserved word **class** tells the compiler that the definition not only contains structural information, but member functions, inheritance information, and pointers to member functions as well.

   ZIL_EXPORT_CLASS, not part of the C++ language, is a Zinc type definition to allow us to use one set of source code when writing programs for multiple operating environments, a key benefit of Zinc. In Windows, for example, **ZIL_EXPORT_CLASS** is defined to be **HUGE**, so that Zinc defines **class HUGE UI_ELEMENT**, whereas in DOS, Zinc defines **class UI_ELEMENT**. Without **ZIL_EXPORT_CLASS** , we'd have to maintain one set of source for each environment we wanted to support.

   The prefix **UI_** indicates a "**U**ser **I**nterface" class, **UID_** a "**U**ser **I**nterface **D**evice" class, **UIW_** a "**U**ser **I**nterface **W**indow object" class, and **ZAF_** a "**Z**inc **A**pplication **F**ramework" class. These prefixes allow us to have other C++ classes, such as list and list elements, without worrying that our definition conflicts with Zinc's. Some sample class definitions are given below:

   ```
   class ZIL_EXPORT_CLASS UI_ELEMENT
     ...
   class ZIL_EXPORT_CLASS UI_DEVICE: public UI_ELEMENT
     ...
   class ZIL_EXPORT_CLASS UIW_WINDOW : public UI_WINDOW_OBJECT,
     public UI_LIST
     ...
   class ZIL_EXPORT_CLASS ZAF_MESSAGE_WINDOW : public UIW_WINDOW
     ...
   ```

2. Define public members first, then protected members, and private members last. This way, we can find the member information we need without wading through the wrong variables and functions.

Any function can access public members, which are documented in the *Programmer's Reference*. Only instances of the class itself, objects derived from those classes, and objects that are friends of that class can access protected members, also documented in the *Programmer's Reference*. Last, only instances of the class itself or friend classes can access a private member variable; derived classes that are not friend classes may not access the private members of another class. *Private members are not documented in any Zinc manual.*

Below, the **UID_KEYBOARD** class, which derives from the **UI_DEVICE** class, shows how this member access order is followed. Note that the **UID_KEYBOARD** class, since it derives from **UI_DEVICE**, could access **UI_DEVICE**'s public and protected members; but since it's not a friend class of **UI_DEVICE,** it may not access any private members.

```
class ZIL_EXPORT_CLASS UID_KEYBOARD : public UI_DEVICE
{
public:
  static EVENT_TYPE breakHandlerSet;
  UID_KEYBOARD(DEVICE_STATE state = D_ON);
  virtual ~UID_KEYBOARD(void);
  virtual EVENT_TYPE Event(const UI_EVENT &event);
protected:
  virtual void Poll(void);
};
```

3. Finally, place member variables and functions in separate logical groups. Zinc groups member variables according to a logical order such as byte boundary alignment, first use, most common usage, or a number of other factors—we may pick the order we like best, but we should stick with it. In contrast, however, we organize member functions in alphabetical order with the constructor and destructor first. The **UIW_BUTTON** class shows how.

```
class ZIL_EXPORT_CLASS UIW_BUTTON : public UI_WINDOW_OBJECT
{
public:
  BTF_FLAGS btFlags;
  EVENT_TYPE value;
  UIW_BUTTON(int left, int top, int width, ZIL_ICHAR *text,
    BTF_FLAGS btFlags = BTF_NO_TOGGLE | BTF_AUTO_SIZE,
    WOF_FLAGS woFlags = WOF_JUSTIFY_CENTER,
    USER_FUNCTION userFunction = NULL, EVENT_TYPE value = 0,
    ZIL_ICHAR *bitmapName = NULL);
  virtual ~UIW_BUTTON(void);
  virtual EVENT_TYPE Event(const UI_EVENT &event);
```

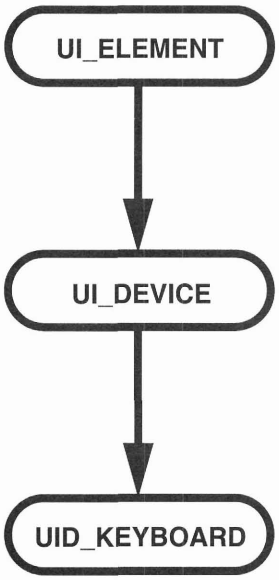
```
ZIL_ICHAR *DataGet(int stripText = FALSE);
void DataSet(ZIL_ICHAR *text);
virtual void *Information(INFO_REQUEST request, void *data,
  OBJECTID objectID = 0);
static EVENT_TYPE Message(UI_WINDOW_OBJECT *object, UI_EVENT &event,
  EVENT_TYPE ccode);
...
```

In addition to the class definition rules described above, Zinc Software employees adhere to a full set of internal coding standards, designed to improve the readability and maintenance of code. For a full explanation of these rules see "Appendix C—Zinc Coding Standards."

Derived classes

Deriving classes, otherwise known as inheritance, is a benefit of C++ that allows us to build applications with more functionality, less code, and fewer bugs. By deriving a class and then adding or changing the behavior we want, we leave other code untouched. If we wanted to do something conceptually similar in C, we would have to copy all the code in a procedure that we would otherwise subclass in C++, and modify much or all of it in order to add or change the behavior we want. Copying and modifying, in contrast to deriving, introduces bugs not present before, increases complexity, and results in larger code and executable size.

One example of inheritance in Zinc is the **UID_KEYBOARD** class, whose hierarchy is shown below:



Deriving **UID_KEYBOARD** from **UI_DEVICE** and **UI_ELEMENT** base classes has two benefits. First, because **UID_KEYBOARD** derives from **UI_ELEMENT**, classes that derive from **UI_LIST** can group and manipulate it; this means the Event Manager can manage a **UID_KEYBOARD** object. Second, because **UID_KEYBOARD** also derives from **UI_DEVICE**, the Event Manager can call **UID_KEYBOARD**'s virtual **Poll( )** function, thereby allowing the keyboard device to place events into the event queue.

Another example of class inheritance are the **UIW_MINIMIZE_BUTTON** and **UIW_MAXIMIZE_BUTTON** classes, both three-dimensional buttons which function when the user clicks on them with the mouse. Fundamentally, they're the same, but we change them by giving them different appearances and by making them do different things.

Multiple inheritance

Multiple inheritance allows classes to inherit behavior from classes with different member functions and variables. This helps us avoid duplicating work when our own classes must inherit behavior common to more than one class. However, multiple inheritance has its critics.

Some programmers using object-oriented languages such as Objective-C and Smalltalk-80 believe that multiple inheritance leads to more complicated classes. Indeed, classes with multiple parents have code that's harder to read. However, Zinc could not have implemented some features as elegantly and in such a small amount of code without multiple inheritance. Despite adding more complexity to a class, multiple inheritance allows us to extend the features of objects with less work, minimal code duplication, and more intuitively than if C++ did not use multiple inheritance.

UIW_WINDOW is an example of the benefits of multiple inheritance, because it derives from both **UI_WINDOW_OBJECT** and **UI_LIST**, using behaviors common to both. Because **UIW_WINDOW** derives from **UI_WINDOW_OBJECT**, which in turn derives from **UI_ELEMENT**, it can act as an element of a list. Also, because **UIW_WINDOW** derives from the **UI_LIST** base class, **UIW_WINDOW** can also behave as a list that manages elements such as buttons, strings, and tool bars. Because of multiple inheritance, **UIW_WINDOW** and other classes can inherit behavior from disparate classes—without it, we would find implementing **UIW-_WINDOW** much more difficult.

Abstract classes

Abstract classes define a function but don't implement it—they leave the implementation to another class, allowing functionality to be decided at run time. For example, Zinc's display function defines a display, but leaves how that display function will work to a derived class that detects what display the computer is using, and configures itself appropriately.

Zinc uses abstract classes in its methods of abstracting devices and displays of native operating environments. For example, Zinc's **UI_DISPLAY** class defines some basic behaviors, such as drawing lines and polygons—but it leaves the implementation of these behaviors to classes that derive from **UI_DISPLAY**. This way, a derived display class can inherit basic behaviors from **UI_DISPLAY**, and implement them for a specific operating environment's display. This is what Zinc calls a "less-thin" layer of abstraction over the native operating environment's API, in contrast to a thin or thick layer.

Because a thin layer is tightly bound to an operating environment, it provides higher performance, but at the cost of less programming flexibility and portability. In contrast, a thick layer of abstraction provides greater programming flexibility and portability, but at the cost of lower performance. Zinc treads a middle ground between thin and thick layers that benefits us two ways.

The first benefit of Zinc is that our Zinc programs run nearly as fast as programs that wrap a thin layer over the operating environment. Second, we will find that writing the program will be nearly as flexible and portable as writing a program using a thick layer of abstraction of the operating environment.

For a class to be considered abstract, it must have one or more pure virtual functions. For example, **UI_DEVICE** has two pure functions, **Event( )** and **Poll( )**. Neither actually do anything in **UI_DEVICE**; rather, their functionality is implemented by the devices that inherit from **UI_DEVICE**. Here's an example of **UI_DEVICE**'s virtual functions:

```
class ZIL_EXPORT_CLASS UI_DEVICE : public UI_ELEMENT
{
  friend class ZIL_EXPORT_CLASS UI_EVENT_MANAGER;
public:
  ...
  virtual EVENT_TYPE Event(const UI_EVENT &event) = 0;
protected:
  ...
  virtual void Poll(void) = 0;
};
```

Abstract classes help us because we can define how a class behaves without associating any specific code with the class. However, some classes appear abstract, even though they are not; for example, the **UI_WINDOW_OBJECT** appears like an abstract class, but it is not an abstract class because it has no pure virtual functions. We'll discuss virtual functions in more detail in this chapter, including how virtual functions free us from tying events to windows and window objects.

Friend classes

Friend classes allow a specified class to gain access to the protected and private members of another class; we can hide the implementation of one class but let a similar or corresponding class have special access rights. Often, a Zinc class grants friend rights to other classes, most often, in Zinc Designer. Other times, a class derived from the **UI_ELEMENT** base class grants friend access to its parent list, allowing it to optimize access to its list elements.

Object creation

Explicit instantiation

Once we've defined a class, the next logical step is to put it to work by *instantiating* it, which means creating an object from the definition of a class by allocating memory for it. When we instantiate objects, we either use the **new** operator, or we create a static instance that is deleted automatically when the program moves out of scope. Using the **new** operator is called *explicit instantiation*, because by doing so, we state explicitly that we want to instantiate a new object. Explicit instantiation is dynamic; the memory for the new object is allocated from the freestore of available memory. The **new** operator initializes a class and maintains its information until it sees a **delete** operator, which frees the memory; if we didn't use the **new** operator, the object would be destroyed when the scope of the function ended.

Here is some sample code that initializes the display, the Event Manager, and the Window Manager using the **new** operator:

```
#include <ui_win.hpp>
main()
{
  // Initialize the screen.
  UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    ...
  // Initialize the event manager.
  UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    ...

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

Implicit instantiation and scope

In contrast to using **new** to explicitly instantiate an object, we can write a function to *implicitly* instantiate an object, which means the program instantiates the object when it reaches the scope of its class. The biggest difference between implicit and explicit instantiation is that implicit instantiation is static; the compiler is responsible for allocating memory for the object.

In this example, the window created will be automatically deleted when the scope of the function ends.

```
#include <ui_win.hpp>
ExampleFunction()
{
    // Create a window.
    UIW_WINDOW window(0, 0, 25, 5);
    ...
}
```

Base class construction

A constructor initializes a new instance of an object, assigning to the object the appropriate startup information. But C++ classes also call the constructors of their base classes to assign startup information to them as well. For example, the **UI_TEXT_DISPLAY**, which inherits from **UI_DISPLAY** and **UI_REGION_LIST**, calls the **UI_DISPLAY** constructor and the **UI_-REGION_LIST** constructor before it initializes any information:

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

C++ initializes a base class with no arguments automatically, whether or not the derived constructor calls the base class. But Zinc calls base classes explicitly in order to make code more readable. The **UI_REGION_LIST** code above is one example of this—notice that we called **UI_REGION_LIST** from the constructor of **UI_TEXT_DISPLAY**. In another example, here the **UID_KEYBOARD** constructor calls **UI-_DEVICE** to initialize its base class information:

```
UID_KEYBOARD::UID_KEYBOARD(DS_STATE initialState) :
  UI_DEVICE(E_KEY, initialState)
{
  ...
}
```

Sometimes, this base class initialization goes several levels up the inheritance hierarchy. In the following example, **UIW_POP_UP_ITEM** class calls the **UIW_BUTTON** class for initialization, which in turn calls **UI_WINDOW_OBJECT** for base class initialization. This saves a lot of code we'd otherwise need to write to initialize each object separately:

```
UIW_BUTTON::UIW_BUTTON(int left, int top, int width,
  ZIL_ICHAR *_text, BTF_FLAGS _btFlags, WOF_FLAGS _woFlags,
  USER_FUNCTION _userFunction, EVENT_TYPE _value,
  ZIL_ICHAR *_bitmapName) :
  UI_WINDOW_OBJECT(left, top, width, 1, _woFlags,
    WOAF_NO_FLAGS),
  text(ZIL_NULLP(ZIL_ICHAR)), btFlags(_btFlags),
  value(_value), depth(2),
  btStatus(BTS_NO_STATUS), bitmapWidth(0), bitmapHeight(0),
  bitmapArray(ZIL_NULLP(UINT8))
{
  ...
}

UIW_POP_UP_ITEM::UIW_POP_UP_ITEM(void) :
  UIW_BUTTON(0, 0, 1, ZIL_NULLP(ZIL_ICHAR), BTF_NO_3D,
  WOF_NO_FLAGS),
  menu(0, 0, WNF_NO_FLAGS, WOF_BORDER,
    WOAF_TEMPORARY | WOAF_NO_DESTROY),
  mniFlags(MNIF_SEPARATOR)
{
  ...
}
```

Array constructors

An *array constructor* initializes an array, and an example of a class that uses an array constructor is **UI_QUEUE_BLOCK**. Array constructors help the Event Manager run more efficiently by allowing it to allocate memory for the queue all at once, rather than allocating it as events come into the queue, and then deallocating the blocks after it has been used. The code below shows how the queue block initializes event information:

```
UI_QUEUE_BLOCK::UI_QUEUE_BLOCK(int _noOfElements) :
  UI_LIST_BLOCK(_noOfElements)
{
  // Initialize the queue block.
  UI_QUEUE_ELEMENT *queueBlock = new
    UI_QUEUE_ELEMENT[_noOfElements];
  elementArray = queueBlock;
  for (int i = 0; i < _noOfElements; i++)
    freeList.Add(NULL, &queueBlock[i]);
}
```

Overloaded constructors

Overloaded constructors are constructors that let us specify different parameters, depending on how we would like to initialize the information in a new instance of an object. For example, the **ZIL_DATE** class overloads its constructor in the following manner:

```
class ZIL_EXPORT_CLASS ZIL_DATE
{
  ZIL_DATE(void);
  ZIL_DATE(const ZIL_DATE &date);
  ZIL_DATE(int year, int month, int day);
  ZIL_DATE(const ZIL_ICHAR *string,
    DTF_FLAGS dtFlags = DTF_NO_FLAGS);
```

Overloaded date constructors in the **ZIL_DATE** class allow us to create a date object according to:

- the computer's system date, which requires no arguments;
- a previously created date object;
- three integer values, the year, month, and day; and
- a country-independent, alphanumeric date.

Most classes derived from **UI_WINDOW_OBJECT** have at least two overloaded constructors: one, or more, for basic run-time setup, and another for persistent object access. For example, the **UIW_POP_UP_ITEM** class has the following definitions:

```
class ZIL_EXPORT_CLASS UIW_POP_UP_ITEM : public UIW_BUTTON
{
  UIW_POP_UP_ITEM(void);
  UIW_POP_UP_ITEM(ZIL_ICHAR *text,
    MNIF_FLAGS mniFlags = MNIF_NO_FLAGS,
    BTF_FLAGS btFlags = BTF_NO_3D, WOF_FLAGS woFlags = WOF_NO_FLAGS,
    ZIL_USER_FUNCTION userFunction =
    ZIL_NULLF(ZIL_USER_FUNCTION), unsigned value = 0);

  // Persistent object constructor.
  UIW_POP_UP_ITEM(const ZIL_ICHAR *name,
    ZIL_STORAGE_READ_ONLY *file,
    ZIL_STORAGE_OBJECT_READ_ONLY *object);
```

The first constructor provides menu item separators, the second creates the pop up item according to the information in the constructor, and the last construct the pop-up item from disk information.

Copy constructors

A copy constructor lets us pass a previously created class into the constructor of another object. We use copy constructors when we want to instantiate a new object with the data contained in another object. Several library classes use copy constructors: **ZIL_BIGNUM**, **ZIL_DATE, ZIL_TIME**, and **ZIL_UTIME**. An example of the date constructor is shown below:

```
class ZIL_EXPORT_CLASS ZIL_DATE
{
  ZIL_DATE(void) { DataSet(); }
  ZIL_DATE(const ZIL_DATE &date);
  ZIL_DATE(int year, int month, int day);
  ZIL_DATE(const ZIL_ICHAR *string,
    DTF_FLAGS dtFlags = DTF_NO_FLAGS);
```

Default arguments

Often, constructors give us the choice whether or not to use a default argument, which sets up some default behavior for an object when we instantiate it. When we call a constructor, we can leave out any arguments and use the constructor's default, which Zinc specifies. The text display class uses a default argument, *TDM_AUTO*, which sets the display to the highest possible text resolution.

```
class ZIL_EXPORT_CLASS UI_TEXT_DISPLAY : public UI_DISPLAY,
  public UI_REGION_LIST
{
public:
  UI_TEXT_DISPLAY(TDM_MODE mode = TDM_AUTO);
```

If we want to use the text display's default, we can call the constructor with no arguments:

```
UI_DISPLAY *display = new UI_TEXT_DISPLAY;
```

Otherwise, we can override the default by providing an argument. In this case, our argument tells the constructor to create an 80 x 43 text display.

```
// Force 43 line mode.
UI_DISPLAY *display = new UI_TEXT_DISPLAY(TDM_43x80);
```

Many other member functions contain default information. The *Programmer's Reference* contains information about the types of default arguments, their use, and overriding their definition.

Object deletion

Explicit deletion

Once we're done with an object, the next logical step is to delete it. When we delete an object, we either use the **delete** operator, or allow the system to delete the object when the scope of the function that instantiated the object ends. The order of class creation and destruction is important. Generally, the objects we create first we destroy last.

If we created an object using the reserved word **new**, we must delete it. For example, when we create a display, Event Manager, and Window Manager with **new**, we must use **delete** to free them.

```
#include <ui_win.hpp>
  main()
{
  // Initialize Zinc using the new operator.
    UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(
      display);
    UI_WINDOW_MANAGER *windowManager =
      new UI_WINDOW_MANGER(display, eventManager);
    ...
    // Restore the system.
    delete windowManager;
    delete eventManager;
    delete display;
  }
```

Implicit deletion and scope

The example above showed how we could use the reserved word **delete** to delete new objects. However, when a function creates a static instance of an object, when the function's scope ends, the object will be deleted automatically. In the example below, the class destructor is called automatically when the scope of **ExampleFunction(**) ends.

```
ExampleFunction()
{
  UIW_WINDOW window(0, 0, 25, 5);
  ...
  // The window is automatically destroyed when the scope of
  // ExampleFunction ends.
}
```

The order of class creation and destruction is important. In general, those objects that you create first, must be destroyed last.

Virtual destructors

Virtual destructors allow Zinc to call the destructor of the base class, rather than the destructor of the derived object. This saves us from writing functions that delete instances of classes that derive from our base class. For example, the keyboard, cursor, and mouse derive from **UI_DEVICE**, which is derived from **UI_ELEMENT**. If we delete the Event Manager, when its list is destroyed, all objects attached to the list will be destroyed, even though the list cannot possibly know what types of objects it is deleting.

```
class ZIL_EXPORT_CLASS UI_LIST
{
public:
  virtual ~UI_LIST(void) { Destroy(); }
  ...
}
void UI_LIST::Destroy(void)
{
  UI_ELEMENT *tElement;
  // Delete all the elements in the list.
  for (UI_ELEMENT *element = first; element; )
  {
    tElement = element;
    element = element->next;
    delete tElement;
  }
  ...
}
```

Base class destruction

When we call the destructor of a derived class, C++ calls the destructor of the base class. This saves us from calling the destructor by hand, saving us code. The **UIW_BUTTON** class's destructor is a good example of how a derived class constructor calls its base class's destructor.

```
UIW_BUTTON::~UIW_BUTTON(void)
{
  if (string)
    delete string;
}
```

After the button class destructor is executed, C++ automatically calls the destructor of **UI_WINDOW_OBJECT**, then the destructor for **UI_ELEMENT**. Thus, destruction of class objects works in an order opposite of class construction. This way, member variables in base classes that a derived class may rely on will still exist until after the derived object has been completely destroyed.

Array destruction

UI_QUEUE_BLOCK uses an array destructor to delete its queue elements. Array destructors should be used only in conjunction with array constructors. Further, some compilers require that we specify the number of elements in the array when deleting it, whereas others do not. The code for array destruction is shown below.

```
UI_QUEUE_BLOCK::~UI_QUEUE_BLOCK(void)
{
  // Free the queue block.
  UI_QUEUE_ELEMENT *queueBlock = (UI_QUEUE_ELEMENT *)elementArray;
  delete queueBlock;
}
```

Member variables

Variable definitions

As we discussed earlier in the chapter, Zinc member variables begin with a lowercase character and are organized according to a logical order, such as byte boundary alignment, first use, most common usage, or whatever makes sense. An example of how Zinc defines member variables is the **UI_LIST** class, with several of its member variables shown below:

```
class ZIL_EXPORT_CLASS UI_LIST
{
protected:
  UI_ELEMENT *first, *last, *current;
  ZIL_COMPARE_FUNCTION compareFunction;
```

Zinc objects define and use member variables as bitwise flags. **UI_WINDOW_OBJECT::***woFlags* is a good example of this:

```
// --- woFlags ---
typedef unsigned WOF_FLAGS;
const WOF_FLAGS WOF_NO_FLAGS= 0x0000;
const WOF_FLAGS WOF_JUSTIFY_CENTER= 0x0001;
const WOF_FLAGS WOF_JUSTIFY_RIGHT= 0x0002;
const WOF_FLAGS WOF_BORDER= 0x0004;
const WOF_FLAGS WOF_VIEW_ONLY= 0x0010;
const WOF_FLAGS WOF_UNANSWERED= 0x0080;
const WOF_FLAGS WOF_INVALID= 0x0100;
const WOF_FLAGS WOF_NON_FIELD_REGION= 0x0200;
const WOF_FLAGS WOF_NON_SELECTABLE= 0x0400;
const WOF_FLAGS WOF_AUTO_CLEAR= 0x0800;
class ZIL_EXPORT_CLASS UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
  WOF_FLAGS woFlags;
```

The base class **UI_WINDOW_OBJECT** logically ORs together the bits of *woFlags* to form composite values to determine its mode of operation. See the *Programmer's Reference* for what each flag sets.

Static member variables

Occasionally, classes define static member variables, which provide the same information to any instance of the class or of a derived class. For example, the **UI_WINDOW_OBJECT** class has a static member variable called *windowManager*, which is a pointer to the Window Manager. All objects that derive from **UI_WINDOW_OBJECT** will therefore point to the same Window Manager without any added work on our part. Other pointers in **UI_WINDOW_OBJECT**, such as *eventManager* and *display*, allow all window objects to use the same error and help systems.

```
static UI_DISPLAY *display;
static UI_EVENT_MANAGER *eventManager;
static UI_WINDOW_MANAGER *windowManager;
```

In addition to providing the same information to all objects of a class or that derive from a class, static variables store internal information. For example, in top-down operating systems such as DOS, Macintosh, and Curses, and under certain conditions in bottom-up operating systems, the **UI_WINDOW_OBJECT** class uses a static variable called *repeatRate* to store the rate at which an object will repeat a character when the user holds down a key, as well as another called *doubleClickRate*, which determines how fast a window will respond to the double-click of a mouse.

```
static int repeatRate;
static int doubleClickRate;
```

Remember that when we use static pointers as part of a class, C++ requires that we declare space for them outside of the class definition.

Member functions

Function definitions

Zinc functions begin with an uppercase letter and usually form complete words that describe the function. For example, the **UI_ELEMENT** class has the member functions **ListIndex( )**, **Next( )** and **Previous( )**:

```
class ZIL_EXPORT_CLASS UI_ELEMENT
{
public:
  int ListIndex(void);
  UI_ELEMENT *Next(void);
  UI_ELEMENT *Previous(void);
```

Default arguments

Earlier we learned that constructors often give us the choice whether or not to use a default argument, which sets up some default behavior for an object when we instantiate it. Just as constructors can use default arguments, so can member functions, which use default arguments to behave consistently. For example, **UI_DISPLAY** uses many default arguments for filling zones and XORing the screen output. Notice the default arguments in **UI_DISPLAY**'s **Bitmap**( ), **Ellipse**( ),and **MapColor**( ) functions.

```
class ZIL_EXPORT_CLASS UI_DISPLAY : public ZIL_INTERNATIONAL
{
public:
...
virtual ~UI_DISPLAY(void);
virtual void Ellipse(ZIL_SCREENID screenID, int column, int line,
    int startAngle, int endAngle, int xRadius, int yRadius,
    const UI_PALETTE *palette, int fill = FALSE, int _xor = FALSE,
    ...

virtual void Line(ZIL_SCREENID screenID, int column1,
    int line1, int column2, int line2,
    const UI_PALETTE *palette, int width = 1, int _xor = FALSE,
    const UI_REGION *clipRegion = ZIL_NULLP(UI_REGION));
    ...

virtual void Polygon(ZIL_SCREENID screenID, int numPoints,
    const int *polygonPoints, const UI_PALETTE *palette,
    int fill = FALSE, int _xor = FALSE;
};
```

Virtual member functions

In C++, virtual member functions ensure that when we call an object's member function, we don't call the member function of the base class with the same name. Zinc takes advantage of virtual functions by defining them in a base class, and overloading them in derived classes to give those classes basic behavior. In one example, the **UI_DEVICE** class defines virtual **Event**( ) and **Poll**( ) routines.

```
class ZIL_EXPORT_CLASS UI_DEVICE : public UI_ELEMENT
{
public:
  virtual EVENT_TYPE Event(const UI_EVENT &event) = 0;
protected:
  virtual void Poll(void) = 0;
```

When the Event Manager calls its devices' **Poll**( ) functions, instead of calling these functions, the Event Manager calls the virtual **Poll**( ) functions of the keyboard, mouse, and cursor.

Virtual functions and message handling

Earlier, we discussed how each window and window object interprets events according to how the object operates, eliminating the need for use to write code to tie events to window objects. Virtual functions help this happen. If we wrote a program with a Window Manager and an attached window, when the user clicked the mouse button, the Window Manager would send a message to the window, where it calls the **UIW_WINDOW::Event( )** function to override the actions that the **UI_WINDOW_OBJECT** base class would normally perform. But if the window doesn't know how to handle the message, the window would pas the event up its inheritance hierarchy by calling the **UI_WINDOW_OBJECT::Event( )** base class function, which may know how to handle the message. One benefit to how Zinc uses virtual functions is that we can send messages to an object without having to know how the object works; we let Zinc handle those details for us. A second benefit, the one we've already discussed, is that we don't need to tie events to window objects; we can tell a window object which events to watch for, and let the window object work naturally.

Overloaded member functions

Overloaded member functions allow us to specify different parameters and values that a function accepts by default. For example, the **ZIL_DATE** class overloads two member functions, **Export( )** and **Import( )**:

```
class ZIL_EXPORT_CLASS ZIL_DATE : public ZIL_UTIME
{
public:
  void Export(int *year, int *month, int *day,
    int *dayOfWeek=ZIL_NULLP(int));
  void Export(ZIL_ICHAR *string, DTF_FLAGS dtFlags);
  void Export(int *packedDate);
  DTI_RESULT Import(void);
  DTI_RESULT Import(const ZIL_DATE &date);
  DTI_RESULT Import(int year, int month, int day);
```

The overloaded **Export( )** functions allow us to get

- a date based on three integers, year, month, and day
- a date based on an alphanumeric value
- a date in a packed integer format

The overloaded **Import( )** functions allow us to set

- a system date, which requires no arguments;
- a date based on a date class object previously constructed;
- a date based on the year, month, and day; and

Overloaded operators

Used properly, operator overloading is a major benefit of C++ for writing more elegant and readable code.

Zinc uses operator overloading two different ways. The most common way is to add an element to an existing list, as do the base classes **UI_LIST**, **UI_EVENT_MANAGER**, **UI_WINDOW_MANAGER**, **UIW_WINDOW**, and all objects that derive from the **UIW_WINDOW** class. The **+** operator allows us to add a border, a maximize button, a minimize button, a system button, and a title to a parent control class, such as a window. For example, we could use the following code to create a window and then attach to it sublevel window objects:

```
// Create a simple window and attach sublevel window objects.
UIW_WINDOW *window = new UIW_WINDOW(5, 5, 40, 6);
*window
  + new UIW_BORDER
  + new UIW_MAXIMIZE_BUTTON
  + new UIW_MINIMIZE_BUTTON
  + new UIW_SYSTEM_BUTTON(SYF_GENERIC)
  + new UIW_TITLE("Simple Window");
```

The second way Zinc uses overloaded operators is with the **ZIL_DATE** and **ZIL_TIME** classes, which define operations for =, +, -, >, >=, <, <=, ++, --, +=, -=, == and !=. In **ZIL_DATE** and **ZIL_TIME**, these operators increment the values of date or time objects or compare the chronological value of two date or time objects. Below is an example of how **ZIL_DATE** does this.

```
// ---- ZIL_DATE ------------------------------------------
class ZIL_EXPORT_CLASS ZIL_DATE : public ZIL_UTIME
{
public:
  long operator=(long days) { jday = days; return (jday); }
  long operator=(const ZIL_DATE &date)
    { jday = date.jday; usec = date.usec; return (jday); }
  long operator+(long days) { return (jday + days); }
  long operator+(const ZIL_DATE &date)
    { return (jday + date.jday); }
  long operator-(long days) { return (jday - days); }
  long operator-(const ZIL_DATE &date)
    { return (jday - date.jday); }
  long operator++(void) { jday++; return (jday); }
  long operator--(void) { jday--; return (jday); }
  void operator+=(long days) { jday += days; }
  void operator-=(long days) { jday -= days; }
  int operator==(const ZIL_DATE& date)
    { return (ZIL_UTIME::operator==(date)); }
```

```
        int operator!=(const ZIL_DATE& date)
          { return (ZIL_UTIME::operator!=(date)); }
        int operator>(const ZIL_DATE &date)
          { return (ZIL_UTIME::operator>(date)); }
        int operator>=(const ZIL_DATE &date)
          { return (ZIL_UTIME::operator>=(date)); }
        int operator<(const ZIL_DATE &date)
          { return (ZIL_UTIME::operator<(date)); }
        int operator<=(const ZIL_DATE &date)
          { return (ZIL_UTIME::operator<=(date)); }
        void SetBasis(int _basisYear) { basisYear = _basisYear; }
        int GetBasis() { return basisYear; }
      };
```

The example below shows how we can use the overloaded date operators to compare a date to special times throughout the year.

```
ZIL_DATE currentDate;// Initialize the system date.
ZIL_DATE newYears1990("Jan. 1, 1990");
ZIL_DATE twentyFirstCentury("Jan. 1, 2001");
// Check the dates
if (currentDate == newYears1990)
  printf("Happy new year!\n");
else if (currentDate < twentyFirstCentury)
  printf("It's not the twenty-first century.\n");
else
  printf("It's the twenty-first century.\n");
```

Static member functions

Analogous to a static member variable, a static member function provides to all instances of a class or of a derived class a common function. Here's why Zinc uses static member functions.

Static member functions allow us to check programmatically class information before calling the class's constructor. A good example of this is the **ZIL_STORAGE_READ_ONLY** class, where we can check the validity of a file or directory path without first creating a storage unit. We can do this by calling the **ZIL_STORAGE_READ_ONLY::ValidName( )** member function.

```
class ZIL_EXPORT_CLASS ZIL_STORAGE_READ_ONLY :
  public UI_LIST
{
public:
  static int ValidName(const ZIL_ICHAR *name,
    int createStorage = FALSE);
```

Static member functions perform generic operations. Two static members fit into this category: **UIW_WINDOW::Generic( )** and **UIW_-SYSTEM_BUTTON::Generic( )**. We can use these member functions, not only to construct the object, but also to place generic subobjects in their lists. For example, the definition for **UIW_WINDOW::Generic( )** lets us make one call that initializes a window and adds the border, maximize button, minimize button, system button, and title:

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

Static member functions send system messages to the Event Manager. For example, when the end user presses <Enter> or clicks the mouse button on a **UIW_BUTTON** object whose *BTF_SEND_MESSAGE* flag is set, the button sends a message, whose type is *UIW_BUTTON::value*, to the Event Manager. It does this by calling a static member function called **Message( )**, which simply places the event on the queue.

All window objects use static member functions when our programs call the persistent object constructor. Each window object loaded from a Zinc Designer file has a static member function called **New( )**, which links all code related to the class into the executable when the program calls an object's constructor. Below is an example of an object's **New( )**:

```
static UI_WINDOW_OBJECT *New(const ZIL_ICHAR *name,
  ZIL_STORAGE_READ_ONLY *file,
  ZIL_STORAGE_OBJECT_READ_ONLY *object)
  { return (new UIW_BUTTON(name, file, object)); }
```

Often, when using static member functions, we'll find certain reasons for using pointers to those functions. One important use for pointers to static member functions is the addition of user functions to objects. For example, when we create a button, we may want that button to call some function that we wrote instead of one that Zinc wrote. If the user function is a member function, it must be declared static because otherwise C++ doesn't allow its address to be passed.

Conclusion

In this chapter, we've discussed how Zinc uses C++ features in defining classes, instantiating and destroying objects, and working with member variables and overloaded functions, in addition to how scope affects writing programs in Zinc.

In the next chapter, we'll discuss the concepts of globalizing an application.

Chapter 8

Globalization

In this chapter we'll discuss the concepts of globalizing an application. The basic Zinc package is already fully globalized. You can build globalized applications that use either 8-bit character strings or 16-bit Unicode character strings.

Globalizing an application takes two steps: enabling and localizing. Enabling a program means to create the program in such a way that it can be easily ported to any locale. Typically, an application is not enabled, unless the program can be localized without recompiling the source code. Therefore an enabled program must detect its locale and resolve any hardware dependencies at run time. One example of how difficult this can be is writing a program enabled for the Japanese marketplace. Since most Japanese PCs are non-ISA compliant, a program enabled for the Japanese marketplace must

Key Concepts

enabling a Zinc program

how to use ISO 8859-1 and Unicode characters

shipping a globalized application

use different low-level functions. Therefore our application must know how to detect that it's running on Japanese PC hardware and configure itself accordingly.

Localizing an application means to adapt the application to run properly for a particular locale. This means that the program displays and formats date, time, currency, and number fields consistently with how someone native to that locale would expect to see them. Additionally, the program should translate any of its text appropriately.

Zinc is already enabled and has been localized for many different languages and locales. For an up-to-date list of the supported languages and locales, see the **READ.ME** file.

As mentioned above, globalizing an application is done in two steps: first enabling the application and then localizing it. Enabling the application is the foundation upon which globalization is built, and so we must enable our programs by design, not after-the-fact. Here are some issues to consider when designing your applications..

Enabling Zinc objects

Zinc's architects have enabled all objects in the Zinc library specifically to ease globalizing our programs. We need not do anything to Zinc objects to use them in our globalized applications.

Enabling objects

Any object we use that presents information will likely need to be localized later. This means we must provide a mechanism to allow the program to set its data dynamically. We can follow three approaches.

1. Hardcode the data and change it for each locale. This is not a recommended approach, since we may miss translating something, and since we would have to provide a separate executable for each locale or language.
2. Place the data in a separate module, in a table perhaps, that we can compile and link into our application. This is not a good approach, since the executable can only support a single language or locale;

3. Place the data in a data file that can be accessed at run time. Zinc uses a combination of methods two and three. Later in this chapter, we will discuss how Zinc uses these methods.

Character types

Zinc uses one of two character sets: ISO 8859-1 or Unicode. The ISO 8859-1 characters are eight bits wide, while the Unicode characters are 16 bits. We choose our program's characters by examining its requirements. By default, Zinc programs use ISO 8859-1 characters, but if we want to build a Unicode application we define **ZIL_UNICODE** in **UI_ENV.HPP,** and our application will use Unicode characters. Also, whenever Zinc needs to output text to the screen or get text from the user, it maps the characters to or from the hardware character set. Zinc provides mappings for many common hardware character sets.

Because Zinc can support either 8- or 16-bit characters, our programs must be written for either type. We do this by using **ZIL_ICHAR** types, a specific Zinc data type, instead of the **char** type wherever we use characters. A **ZIL_ICHAR** variable resolves to an 8-bit char when the program doesn't use Unicode, or to a 16-bit unsigned short if it does. The **ZIL_UNICODE** macro defines its type definition. If **UI_ENV.HPP** doesn't define **ZIL_UNICODE** when compiling the library, **ZIL_ICHAR** will be 8 bits wide, whereas if it does, **ZIL_ICHAR** will be 16 bits wide.

Remember that the **ZIL_UNICODE** definition must be consistent between the library and our source code.

Some compilers provide a **wchar_t** type, which should resolve to a 16-bit or wider character type. Unfortunately, not all compilers support *wchar_t* or define it to be a 16-bit or wider value. So even if our compiler supports **wchar_t**, Zinc recommends using **ZIL_ICHAR** in case we must port our application to another environment without a compiler that correctly supports the **wchar_t** type. In either case, **ZIL_ICHAR** provides more flexible portability, as it resolves to either an 8-bit character or a 16-bit character as appropriate for the application, whereas **wchar_t** stays the same.

Using wide character strings

This section applies to Unicode programs only. If we don't currently support Unicode, Zinc recommends nonetheless the use of the techniques presented here, or at least encourages familiarity with them, in case our application utilizes the Unicode character set in the future.

If our compiler supports wide character strings correctly, meaning **wchar_t** is a 16-bit value or wider as discussed above, we can define literal strings with the 'L' type specifier, such as **wchar_t** **wideString* = L"wide string."

But if our compiler does not support wide character strings—and many compilers don't—we must use an alternate method of creating string literals, because a string in double-quotes resolves to 8-bit characters. Though the practice looks unconventional, we must create strings as arrays of **ZIL_ICHAR** characters, initialized by specifying each character individually. For instance, the text "wide string" would look like this:

```
ZIL_ICHAR wideString[] = { 'w', 'i', 'd', 'e',
  ' ', 's', 't', 'r', 'i', 'n', 'g', 0};
```

Note the terminating 0 at the end of the string. The variable *wideString* from this example is a 16-bit string we can use like a normal 8-bit string, except that we must use the **ZIL_INTERNATIONAL** string functions for string manipulation.

Localizing our application

Once we've enabled our application, localizing it is a matter of mere language translation.

Localizing Zinc objects

Many Zinc objects must be localized, which includes setting the correct date, time, or number formatting; using translated text for things like system menu options; providing default strings for things like error messages, and possibly even changing bitmaps for icons and buttons. Zinc automatically localizes objects based on the system's language and locale. If the program cannot support the system's language or locale for some reason, the program will use its fallback data, originally linked at compile time.

The fallback data is the language, locale, and image information that is to be used if the run-time system's setup is not supported. We link in the fallback language as the **LANG_DEF.CPP** file. By default, this file contains English translations. If we wish to use another language, we need only copy the desired **LANG_<ISO>.CPP** file from the **ZINC\SOURCE\INTL** directory to the **ZINC\SOURCE** directory, rename it to **LANG_DEF.CPP,** and recompile the library. Similarly, **LOC_DEF.CPP** contains the fallback locale**,** which we can change by copying the desired **LOC_<ISO>.CPP** file from **ZINC\SOURCE\INTL** directory to the **ZINC\SOURCE** directory, renaming it to **LOC_DEF.CPP**, and recompiling. The images used to draw objects are in **IMG_DEF.CPP**. If we wish to use different fallback images, copy the desired **IMG_<ISO>.CPP** file from **ZINC\SOURCE\INTL** to **ZINC\SOURCE**, rename it to **IMG_DEF.CPP** and recompile the library. In order for our program to support any of the languages and locales that Zinc supports, the application must find the **I18N.DAT** file at run time. This file contains all the localization data for the various languages and locales, as well as the map tables for using hardware character sets.

We can easily change the language, locale, or images of any single object or of the entire application. To change the language for the entire application, simply call *languageManager.**LoadDefaultLanguage( )**—languageManager* is a static, global variable of type **ZIL_LANGUAGE_MANAGER**—and pass it the two-letter ISO language code. Similarly, we can call the *localManager.LoadDefaultLocale()* function to set the application's locale, and *decorationsManager.LoadDefaultDecorations()* will set the decorations for the entire application. Each object that uses language, locale, or decoration information also has a **SetLanguage**( ), **SetLocale**( ), or **SetDecorations**( ) function that can set the localization data for that instance of the object. We will discuss this further as part of the tutorial later in this book.

Localizing our objects

If we are hardcoding data for our objects by embedding the data in the code or by placing all the data in a single module or table, localizing is straightforward. If we are looking data up at run time, however, we need to know which language and locale should be presented.

Detecting the language

We may want to detect at run time which language the environment is using. We can do this by inspecting the global *languageManager* variable, which is a static variable of type **ZIL_LANGUAGE_MANAGER**. *languageManager.defaultName* is the two-letter ISO language code identifying which language is in use.

One way to use the language code is to use it as an extension on a data file previously translated to the language specified by that language code. For example, our data file **SUPPORT.FR** may contain the French translations of all the windows and text in our application. We may also have a file called **SUPPORT.DE** that contains the German translations. After determining the language code, we can create a file name, which in turn sets up the **UI-_WINDOW_OBJECT::***defaultStorage*. The system will then use that data file automatically when loading windows. For a complete list of the ISO language codes, see "Appendix H—ISO Language Codes" in the *Programmer's Reference, Volume 2.*

Detecting the locale

We may want to detect at run time which locale the environment is using, which will affect how our program formats dates, numbers, and times. Typically, we will not care about the locale, since Zinc objects format themselves. If we do need to know the locale, we can find out by inspecting the global *localeManager.defaultName* variable, which will contain the two-letter ISO locale or country code. For a complete list of ISO country codes see "Appendix G—ISO Country Codes" in the *Programmer's Reference, Vol. 2.*

Building our application

There are two considerations when building our applications. The first is whether the application is using the Unicode character set. Zinc applications do not need the Unicode character set to be globalized. Support for this character set is required only if the application supports double-byte characters. If the application does support Unicode, though, the Zinc library must be compiled for Unicode support. This is done by defining the **ZIL_UNICODE** precompiler variable in the **UI_ENV.HPP** source file and rebuilding Zinc Application Framework's libraries. **ZIL_UNICODE** must be defined when building our application, as well.

The second consideration when building our application is how we choose to localize it. If we choose to compile in a file containing the library globalization data for a specific locale, we must link it in. If we choose to either hardcode the globalization data or access it at run time, we need take no other steps at compile time.

Shipping our application

In this section we talk about the files that we must ship with our application. We discuss the requirements for both 8-bit character and 16-bit character modes.

Non-Unicode applications

When shipping non-Unicode applications, we must ship the following files:

- The executable (**.EXE**);
- The data file (**.DAT**) containing resources created in the Designer, if we use one.
- Any data files (**I18N.DAT**) with the library globalization data, if different than our data file.

Required files for Unicode applications

When shipping Unicode applications we must ship the following files:

- The executable (**.EXE**);
- The data file (**.DAT**) containing resources created in the Designer, if we use one.
- Any data files (**I18N.DAT**) with the library globalization data, if different than our data file.

If our application uses the GFX graphics library to support DOS graphics, we must also ship the **UNICODE.FNT** file. This file contains a font table for most languages in the Unicode character set. Note that the GFX graphics library is the only Zinc-supported DOS graphics library that supports the Unicode character set. Therefore if our application must support DOS graphics, we must use the GFX graphics library.

Conclusion

In this chapter, we've discussed how to globalize a Zinc application, including enabling our objects and localizing our code for Unicode.

This is also the end of the section on Zinc's main concepts. In this section we've learned about Zinc's architecture, its windows and window objects, event handling and mapping, library classes, C++ features of Zinc, and globalizing a program. In the next section, we'll begin learning how to write Zinc programs.

section two

Zinc programming

Chapter 9

"Hello, Universe!"

Welcome to the section of this book on Zinc programming. This section is full of tutorials and tips on how to write full-featured Zinc applications. We'll start out by writing a small Zinc program.

Most programmers who learn a new language or programming environment will write a program that prints the phrase, "Hello, world," in a terminal window. But since our Zinc program can run on nearly every major platform in the computer marketplace, we'll print the phrase, "Hello, Universe!" into a text field in a graphical window.

Key Concepts

Using **UI_APPLICATION**

Learning to write a simple Zinc application

Shutting down an application

What we'll do

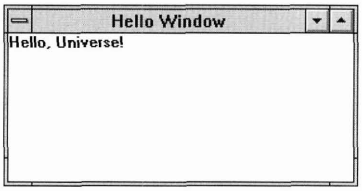
Here are the steps we'll take in writing **HELLO1.CPP**.

1. Load the library called **UI_WIN.HPP** to use Zinc's window object definitions and implementations.
2. Create a function called **UI_APPLICATION::Main( )**, which sets up the infrastructure of writing portable, event-driven applications.
3. Create a generic window with the title "Hello Window."
4. Add to the window the text "Hello, Universe!", and some other data, including flags.
5. Add the window to the Window Manager, the control center for all windows and window objects.
6. Call a function called **Control( )**, which acts as the main event loop, getting events from the system and dispatching them to the application.

Here's the source code to **HELLO1.CPP**:

```
//  HELLO1.CPP (HELLO)
//  COPYRIGHT (C) 1990/1994.  All Rights Reserved.
//  Zinc Software Incorporated.  Pleasant Grove, Utah  USA
//  May be freely copied, used and distributed.
#include <ui_win.hpp>
int UI_APPLICATION::Main(void)
{
UIW_WINDOW *window = UIW_WINDOW::Generic(2, 2, 40, 6,
  "Hello Window");
*window
  + new UIW_TEXT(0, 0, 0, 0, "Hello, Universe!", 256,
    WNF_NO_FLAGS, WOF_NON_FIELD_REGION);
*windowManager
  + window;
Control();
return (0);
}
```

When we compile the program and run the executable, we see a screen like this:



Hello, Universe!

Include files

The first step we took in **HELLO1.CPP** is declaring the include files **UI_WIN.HPP**.

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

In Zinc, **UI_WIN.HPP** is the header file that, among other things, contains the definitions for window objects. Zinc also contains other include files in addition to **UI_WIN.HPP** for handling other types of Zinc information, for example, information for list objects, for screen displays, and so forth.

These include files initialize information specific to Zinc-supported compilers, freeing us from worrying about which files to include, and which not to include. We can always use the same headers no matter what compiler we use, making writing Zinc programs easier.

One Zinc include file, **UI_ENV.HPP**, initializes information for specific environments. For example, it includes **WINDOWS.H**, which contains information for Microsoft Windows; **OS2.H**, which contains information for OS/2, and so forth. This is what allows your Zinc application to compile for different environments. Here is a list of all Zinc's include files, and what they do.

TABLE 11. **Include files in Zinc**

Include file	*What it contains or defines*
UI_ENV.HPP	All values and information for specific compilers and environments
UI_GEN.HPP	Low-level classes like user interface elements and lists
UI_DSP.HPP	How to handle screen displays for different environments
UI_MAP.HPP	Keyboard scan codes and virtual key mappings

TABLE 11. Include files in Zinc

Include file	*What it contains or defines*
UI_EVT.HPP	Basic infrastructure for event handling
UI_WIN.HPP	Window objects

When we include the **UI_WIN.HPP** file, we also include the **UI_EVT.HPP**, **UI_MAP.HPP, UI_DSP.HPP**, **UI_GEN.HPP**, and **UI_ENV.HPP** files. This means we need only include **UI_WIN.HPP** to use *all* of Zinc's include files.; under normal programming circumstances we'll find it highly unlikely that we'll have to include any of those classes separately from **UI_WIN.HPP**.

A new Main( )

The next step we took in **HELLO1.CPP** after declaring include files was to create a function called **Main()** from the class **UI_APPLICATION**. Using this function will save a lot of code if we want to write an application that takes advantage of Zinc's benefits. Here's the code again:

```
int UI_APPLICATION::Main(void)
{
  UIW_WINDOW *window = UIW_WINDOW::Generic(2, 2, 40, 6,
    "Hello Window");
  *window
    + new UIW_TEXT(0, 0, 0, 0, "Hello, Universe!", 256,
      WNF_NO_FLAGS, WOF_NON_FIELD_REGION);
  *windowManager
    + window;
  Control();
  return (0);
}
```

Here's what this function does. Any meaningful Zinc program like **HELLO1.CPP** uses a certain amount of infrastructure to display information on the screen; enable windows to respond to events from the mouse, keyboard, and the cursor; and to use and manage window objects—and to do all these things across every environment Zinc supports. We could build that infrastructure by hand for whatever environment under which we'd like to run our applications, or we could merely use the **UI_APPLICATION::Main( )** function to create the infrastructure for us for every environment Zinc supports.

In **HELLO1.CPP**, **UI_APPLICATION** saves us from having to write a lot of code, specifically code for managing windows and events. For example, the Window Manager, the part of **HELLO1.CPP**'s infrastructure that handles incoming events from the Event Manager, comes from **UI_APPLICATION**. Also, **Control( )**, the function that contains the main event loop, comes from **UI_APPLICATION**. In short, **UI_APPLICATION** is a quick and easy way to create that infrastructure so we don't have to create our own—and the infrastructure we needn't create is the one that won't break. If you want to know more about **UI_APPLICATION::Main( )**, hang on—we'll discuss it further in just a moment.

Creating a window and adding a text field

The next step we took in **HELLO1.CPP** in **UI_APPLICATION::Main( )** was to create a new window. To do this, we used a function in the Zinc **UIW_WINDOW** class.

```
UIW_WINDOW *window = UIW_WINDOW::Generic(2, 2, 40, 6,
  "Hello Window");
```

UIW_WINDOW is Zinc's class for working with windows and window fields that we display on the screen. We've created a pointer to the **UIW_WINDOW** class called **window**, then called the class's member function **Generic( )** with some parameters, and assigned the result to **window**. When we called **Generic( )** with those parameters and assigned the result to **window**, in a short line of code we created a full-fledged window with a border, a maximize button, a minimize button, a system button, and a title.

The next thing we did was to put some text into our window. Notice how we added a pointer to an instance of a **UIW_TEXT** class to **window** with the overloaded **+** operator:

```
*window
  + new UIW_TEXT(0, 0, 0, 0, "Hello, Universe!", 256,
    WNF_NO_FLAGS, WOF_NON_FIELD_REGION);
```

The **UIW_TEXT** constructor contains more parameters, one of which is the text, "Hello, Universe!," that we've seen displayed inside the window. The instance of **UIW_TEXT** also contains two *flags*, values that, when turned on or off, affect the behavior of the object.

The first flag, *WNF_NO_FLAGS*, tells the object not to associate any special flags with the text object. The second flag, *WOF_NON_FIELD_REGION*, tells the object to ignore any parameters it receives concerning where to display itself and to use the remaining space in the window. If we hadn't included this flag, the object would display "Hello, Universe!" wherever the positional parameters told it to.

The last thing we did with our window was attach it to the Window Manager.

```
*windowManager
  + window;
```

The Window Manager is Zinc's method of managing how windows behave, including their position and priority, and of accepting events from the Event Manager and passing them in turn to the windows that need to respond to those event. By attaching our window to the Window Manager, we placed the window and its subobjects on the screen and gave it the ability to accept events like "move the window."

Let's take a step back from our code and look at a couple things. Notice that we've followed a certain order when we worked with windows. We first created the window, then we attached the text to the window, and after we finished taking care of the window we attached the whole concatenation to the Window Manager. We followed this certain order because we wanted the window to appear on the screen all at once, instead of a piece at a time. If we hadn't followed this certain order the window would have displayed itself in a messy, semi-organized manner.

Next, notice that we didn't explicitly create an instance of the Window Manager, though we know one exists, since we added a window to it. We didn't have to create an instance; **UI_APPLICATION::Main( )** did it for us. Again, **UI_APPLICATION::Main( )** has saved us code while writing **HELLO1.CPP.**

Responding to events

The next step we took in **HELLO1.CPP** after creating a window and adding a text field was to call a function called **Control()**. This function is the main event loop, the central structure of **HELLO1.CPP**, which takes over the application and waits for the user to create events.

```
    Control();
```

When the user sends the "quit" event by closing the window or by pressing the appropriate keys, the main event loop quits the program.

The main event loop is how Zinc gives us the ability to easily write event-driven programs, one of Zinc's major design goals—and by using the **Control( )** function we'll save time and code that we'd otherwise spend writing a main event loop by hand. Once we call **Control( )**, we can sit back and let **UI_APPLICATION::Main( )** get the events from the queue and route them to the window we just added to the Window Manager.

Under the hood of ***UI_APPLICATION::Main( )***

A little while ago, we introduced the idea of **UI_APPLICATION::Main( )** without saying much about it. Now let's fill in the details of what's going on under the hood.

What UI_APP does

Right after we declared the proper include file, we created the function

```
int UI_APPLICATION::Main(void)
```

Notice that this function has displaced the **main( )** function we'd write in a non-Zinc program. Also, it comes from the Zinc class, **UI_APPLICATION**. How does **UI_APPLICATION::Main( )** work?

Every meaningful Zinc program includes a certain amount of infrastructure to display information on the screen; respond to events from the mouse, keyboard, and the cursor; and manage window objects. What Zinc has given us with **UI_APPLICATION** is a single function call to set up that infrastructure for use, giving us more time to write the core engine of our program.

The *Programmer's Reference* tells us that the class initializes the standard control objects. This means using the **UI_APPLICATION** class will initialize:

- the screen display;
- the Event Manager; and
- the Window Manager

In addition to setting up the infrastructure for us, **UI_APPLICATION** gives us *automatic portability between environments*. Using **UI_APPLICATION** lets us simply compile **HELLO1.CPP** to run under any environment *with no modifications* because this class contains the code needed to compile under all supported environments. If we didn't use **UI_APPLICATION::Main( )**, we'd have to duplicate Zinc's efforts to compile our program under Microsoft Windows or any other supported platform.

Main( )

Now that we know more about what **UI_APPLICATION** does for us, let's look up in the *Programmer's Reference* the actual function we called from the **UI_APPLICATION** class—**Main( )**. We'll find that **Main( )** does two things: it

- sets up the initial application windows
- calls or creates the main event loop.

Here's why **Main( )** takes the place of **main( )** in our program. First, all operating environments don't support **main( )** transparently—and every C++ program ever written must have that function or it'll refuse to compile. For example, if you've written programs for Microsoft Windows, you'll know you need to start out your program with the function **WinMain( )**, not **main( )**.

When we write a Windows program, we must include some special Microsoft libraries that, among other things, provide a **main( )** function, which then call an undefined function called **WinMain( )**, which, of course, we define ourselves. This satisfies the requirement of C++, and therefore your program will compile.

When we use **UI_APPLICATION::Main( )** in a Zinc program, we're doing something conceptually similar to what we just discussed. We're *abstracting* the idea of **main( )** and **WinMain( )** and *generalizing* the code required to handle those functions in multiple environments. Obviously, Zinc has done us a good turn by giving us one function for handling the **Main( )** function in programs that run under multiple operating systems.

Event flow and ***Control( )***

One of the key concepts of **HELLO1.CPP**, and of Zinc applications in general is that all Zinc applications are by design *event driven*. Zinc programs wait for the user to create an event by pressing a key on the keyboard, or manipulating the mouse—and when the user creates an event, the program reacts by calling the appropriate function. This program flow that consists of reacting to user input is the essence of Zinc's event-driven architecture.

If we wanted, we could get events and route them by hand—later we'll learn how. But the **UI_APPLICATION** class allows us to include in **HELLO1.CPP** a function called **Control( )** that automatically gets and routes events for us. All we must do is call **Control( )** inside the **Main( )** curly braces.

But **Control( )** may still seem a little mysterious. Here's what's going on inside the main event loop **Control( )** creates:

1. First, the user creates an event by pressing a keyboard key or by manipulating the mouse.
2. Next, the loop gets the event from the Event Manager and sends it to the Window Manager. The the Window Manager sends that information to the "Hello, Universe!" window. For example, if we click on the system button, the button that closes the window, we would create a "close" event. In turn, the Event Manager would get this event, and pass it to the Window Manager.

3. Last, the **Control( )** function examines the Window Manager's return code. If it sees the "quit" event or if it sees that there are no more windows attached to the Window Manager, it will quit the program. Otherwise it will return to the first step—and start the main event loop all over again.

HELLO1.CPP without UI_APPLICATION

If you looked through the source code to **UI_APPLICATION**, you'd find that Zinc has written a huge amount of code to set up the display, the Event Manager, and the Window Manager for every platform it supports. If we didn't use **UI_APPLICATION**, we'd have to write a significant amount of code to

- set up the display by hand for each environment under which we wanted to run **HELLO1.CPP**;
- add by hand the keyboard, mouse, and cursor to the Event Manager;
- create by hand the Window Manager; and
- write a main event loop for routing events to the Window Manager.

Not only would we have to do these things by hand, we'd have to do some of them once *for each environment* under which we planned to run **HELLO1.CPP**. Like we said before, using **UI_APPLICATION** gives us automatic portability between environments.

The Event Manager

Writing a program without **UI_APPLICATION::Main( )** means setting up the Event Manager by hand and attaching input devices. Setting up the Event Manager by hand requires we use one parameter, *display*, which directs the input devices to display information on the screen. We tell the Event Manager it can accept input from three devices, the keyboard, mouse, and cursor, or we could derive our own input device and add it as well.

HELLO1.CPP only has one window, and so the Window Manager will route all events to that window. In other programs we'll write, however, the Window Manager will route events to the current window. This happens transparently, with or without **UI_APPLICATION::Main( )**, and is a major advantage to Zinc over other environments.

Shutting down HELLO1.CPP

Without **UI_APPLICATION::Main( )**, we'd have to take care of one more thing by hand—deleting the Window Manager, Event Manager, and display to free up memory. We'd use the following code to delete the Window Manager, Event Manager, and display:

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

Notice that we delete the Window Manager, Event Manager, and the display in the reverse order of their construction. Since the Window Manager maintains pointers to the Event Manager and to the display; if we didn't delete it first, it would have valid pointers pointing to deleted objects. Also, we'd have to delete the Event Manager before the display, since the Event Manager maintains a pointer to the display. One thing we *don't* have to delete are objects attached to the event or window managers—the input devices, and the "Hello, Universe!" window, for example, are automatically destroyed when their respective manager is destroyed.

Conclusion

Writing **HELLO1.CPP** using **UI_APPLICATION::Main( )** does a lot of things for us. Zinc recommends that we use this function in our Zinc programs to save us time setting up Zinc's infrastructure, increase reliability by eliminating unneeded code, and making it easy to set up a main event loop.

In the next chapter, we're going to expand **HELLO1.CPP** to include other objects, including a help system and an error system.

Chapter 10

Help and Error Systems

In the last chapter, we learned how to create a window using Zinc. In this chapter, we'll extend **HELLO1.CPP** by adding windowed help and error systems, an exit function, and a "Universe Information" window.

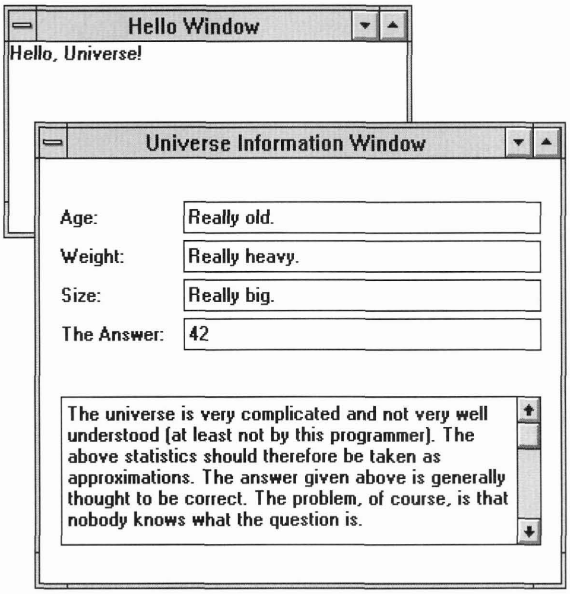
Key Concepts

Using Zinc's help and error systems

Writing an exit function

Creating user interfaces programmatically

The code is located in **\ZINC\TUTOR\HELLO\HELLO2.CPP**. When we compile the program and run the executable, we see a screen like this:



Hello, Universe!

Age: Really old.

Weight: Really heavy.

Size: Really big.

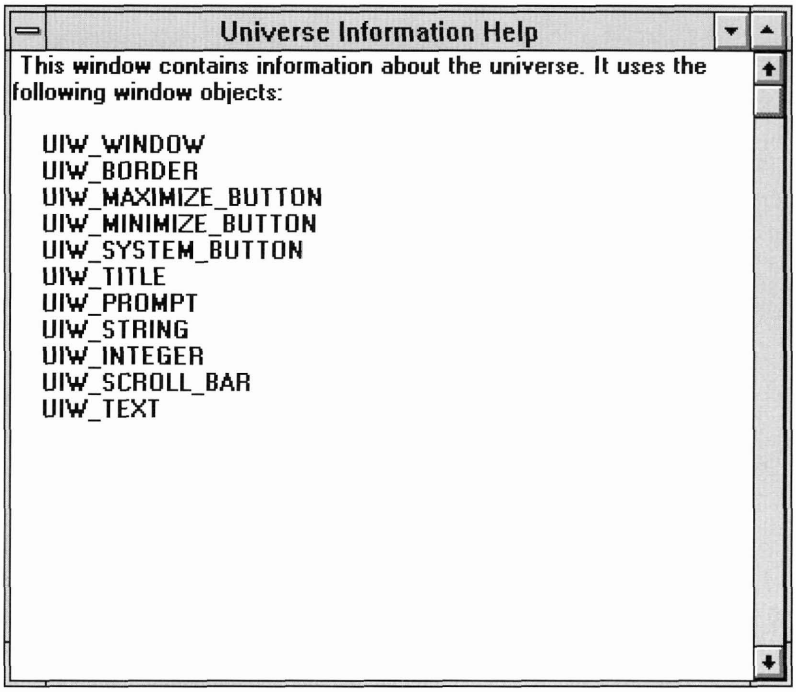
The Answer: 42

The universe is very complicated and not very well understood (at least not by this programmer). The above statistics should therefore be taken as approximations. The answer given above is generally thought to be correct. The problem, of course, is that nobody knows what the question is.

The help system

The help system displays a window containing help information when the user asks for help. Zinc does *not* use the **UI_HELP_SYSTEM** unless we specifically ask for it. This way, we don't have to have the help system modules linked into our executables unless we tell Zinc to include it.

The following figure is an example of a help system window:



This window contains information about the universe. It uses the following window objects:

```
UIW_WINDOW
UIW_BORDER
UIW_MAXIMIZE_BUTTON
UIW_MINIMIZE_BUTTON
UIW_SYSTEM_BUTTON
UIW_TITLE
UIW_PROMPT
UIW_STRING
UIW_INTEGER
UIW_SCROLL_BAR
UIW_TEXT
```

We include the help system in **HELLO2.CPP** by creating a new instance of **UI_HELP_SYSTEM** with three parameters.

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

Here's an explanation of the parameters we use to create a new help system.

- **HELLO.DAT** is the name of the binary help file that the Designer generates from a text file.
- *windowManager* is a pointer to the Window Manager. This argument allows the help system to display information if it encounters an error while initializing the help system.
- *HELP_GENERAL* is the default help context the Window Manager will use if no context-specific help is available when requested. If we were creating a help system with more than one help context, we'd need to specify the name of the help context we wanted to use.

Notice that when we created a **UI_HELP_SYSTEM** object, we assigned it to the static member variable of **UI_WINDOW_OBJECT** called *helpSystem*. The reason we do this is that all of Zinc's window and window objects derive from **UI_WINDOW_OBJECT**, and since *helpSystem* is a static member variable, all windows and window objects we'll create will point to the same help system.

We explained earlier in the chapter that the help system displays a window containing help information when the user asks for help. Here are the steps our window takes when the user requests help.

1. The user sends a message asking for help from an object, which receives the message and, in turn, calls the help system with two arguments.

   ```
   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 two arguments the message uses are

   - *windowManager*, a pointer to the Window Manager; and
   - *helpContext*, the help context that specifies the text to display.

2. Next, the help system attaches its help window to the Window Manager, which displays it:

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

   If the help window is already on the screen, the Window Manager updates its title and help text to current help information.

Where does the help information come from? **HELLO2.CPP** stores help text in the **HELLO.TXT** file, which resides on disk. Here's the help information text:

```
--- HELP_GENERAL
General Help
The second "Hello, Universe!" tutorial shows you
how to create two windows using Zinc Application
Framework and how to initialize the help and error
systems.
For more information about one of the windows
presented in this application press <F1> while
the window is at the front of the display.

--- HELP_HELLO_UNIVERSE
Hello Universe Help
This window simply has a greeting.  It uses
the following window objects:
  UIW_WINDOW          \
  UIW_BORDER          \
  UIW_MAXIMIZE_BUTTON \
  UIW_MINIMIZE_BUTTON \
  UIW_SYSTEM_BUTTON   \
  UIW_TITLE           \
  UIW_TEXT            \

--- HELP_UNIVERSE_INFORMATION
Universe Information Help
This window contains information about the universe.
It uses the following window objects:
  UIW_WINDOW          \
  UIW_BORDER          \
  UIW_MAXIMIZE_BUTTON \
  UIW_MINIMIZE_BUTTON \
  UIW_SYSTEM_BUTTON   \
  UIW_TITLE           \
  UIW_PROMPT          \
  UIW_STRING          \
UIW_SCROLL_BAR        \
UIW_SCROLL_BAR        \
UIW_TEXT              \
```

However, the help system doesn't directly retrieve this text; rather, it retrieves a binary file that we must generate by running the text through the Help Editor in the Designer.

When we convert **HELLO.TXT**, we get the following.

- **HELLO.DAT**, which contains the help information and help contexts. This file is stored in binary form and should not be modified by the programmer. It is the only file **HELLO2.CPP** will use, except, of course, the executable itself.
- **HELLO.HPP**. This file contains the C++ definitions for the help contexts.

Each help context has some elements in common. They are:

- *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 in **HELLO.TXT** is *HELP_GENERAL*.
- *Help context title*. The text in the help window's title bar. It should describe the help context; our first help context title is "General Help," describing help for the entire application.
- *Help information*. The text displayed in the help window. It should contain all the information to help the user with what he is doing.

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

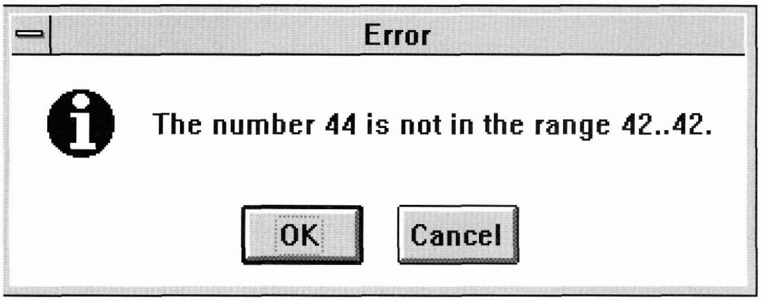
```
#ifdef USE_HELP_CONTEXTS
const UI_HELP_CONTEXT HELP_GENERAL                    = 0x0001;
  // General Help
const UI_HELP_CONTEXT HELP_HELLO_UNIVERSE             = 0x0002;
  // Hello Universe Help
const UI_HELP_CONTEXT HELP_UNIVERSE_INFORMATION       = 0x0003;
  // Universe Information Help
#endif
```

We must include the **.HPP** file in all our programs that use help indexes. Here's the include statement in **HELLO2.CPP**:

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

The error system

The error system resembles the help system in that Zinc doesn't include it unless we specifically ask for it. Below is one of **HELLO2.CPP**'s error windows:



The number 44 is not in the range 42..42.

We can create an error system by setting the value of the **UI_WINDOW_OBJECT::***errorSystem* variable in the same way we did with the help system:

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

Control flow of the error system

Here's what happens when the user creates an error condition:

1. A window object calls the error system. In the example shown above, **UIW_INTEGER** is the window object that calls the error system with an error message from its error message table.

```
    int UIW_DATE::Validate(int processError)
    {
      ...
      ZIL_ICHAR *errStr = myLanguage->GetMessage(errorCode);
      if (errStr)
      {
        WOS_STATUS _woStatus = woStatus;
        woStatus |= WOS_INTERNAL_ACTION;
        UIS_STATUS errorStatus =
          errorSystem->ReportError(windowManager,
          WOS_INVALID, errStr, stringNumber, rBuffer);
        if (!FlagSet(_woStatus, WOS_INTERNAL_ACTION))
          woStatus &= ~WOS_INTERNAL_ACTION;
        if (errorStatus == WOS_INVALID)
          return (-1);// This will cause the number to be
                      // restored.
        woStatus |= WOS_INVALID;
      }
  }
```

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, ZIL_ICHAR *format,
     ...)
   {
     ...
     *windowManager + window;
   ```

   Modal windows prevent the 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 the user acknowledges the error and closes the window by selecting **OK** or **Cancel**.

3. Once the user closes the window by selecting **OK** or **Cancel**, the error system destroys the error window.
4. The object that sent the error request processes the error response and program flow continues.

Exit function

When a program is about to quit, sometimes we may want to call special functions—cleanup functions for example—or perhaps merely inform the user that the program will exit. Zinc has provided a way for us to do so. **UI_-WINDOW_MANAGER** has a special member variable, *exitFunction*, which is a function called when the user attempts to exit the program, or, more precisely, when the Window Manager receives an *L_EXIT* or *L_EXIT_FUNCTION* message. The exit function can have any function name, but must have the following declaration:

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

This declaration gives the exit function pointers to the current display, Event Manager, and Window Manager, which the function can use to draw to the screen, post events, or display windows.

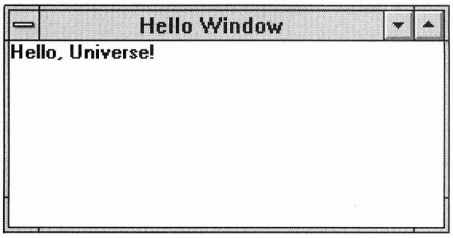
The example above displays a message window with an **OK** button and a **Cancel** button. When the user presses the **OK** button, the program places an *L_EXIT* message on the event queue, and the application ends. Otherwise, the program simply removes the message window and continues. The following code shows the implementation of this exit function:

```
static EVENT_TYPE ExitFunction(UI_DISPLAY *display, UI_EVENT_MANAGER *,
  UI_WINDOW_MANAGER *windowManager)
{
  ZAF_MESSAGE_WINDOW *window =
```

```
    new ZAF_MESSAGE_WINDOW("Hello Universe Tutorial",
      UIW_ICON::_asteriskIconName, ZIL_MSG_OK | ZIL_MSG_CANCEL,
      ZIL_MSG_OK,
      "This will close the Hello Universe application.");
    EVENT_TYPE ccode = S_CONTINUE;
    // Get user response.
    if (window->Control() == ZIL_DLG_OK)
      ccode = L_EXIT;
    // Control() removes window from the Window Manager but doesn't
    // delete it.
    delete window;
    return (ccode);
}
```

Multiple windows

In the last chapter, we created this window with the accompanying code:



```
UIW_WINDOW *window = UIW_WINDOW::Generic(2, 2, 40, 6,
  "Hello Window");
*window
  + new UIW_TEXT(0, 0, 0, 0, "Hello, Universe!", 256,
      WNF_NO_FLAGS, WOF_NON_FIELD_REGION);
```

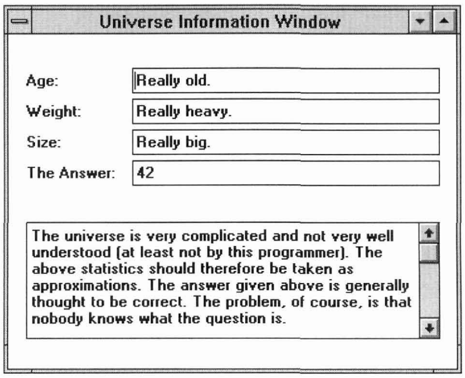
To simplify this window's code, we'll use **Generic( )** static functions. Two Zinc objects have a **Generic( )** function: **UIW_WINDOW** and **UIW-_SYSTEM_BUTTON**. The **UIW_WINDOW::Generic( )** member function automatically creates a window with a border, maximize button, minimize button, system button, and title. The following function shows how we can replace this code:

```
static UIW_WINDOW *HelloWorldWindow1()
{
  // Create the standard Hello World! window.
  UIW_WINDOW *window = UIW_WINDOW::Generic(2, 2, 40, 6,
    "Hello World Window", NULL, WOF_NO_FLAGS, WOAF_NO_FLAGS,
     HELP_HELLO_WORLD);
  // Add the window objects to the window.
  *window
    + new UIW_TEXT(0, 0, 0, 0, "Hello World!", 256,
      WNF_NO_FLAGS, WOF_NON_FIELD_REGION);
    // Return a pointer to the window.
    return (window);
  }
```

Using this method, the **new** operator is not required for window creation. The **UIW_WINDOW::Generic( )** function actually calls the **new** operator for the **UIW_WINDOW** 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 nonfield 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, the border, buttons, and title. Under normal circumstances, a nonfield region object takes up the entire remaining window space, and will cover up any field region objects. However, more than one nonfield region object may reside with field region objects within a single window.

Our Universe Information window is an example of a window that uses field window objects to display information. This window and its code implementation is shown below:



Age:	Really old.
Weight:	Really heavy.
Size:	Really big.
The Answer:	42

The universe is very complicated and not very well understood (at least not by this programmer). The above statistics should therefore be taken as approximations. The answer given above is generally thought to be correct. The problem, of course, is that nobody knows what the question is.

```
static UIW_WINDOW *HelloWindow2(void)
{
  // Create the universe information window.
  UIW_WINDOW *window = UIW_WINDOW::Generic(5, 5, 52, 12,
    "Universe Information Window", ZIL_NULLP(UI_WINDOW_OBJECT),
    WOF_NO_FLAGS, WOAF_NO_SIZE, HELP_UNIVERSE_INFORMATION);
  int answerValue = 42;
  // Add the window objects to the window.
  *window
    + new UIW_PROMPT(2, 1, "Age:")
```

```
        + new UIW_STRING(14, 1, 35, "Really old.", 50)
        + new UIW_PROMPT(2, 2, "Weight:")
        + new UIW_STRING(14, 2, 35, "Really heavy.", 50)
        + new UIW_PROMPT(2, 3, "Size:")
        + new UIW_STRING(14, 3, 35, "Really big.", 50)
        + new UIW_PROMPT(2, 4, "The Answer:")
        + new UIW_INTEGER(14, 4, 35, &answerValue, "42..42")
        + &(*new UIW_TEXT(2, 6, 47, 4,
      "The universe is very complicated and not very well understood "
      "(at least not by this programmer). The above statistics should"
      "therefore be taken as approximations. The answer given above "
      "is generally thought to be correct. The problem, of course, is"
      "that nobody knows what the question is.",
          2048, WNF_NO_FLAGS, WOF_BORDER)
        + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_VERTICAL));
      // Return a pointer to the window.
      return (window);
    }
```

Notice the difference between the code to create the text object in the first window . . .

```
      new UIW_TEXT(0, 0, 0, 0, "Hello, Universe!", 256,
          WNF_NO_FLAGS, WOF_NON_FIELD_REGION);
```

. . . and the code to create the text object in the second window.

```
      new UIW_TEXT(2, 6, 47, 4,
      "The universe is very complicated and not very well understood "
      "(at least not by this programmer). The above statistics should"
      "therefore be taken as approximations. The answer given above "
      "is generally thought to be correct. The problem, of course, is"
      "that nobody knows what the question is.",
          2048, WNF_NO_FLAGS, WOF_BORDER)
```

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

Program flow

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 the user requests help, when the help or error system adds its window to the Window Manager—and then the program may display up to four windows.

Cleanup

Since we created new help and error systems, we must destroy them at the end of the application. Although they are members of **UI_WINDOW_-OBJECT**, they must be destroyed explicitly since they are static.

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

Conclusion

In this chapter, we learned how to create a user interface programmatically, and to use a help and error system in a Zinc application. In the next chapter, we'll create the same user interface using Zinc Designer, an interactive tool for creating user interfaces visually.

Chapter 11

Using the Designer

In the last tutorial, we created a user interface programmatically. In this tutorial, we'll create the same window in a manner of minutes and with a single line of code using Zinc Designer.

The code for this tutorial is in **\ZINC\TUTOR\HELLO\HELLO3.CPP**.

Key Concepts

Working with persistent objects

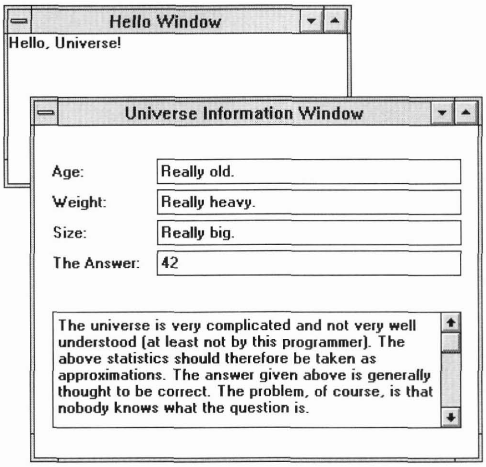
Creating user interfaces with Zinc Designer

What we'll do

We'll use Zinc Designer to accomplish nearly all of the steps in this tutorial.

1. Using the Designer, create a new persistent object file.
2. Using the Designer, create a window and edit its information.
3. Using the Designer, create a window object and edit its information.

Once the application is running, we should see the following on the screen:



Hello, Universe!

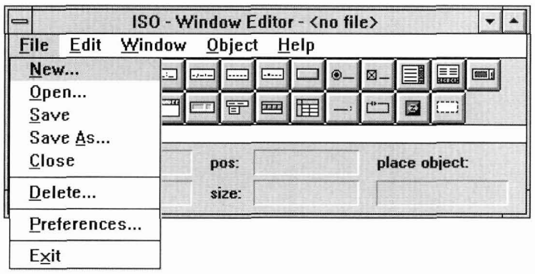
Age: Really old.
Weight: Really heavy.
Size: Really big.
The Answer: 42

The universe is very complicated and not very well understood (at least not by this programmer). The above statistics should therefore be taken as approximations. The answer given above is generally thought to be correct. The problem, of course, is that nobody knows what the question is.

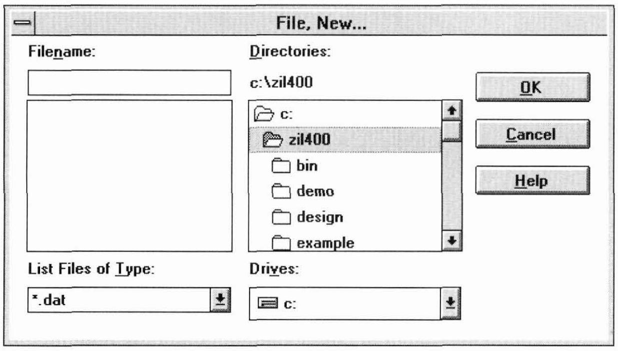
Creating a file

Follow these steps to create a persistent object that will store the "Hello, Universe!" windows:

1. Select **File** from the main control menu. This displays the following menu:



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 **Filename** prompt. This is the file name the Designer calls our file when we save it to disk.

4. Create the file by selecting the **OK** button. Now Zinc Designer does the following:

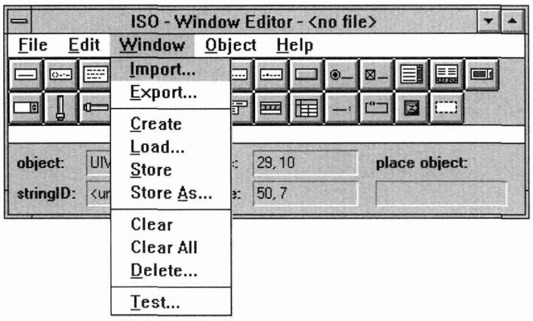
   - Creates a **HELLO.DAT** file that will store the "Hello, Universe!" windows;
   - Removes the **New..** window from the screen;

- Updates the control window's title to reflect the active **HELLO.DAT** file.

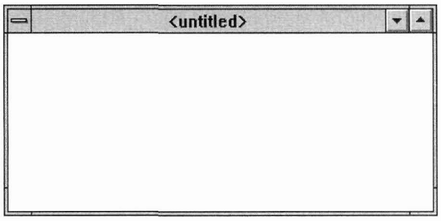
Creating a window

.We created a window and its window objects in the last chapter by writing some code. Now we're going to create the same window and window objects with Zinc Designer by following these steps.

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

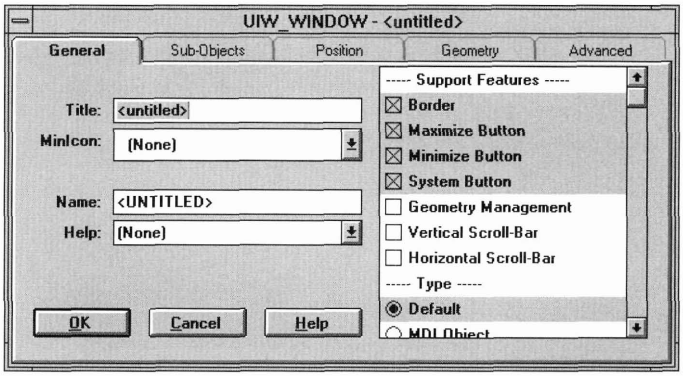


2. Select **Create** from the pop-up window. Now a generic window appears on the screen:



3. Size the window to a size that looks about right. You can adjust the size later if necessary. You should make the window large enough to handle the new title information and default "Hello, Universe!" 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:



Title: <untitled>
MinIcon: (None)
Name: <UNTITLED>
Help: (None)

----- Support Features -----
☒ Border
☒ Maximize Button
☒ Minimize Button
☒ System Button
☐ Geometry Management
☐ Vertical Scroll-Bar
☐ Horizontal Scroll-Bar
----- Type -----
◉ Default
○ MDI Object

5. Enter

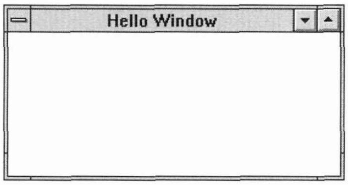
```
Hello Window
```

in the **Title:** field.

6. Enter the window identification by typing **HELLO_UNIVERSE_-WINDOW** in the **Name:** field.

7. Save the identification by selecting the **OK** button.

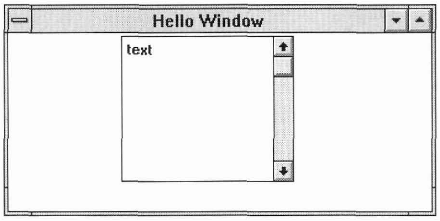
The window should now look like this:



Creating a window object

We create the "Hello, Universe!" text the same way we created the window in the last few steps:

1. Select the **Text** object button from the main control window's toolbar or select **Object | Input | Text** from the main control menu.
2. Place the text object in the middle of the "Hello, Universe!" window. The window should now have a text field within its border:



3. Change the text object's default information by
   - calling the text object editor by double-clicking the left mouse button on the text object
   - typing

     ```
     Hello, Universe!
     ```

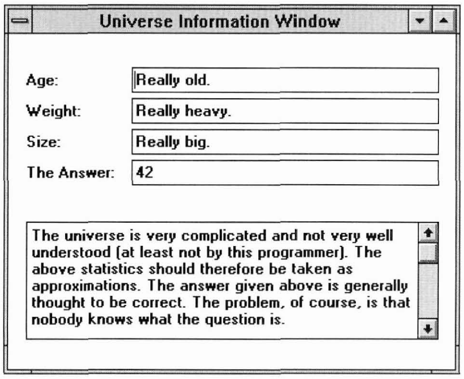
     in the field adjacent to the **Text:** prompt
   - typing

     ```
     256
     ```

     in the field adjacent to the **Length:** prompt
   - turning off the **vertical scroll bar** option
   - turning off the **Don't wrap text** option
   - turning on the nonfield region option in the **Advanced** options

Creating additional windows

The universe information window that we created programmatically in the last chapter looked like this:



Age: Really old.
Weight: Really heavy.
Size: Really big.
The Answer: 42

The universe is very complicated and not very well understood (at least not by this programmer). The above statistics should therefore be taken as approximations. The answer given above is generally thought to be correct. The problem, of course, is that nobody knows what the question is.

Follow these steps to create this window in the Designer:

1. Create the window by selecting **<u>W</u>indow** | **<u>C</u>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. Use the window editor to change the window title to read

   ```
   Universe Information Window
   ```
3. Change the window identification by calling the window editor and entering **UNIVERSE_INFORMATION_WINDOW** as the **Name**.
4. Select **<u>O</u>k** to exit the window editor.
5. Create the age prompt by selecting the Prompt button from the toolbar or selecting **<u>O</u>bject** | **<u>S</u>tatic** | **<u>P</u>rompt** from the control menu and place the field at the left-top corner of the window. Call the prompt editor by double-clicking on it with the mouse or by selecting **<u>E</u>dit** | **<u>O</u>bject** from the control menu and enter

   ```
   Age:
   ```

   as the prompt's text.
6. Create the age string field and place it next to the age prompt. Enter

   ```
   50
   ```

   as the default length for the string field, and enter

   ```
   Really old.
   ```

   as the string's text.

7. Create the weight prompt and place it under the **Age** prompt. Change the prompt's text to

   ```
   Weight:.
   ```

q&a

"How do I create an icon in the Designer, create a window in code, and have the window minimize to the icon?"

First, create an icon in the Image Editor of the Designer and save it to a **.DAT** file. You must save the icon in the Image Editor, and save the file opened in the Designer.

Next, assign **UI_WINDOW_OBJECT::defaultStorage** to point to the **.DAT** file containing the icon image.

After that, create the window in code. Create a **UIW_ICON** object with the saved image and set the icon object's **ICF_MINIMIZE_OBJECT** and **WOF_SUPPORT_OBJECT** flags. Add the icon object to the window. Add the window to the Window Manager.

To test your handiwork, minimize the window to see the icon with its assigned image.

8. Create the weight string field and place it next to the weight prompt. Enter

   ```
   50
   ```

   as the default length for the string, and enter

   ```
   Really heavy.
   ```

   as the string's text.
9. Create the size prompt and place it under the weight prompt. Enter

   ```
   Size:
   ```

   as the prompt's text.
10. Create the size string and place it next to the size prompt. Set the length for this object to

    ```
    50
    ```

    and enter

```
Really big.
```

as the string's text.

11. Create the **Universe Information** text field and place it under the size prompt. Set the length to

    ```
    256
    ```

    and the default text to

    ```
    The universe is very complicated and not very well understood
    (at least not by this programmer). The above statistics should
    therefore be taken as approximations. The answer given above is
    generally thought to be correct. The problem, of course, is that
    nobody knows what the question is.
    ```

    To add a vertical scroll bar to the text field, check the **Vertical Scroll Bar** checkbox.

12. Select the **OK** button to complete the changes to the window.

Now we're finished creating the Universe Information window.

Saving the file

The "Hello, Universe!" windows are saved when we select **File | Save** from the control menu. Here's what Zinc Designer does when the windows are saved:

*Updates the **HELLO.DAT** file.* Contains the binary information associated with the objects saved during the design session. Help contexts and window objects like those in this and the last chapter live in the same **.DAT** file.

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

*Creates the **HELLO.HPP** file.* Contains the numeric identifications, which are IDs associated with those strings we entered next to the **stringID** prompt and the help context definitions. The string identification for each field within a window is unique. Items within subwindows, 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 universe information windows to the Window Manager:

```
// Add the two windows to the window manager.
UI_WINDOW_OBJECT *window1 =
  UI_WINDOW_OBJECT::New("hello.dat~HELLO_UNIVERSE_WINDOW");
UI_WINDOW_OBJECT *window2 =
  UI_WINDOW_OBJECT::New("hello.dat~UNIVERSE_INFORMATION_WINDOW");
*windowManager
  + window1
  + window2;
```

In the code above, we retrieve **HELLO_UNIVERSE_WINDOW** and **UNIVERSE_INFORMATION_WINDOW** from the **HELLO.DAT** data file, then add them to the Window Manager.

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

```
  *windowManager
    + UI_WINDOW_OBJECT::New("hello.dat~HELLO_UNIVERSE_WINDOW")
    +
UI_WINDOW_OBJECT::New("hello.dat~UNIVERSE_INFORMATION_WINDOW");
```

This method allows for error correction. For example, if 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.

As we mentioned before, Zinc Designer created a **HELLO.CPP** code file. This file must be compiled and linked with the **HELLO3** executable. It contains an object table, used by window object constructors to read class information from the data file.

Run time features

The persistent window objects contain all the information necessary to ensure that the application runs as if we created the object programmatically, as we did in the previous tutorial.

Conclusion

In this chapter we learned how to create a window in the Designer that we created earlier programmatically. The Designer is a major benefit, since creating windows and window objects becomes easier when we can manipulate them on screen the same way we would work with them while running an application.

In the next chapter, we'll learn more about writing Zinc applications that use events in both top-down and bottom-up operating environments.

Chapter 12

Event flow

This tutorial demonstrates how Zinc handles system events in top-down and bottom-up operating environments. When we're finished, we should understand how window objects display information and receive input from the user; how to check data entry with user functions; and how to write a main event loop. Here we'll examine a dictionary program called **WORD2.EXE**.

Key Concepts

working with top-down and bottom-up event flow

writing a user function to validate input

What we'll do

Here are the steps we'll take in writing **WORD2.CPP**.

1. Create the **DICTIONARY_WINDOW** class and all its member functions.
2. Create an instance of the **DICTIONARY_WINDOW** and add it to the Window Manager.
3. The **DICTIONARY_WINDOW** creates a **DICTIONARY**, which loads the data from disk.
4. When the user types a word and hits <Enter> we'll look the word up in the dictionary.

Running the program

To use the dictionary program, compile and run the application **WORD2.EXE**. This program only knows the word *good*, *bad*, *begin*, and *end*. To look up a word, type it in the **Enter a word** field and press <Enter>. If the word is in the dictionary, the program will display the definition, antonyms, and synonyms. If the word is not in the dictionary, it will display the error message, "That word was not found in the dictionary." When you are finished using the dictionary, exit the program by closing the window.

Source code

The source code for this program is located in **\ZINC\TUTOR\WORD**, and contains the following files:

- **WORD2.CPP.** Contains **UI_APPLICATION::Main( )**, as well as the implementation of the **DICTIONARY_WINDOW**, **DICTIONARY**, and **D_WORD** classes.
- **WORD2.HPP.** Contains the declarations for the **DICTIONARY_-WINDOW**, **DICTIONARY, D_WORD**, and **_WORD** classes.
- **WORD.DCT.** Contains the dictionary database file.
- ***.DEF, *.RC.** Contains the environment-specific definitions and resources for compiling Zinc programs for environments other than DOS.
- ***.MAK.** Contains the compiler-dependent makefiles associated with the Word program. Consult "Appendix A—Compiler Considerations" for information on compiling for each Zinc-supported platform.

Class definitions

The dictionary window is implemented in a class called **DICTIONARY_WINDOW**. Here's its definition:

```
class DICTIONARY_WINDOW : public UIW_WINDOW
  {
  public:
    DICTIONARY_WINDOW(void);
    ~DICTIONARY_WINDOW(void);
    int dictionaryOpened;
  private:
    DICTIONARY *dictionary;
    UIW_STRING *inputField;
    UIW_TEXT *definitionField;
    UIW_STRING *antonymField;
    UIW_STRING *synonymField;
    static EVENT_TYPE LookUpWord(UI_WINDOW_OBJECT *string,
      UI_EVENT &event, EVENT_TYPE ccode);
  };
```

DICTIONARY_WINDOW uses the following member variables:

- *dictionaryOpened*, which indicates if the data file 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*, the pointer to the dictionary that is created in the constructor for **DICTIONARY_WINDOW**. This variable is used only by the **DICTIONARY_WINDOW** class and therefore is private.
- *inputField*, 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*, 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*, 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*, 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.

Below is the definition for the **DICTIONARY**:

```
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 variables:

- *opened*, which 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.

D_WORD is the dictionary class that contains the words in the dictionary. Below is the definition for the **D_WORD** class:

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

_WORD is a support class used to hold the words in the antonym and synonym lists:

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

```
    _WORD(const char *_string);
    ~_WORD(void);
    _WORD *Next(void);
  };
```

_WORD uses the following member variable:

- *string*, a character string that contains a word.

Creating the window

We start out by deriving our **DICTIONARY_WINDOW** class from the Zinc class **UIW_WINDOW**. Instead of using an instance of the existing **UIW_WINDOW** class, our derived class will also handle input from and output to the window fields and communicate with our dictionary.

When our program calls the **DICTIONARY_WINDOW** constructor, it creates the dictionary window. The **DICTIONARY_WINDOW** creates each of the fields and adds them to the window using the C++ reserved word *this* and the overloaded + operator. The **DICTIONARY_WINDOW** constructor is shown below:

```
DICTIONARY_WINDOW::DICTIONARY_WINDOW(void) : UIW_WINDOW(16, 6, 41, 14)
  {
    ...
  if (dictionaryOpened)
  {

    // Create the window fields.
    inputField = new UIW_STRING(17, 1, 20, "", 40,
      STF_NO_FLAGS, WOF_BORDER | WOF_AUTO_CLEAR,
      DICTIONARY_WINDOW::LookUpWord);
    definitionField = new UIW_TEXT(17, 3, 20, 4,"", 100,
      TXF_NO_FLAGS, WOF_BORDER);
    antonymField = new UIW_STRING(17, 8, 20, "", 50, TXF_NO_FLAGS,
      WOF_BORDER);
    synonymField = new UIW_STRING(17, 10, 20, "", 50,
      TXF_NO_FLAGS, WOF_BORDER);
    *this
      + new UIW_BORDER
      + new UIW_MAXIMIZE_BUTTON
      + new UIW_MINIMIZE_BUTTON
      + new UIW_SYSTEM_BUTTON
```

```
        + new UIW_TITLE("Dictionary")
        + new UIW_PROMPT(2, 1, "Enter a word:")
        + inputField
        + new UIW_PROMPT(2, 3, "Definition:")
        + definitionField
        + new UIW_PROMPT(2, 8, "Antonyms:")
        + antonymField
        + new UIW_PROMPT(2, 10, "Synonyms:")
        + synonymField;
        ...
    }
}
```

We add the objects in our dictionary window to the window inside the constructor so that when we create our **DICTIONARY_WINDOW** object, we only have to write a few lines of code to display it on the screen. Here's the code taken from the **UI_APPLICATION::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 we add the objects, not in the constructor, but when we create an instance of the **DICTIONARY_WINDOW** class, then we would duplicate code each time we created an instance of that class. Adding the objects inside the constructor lets us write less code and provides a stronger encapsulation of data and code.

The user function

Through the object's constructor, we can assign user functions to a string; the string calls the user function when the string becomes current or noncurrent, or when the user presses <ENTER>. Our user function compares the data in the object's field to the words in the dictionary, and will display either the word's definition and antonyms and synonyms for the word, or it will display an error message that says the word was not found. Let's look at how we assign the user function::

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

When the **UIW_STRING** field is constructed, the last parameter, **DICTIONARY_WINDOW::***LookUpWord*, tells our class instances about the user function. In order for the compiler to generate an address for a user function, we must declare our user functions as static. The user function **LookUpWord( )** has the following parameters that Zinc requires of all user functions:

returnValue$_{\text{out}}$ is the value returned from the operation, and most often *ccode* is the value returned. However, if the operation returns *-1*, the calling window object will be informed that some error occurred and the text will be restored to its previous value.

object$_{\text{in}}$ is a **UI_WINDOW_OBJECT** pointer to the object that invoked this function. In this case, the calling object is a **UIW_STRING** field whose parent is a **DICTIONARY_WINDOW** object.

event$_{\text{in}}$ is the event that caused this function to be called.

ccode$_{\text{in}}$ is the logical interpretation of the event that caused this function to be called.

Here's how we write **LookUpWord( )**:

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

Since the string field calls the user function when it receives the *S_CURRENT*, *S_NON_CURRENT*, or *L_SELECT* messages, the first step is to determine if the *ccode* is *S_CURRENT*. In the dictionary tutorial, unless the input string is selected, the function returns without doing anything.

Here's the initial check in **LookUpWord( )**:

```
// Return if just entering.
if (ccode != L_SELECT)
  return errorCode;
```

As the user function calls the dictionary to verify the input word, it must have a pointer to the current dictionary object. Since the input string is attached to the **DICTIONARY_WINDOW**, we can access the dictionary window using the string's *parent* pointer. Here's how we get a pointer to the correct instance:

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

Now that our user function has the *dictionaryWindow* pointer, we have access to the public variables and functions of the **DICTIONARY_WINDOW** class, including the variable *dictionary*, and it can proceed with calling the dictionary to verify the input word. Now the user function calls the function **DICTIONARY::Get( )** through the *dictionaryWindow* pointer. This function will return a *NULL* if the word is not found, or, if it is found, will return a pointer to a *D_WORD* structure that contains the input word and its associated information; if the return value is a valid pointer, **DICTIONARY::Get( )** writes the word and its antonyms and synonyms to the appropriate window fields by calling each field's **DataSet( )** function. If the word isn't in the dictionary, our program will display an error message and return a -1. Otherwise, we return a 0.

Following events

Now that we understand how the program operates, let's follow how events flow through the system. We can begin by following the event that's created when the user presses the "G" key on the keyboard. Though we'll study our Zinc dictionary running under DOS and Windows, our program running in other operating environments will pass messages in the same way as they do in the DOS and Windows examples, though event messages and their meanings differ.

Event flow—DOS

When the user presses the "G" key, the computer places the character in the computer's keyboard buffer. Here's the code in our dictionary program that actually gets the event from the buffer.

```
EVENT_TYPE ccode;
UI_EVENT event;
do
{
  // Get input from the user.
  eventManager->Get(event);
  // Send event information to the window manager.
  ccode = windowManager->Event(event);
} while (ccode != L_EXIT && ccode != S_NO_OBJECT);
```

As *eventManager*->**Get( )** executes, it polls each of the devices attached to the Event Manager. If the keyboard or another device has placed an event in its buffer, Zinc creates a **UI_EVENT** structure, fills it with the event, and puts on the end of the event queue.

Let's assume that there were no other events on the queue when the program placed the "G" key event on the queue. The **Get( )** function takes the *event* variable and fills it with the "G" event. When program control returns from the **Get( )** function, the call to *windowManager*->**Event( )** passes the "G" event to the Window Manager.

q&a

"How can I intercept an event that is filtered?"

If the message is environment-specific, you must trap it in your derived object's **Event( )**. If you want to convert the message to a logical event, you must place in the event map table assigned to the derived object a mapping for the message

Let's take a look at what happens when the Window Manager receives the "G" key, or any other event under DOS. First, the Window Manager sends the event to the current window object. If the Window Manager can process it, it does. Otherwise it passes it to its current child, which attempts to process it. If it can't, it passes it down, and so forth. This is top-down processing.

If the event carries a specified region like a mouse click, the Window Manager checks to see if another object should become current. If so, the Window Manager makes that object current, and passes the event to that object. If no window can handle the event, the Window Manager just returns an *S_UNKNOWN* message to the system, and the event is ignored.

Now back to our dictionary program. When the user presses the "G" key, the Window Manager's current object is the dictionary window. The window receives the event and sends it to its own current object, the **UIW_STRING** field. The string's **Event( )** function receives the event from the window, and calls **UI_WINDOW_OBJECT::LogicalEvent( )** to look for a logical mapping of the event. Once the **LogicalEvent( )** function determines the event is a "G" keystroke, the character is copied into the string's memory buffer and the string is updated 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—Windows

The Microsoft Windows version of Zinc is simpler than the DOS version. In contrast to DOS, which simply dumps user input in a buffer to wait for a program to use it, Windows handles all the input from the user. This means Zinc need only interpret the messages, and need not handle the events.

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 ported easily to any environment Zinc supports.

In order to follow an event through the Zinc system while running under Windows, we must revisit how Windows passes messages. Windows puts messages on a Windows message queue, which can dispatch those messages directly to the current field on the current object. Messages are passed to an object with a special member function known as a "callback" function, which is the Windows equivalent of Zinc's **Event( )** function.

Now consider the example of the "G" key being pressed while a **UIW_STRING** field is current. Look at the "do" loop in the function **UI_APPLICATION::Main( )**:

```
EVENT_TYPE ccode;
UI_EVENT event;
do
{
  // Get input from the user.
```

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

At some point in the execution of the program, Windows creates a message and puts it on the Windows message queue. When *eventManager*->**Get( )** is called, it doesn't return until Windows has created a message and had put it on the Windows message queue. Once *eventManager*->**Get( )** returns, the call to *windowManager*->**Event( )** instructs Windows to dispatch the message. When Windows dispatches the message, Windows calls the current window object's event function, **UIW_STRING::Event( )** in this case, saying that the user pressed the character "G." When the current window object's event function receives the "G" message, just as in DOS, it determines whether or not it can interpret the event. If it can, it does so, and then passes it back to Windows so that the "G" character may be painted on the screen. If it cannot, it returns an *S_UNKNOWN* and the event goes unprocessed. This is bottom-up processing.

Conclusion

In this chapter, we've seen how objects display information and receive input from the user, how we can use user functions to check data entry, and we've seen more about how Zinc handles events. Further, now that we know how our dictionary application works, we'll find it easier to use in the next chapter, where we'll write a program to store and retrieve data in the Zinc data file.

Chapter 13

The Zinc Data File

In the last chapter, we learned how events flow by watching how our dictionary program responded to events. In this tutorial, we'll use a modified version of the dictionary program to learn how to use the Zinc data file to store data on disk and retrieve it later. To do so, we'll use as a springboard the dictionary program we used in the last chapter. Then we'll modify it to allow us to create and delete our own entries, modify them, and save them to a file on the disk.

Key Concepts

the data file

adding and deleting objects to and from the data file

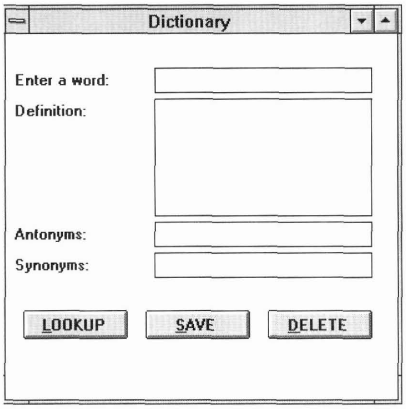
What we'll do

Here are the steps we'll take in writing **WORD3.CPP**.

1. Load the window from the **.DAT** file and create the dictionary. Once we've loaded the window, assign each button the same static user function.
2. Create the member functions.
3. Create an instance of the **DICTIONARY_WINDOW** and add it to the Window Manager.
4. Process user updates and queries.
5. If the user quits the application, commit the data file to disk, close the temporary file, and then free up the memory the program used.

Running the program

Compile the source code and run the executable. You should see the following window on the screen:



At this point, the dictionary database will be empty. To add words to the dictionary, simply type the word, its definition, and an antonym and synonym in the appropriate fields, and press the **Save** button at the bottom of the window. To look up a word you have entered, type it in the **Enter a word:** field and press the **Lookup** button. To delete a word, type it in the **Enter a word:** field and press the **Delete** button When finished using the dictionary, select **Close** from the system button's pop-up menu and exit the program.

Source code

The source code for our tutorial is located in **\ZINC\TUTOR\WORD**, and contains the following files:

- **WORD3.CPP.** Contains the main event loop inside **UI_APPLICATION::Main( )**, as well as the implementation of the **DICTIONARY_WINDOW**, and **D_ENTRY** classes.
- **WORD3.HPP.** The declarations for the **DICTIONARY_WINDOW**, **DICTIONARY,** and **D_ENTRY** classes.
- **WORD_WIN.CPP.** The object table for the objects we created in the Designer.
- **WORD_WIN.DAT.** The data file created in the Designer. Contains the data for creating the dictionary window and its fields.
- **WORD_WIN.HPP.** The header information for **WORD_WIN.DAT** and its help file.
- ***.DEF, *.RC.** The definition and resource files when compiling for different environments.
- ***.MAK.** The compiler-dependent makefiles.

Program flow

Using **UI_APPLICATION::Main( )**'s built-in main event loop, help system, and error handling, our program flow is simple. The first step is to create a new error system. Next we create the **DICTIONARY_WINDOW,** which creates a new dictionary. Once created, we attach the dictionary window to the Window Manager if the load goes well; if the load fails, we ask the error system to report an error to the user. Once we've set these things up, we can turn over event handling to Zinc with **UI_APPLICATION::Control( )**. And when the program flow falls through **Control( )**, we delete the error system we've created.

Here's the code we used to set up **UI_APPLICATION::Main( )**.

```
int UI_APPLICATION::Main(void)
{
  // The UI_APPLICATION constructor automatically initializes the
  // display, eventManager, and windowManager variables.
  // This line fixes linkers that don't look for main in the .LIBs.
  UI_APPLICATION::LinkMain();
  // Initialize the error system.
  UI_WINDOW_OBJECT::errorSystem = new UI_ERROR_SYSTEM;
  // Create the dictionary window.
  DICTIONARY_WINDOW *dictionary = new DICTIONARY_WINDOW("word.dat");
  if (!FlagSet(dictionary->woStatus, WOS_READ_ERROR))
    *windowManager + dictionary;
  else
```

```
    {
      UI_WINDOW_OBJECT::errorSystem->ReportError(windowManager,
        WOS_NO_STATUS,
        "An error was encountered trying to open word_win.dat");
      return (1);
    }
    // Process user responses.
    UI_APPLICATION::Control();
    // Clean up.
    delete UI_WINDOW_OBJECT::errorSystem;
    return (0);
  }
```

Class definitions

The dictionary window is implemented in a class called **DICTIONARY_WINDOW**. Here's its definition:

```
class EXPORT DICTIONARY_WINDOW : public UIW_WINDOW
{
public:
  DICTIONARY_WINDOW(char *dictionaryName);
  ~DICTIONARY_WINDOW(void);
  EVENT_TYPE Event(const UI_EVENT &event);
private:
  DICTIONARY *dictionary;
  UIW_STRING *inputField;
  UIW_TEXT *definitionField;
  UIW_STRING *antonymField;
  UIW_STRING *synonymField;
  static EVENT_TYPE ButtonFunction(UI_WINDOW_OBJECT *item,
    UI_EVENT &event, EVENT_TYPE ccode);
};
```

DICTIONARY_WINDOW contains the input, definition, antonym, and synonym fields, as well as the lookup, save, and delete buttons. This class uses private member variables, accessible only to itself. They are:

- *dictionary,* the pointer to the dictionary itself. The dictionary is created in the constructor for **DICTIONARY_WINDOW**.
- *inputField*, a pointer to the **UIW_STRING** field. Collects input from the user.
- *definitionField*, a pointer to the **UIW_TEXT** field. Displays the definition of the input word.
- *antonymField*, a pointer to the **UIW_STRING** field. Displays antonyms of the input word.
- *synonymField*, a pointer to the **UIW_STRING** field. Displays synonyms for the input word.

It also includes a static user function, **ButtonFunction( )**, that is called when the button is selected. It accepts the following parameters:

- *object*, an object of class **UI_WINDOW_OBJECT**,
- *event*, a structure of type **UI_EVENT**, and
- *ccode*, the event type.

The **D_ENTRY** is the entry in the data file that contains the data we enter in **DICTIONARY_WINDOW**'s fields. Here's the definition for the **D_ENTRY** class:

```
class D_ENTRY
{
public:
  int wasLoaded;
  char *word;
  char *definition;
  char *antonym;
  char *synonym;
  D_ENTRY(const char *name, ZIL_STORAGE *file,
    UIS_FLAGS sFlags = UIS_READ);
  ~D_ENTRY();
  static D_ENTRY *New(const char *name, ZIL_STORAGE *file,
    UIS_FLAGS sFlags = UIS_READ);
  void Save();
private:
  ZIL_STORAGE_OBJECT *object;
};
```

D_ENTRY uses the following member variables:

- *wasLoaded*, a flag that denotes whether or not the entry was loaded.
- *word*, a string that contains the entry in the dictionary.
- *definition*, a string that contains the definition string for the word.
- *antonym*, a list of antonyms that apply to the dictionary entry.
- *synonym*, a list of synonyms that apply to the dictionary entry.

DICTIONARY derives from **ZIL_STORAGE**, which contains methods for saving and loading data to and from a data file.

DICTIONARY has the following parameter:

- *name*, which is the name of the **.DAT** file being used as the dictionary data file.

Creating the user interface

Using the Designer to create the window

The first thing we'll do is use Zinc Designer to recreate the main window and save it in the file **WORD_WIN.DAT**. Follow these steps:

1. First, create a new file in the Designer. Select **File | New**, and then type **WORD_WIN.DAT** for the filename. Click the **OK** button to create the file.
2. Then create a new window by selecting **Window | Create**. Select the string object icon, located at the upper left of the Designer tool bar. Then drag and drop four string fields on the window.

   The Designer gives each string field a default string ID of the form **FIELD_1**, **FIELD_2**, and so forth. In order to access a particular field programmatically, we need to specify that string's ID. But the defaults don't help us remember which field is which, so let's change the string IDs to something we can remember.
3. To change each string ID, bring up one at a time the edit window of each string field by double-clicking on the background of the window. Select the notebook tab called **Sub-Objects**, which will bring up a vertical list of all the subobjects in the window. Find the ones marked **FIELD_1**, **FIELD_2**, and so forth. In the vertical list, double-click on the first one, and a new window will pop up that contains several fields for information related to that subobject. Enter the appropriate string ID in each **Name** field—use *DCT_INPUT* for the first one, then change the string IDs of the other fields to *DCT_DEFINITION*, *DCT_ANTONYM*, and *DCT_SYNONYM*.
4. Create some buttons and change the string IDs of our buttons. To change the **Lookup** button's stringID, double-click on the window's background, click on the **Sub-Objects** notebook tab, and select the first button in the list. Enter *DCT_LOOKUP_BUTTON* in the **Name** field. Likewise, change the **Save** button's string ID to *DCT_SAVE_BUTTON*, and the **Delete**'s button to *DCT_DELETE_BUTTON*.

Wiring up the interface

Now that we've set up the window, the next step is to "wire up" the interface so that we can get data in and out of the fields and cause each button to call the static user function. We do this in the implementation of **DICTIONARY_WINDOW** by setting up pointers to the string fields so we can access their contents programmatically, and by assigning each button the same static user function.

Here's how we wire up the interface.

1. First, we create a pointer to each string field. Then we call the window's **Information( )** function with the *I_GET_STRINGID_OBJECT* request that tells the **Information( )** function to return a pointer to the object whose stringID matches the stringID passed in the second parameter of the **Information( )** function call. We also use the string ID of each field so the **Information( )** function knows from which field to get the text.

```
// Set up the pointers to the window fields.
inputField = (UIW_STRING *)Information(I_GET_STRINGID_OBJECT,
  "DCT_INPUT");
definitionField = (UIW_TEXT *)Information(I_GET_STRINGID_OBJECT,
  "DCT_DEFINITION");
antonymField = (UIW_STRING *)Information(I_GET_STRINGID_OBJECT,
  "DCT_ANTONYM");
synonymField = (UIW_STRING *)Information(I_GET_STRINGID_OBJECT,
  "DCT_SYNONYM");
```

   The next thing is to connect the buttons to the static user function.

2. Create a pointer to a button.
3. Then call the window's **Get( )** function with the *numberID* assigned to the **<u>L</u>ookup** button, which is *DCT_LOOKUP_BUTTON*. The **Get( )** function will return a pointer to a **UIW_BUTTON** object.

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

The Event( ) function

The **Event( )** function is where all the action takes place in our tutorial. It traps the events generated when the user selects a button and performs the appropriate action.

*The **D_ENTRY** class*

The dictionary entry is an instance of the **D_ENTRY** class, which encapsulates the data in the dictionary and provides methods for creating a new entry and saving an existing entry to a file.

ZIL_STORAGE_ OBJECT

The **D_ENTRY** class contains a private member variable called *object*, of type **ZIL_STORAGE_OBJECT** that can be stored in the data file. We'll use it in conjunction with **DICTIONARY**, which derives from **ZIL_STORAGE**, to load and store data in the file.

Although **D_ENTRY** contains a **ZIL_STORAGE_OBJECT** member variable, we must set up two functions in order for it to access the data file. These functions are **New( )** and **Save( )**.

The constructor

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

name, the name of the storage object.

file, the file containing the object. If the object is not found in the file, the member *wasLoaded* is set to *FALSE.* Otherwise, *wasLoaded* is set to *TRUE* and the constructor retrieves the object from the data file.

flags, which indicates whether the object is to be loaded or created. If the program finds the entry, and if we set the *UIS_CREATE* flag, it will delete the existing entry so the program can save the new entry.

When the program finds an existing entry in the data file, it loads the word, its definition, its antonyms, and its synonyms.

The New function

When the program looks up a word in the dictionary and reads in the entry from the data file, it calls the function **D_ENTRY::New( )**, which creates a new object. (**New( )** is a static member function of **D_ENTRY**, not the **new** operator of C++.) The reason for having a static **New( )** function is so the function can return a value indicating if the object was created successfully or not.

The Save function

The purpose of the **Save( )** function is to save the object into a file. The following listing shows how the function stores the words:

```
void D_ENTRY::Save(void)
{
  // Store the field information.
```

```
        object->Store(word);
        object->Store(definition);
        object->Store(antonym);
        object->Store(synonym);
    }
```

When **Save( )** is called, *object*->**Store( )** writes the data to storage. **UI_-STORAGE** actually writes the data to a temporary file and not to the actual data file; that work is done in **UI_STORAGE::Save( )**, found in the destructor for the **DICTIONARY** class. The destructor is called after the user sends us the "quit" event, and the **Control( )** function returns control to us.

The ***DICTIONARY*** *class*

The dictionary class handles the tasks of saving and loading the data to and from the data file. To do so, **DICTIONARY** derives from **ZIL_STORAGE**, which reads and writes Zinc data files.

We can think of **ZIL_STORAGE**, and therefore **DICTIONARY** as well, as a file system that can change directories, make new directories, and add and delete resources. The main difference between a **ZIL_STORAGE** class and a regular file system is that **ZIL_STORAGE** lets us save and retrieve persistent objects as well as items or objects of different types.

DICTIONARY doesn't actually save the data file when we press the **Save** button; as we learned previously, instead it caches it in a temporary file until the program falls through the **Control( )** function. Then the destructor saves the data file using the **Save( )** function it inherited from **ZIL_STORAGE**. Using the **Save( )** function is easy. We gave the function the name of the file to save in the constructor when we loaded the **.DAT** file; therefore we only need to call the function with the parameter *1*. This tells the function to save the data file.

Conclusion

In this chapter, we learned how to use the Zinc data file, and how to add objects to it. This chapter also gave us some more practice on how to use windows created in the Designer, and how to connect code to an interface. In the next chapter, we'll learn how to extend an existing Zinc object with new functionality.

Chapter 14

Virtual List

Displaying records from a database is a common programming task, often complicated by the fact that the database may have many more records than can fit in memory at once. So, to display many records a virtual list is needed. A virtual list does not attempt to load all the records at once. Instead, it only loads those that are visible at any given time. In this chapter we will learn how to use a Zinc object, **UIW_TABLE**, to create a virtual list.

Key Concepts

creating a virtual list

using the **UIW_TABLE** class

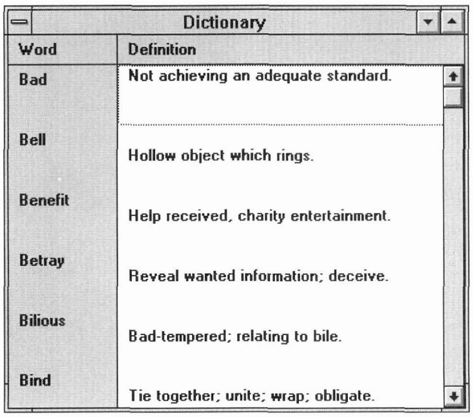
What we'll do

Here are the steps we'll take in writing **VLIST.CPP**.

1. Create a **UIW_TABLE**. The **UIW_TABLE** class has built in virtual capability, so no new functionality is required on our part.
2. Create the **UIW_TABLE_HEADER**s that are used to label the columns and the rows.
3. Create the **UIW_TABLE_RECORD**s that are used to display the information in the headers and in the table. Add all fields to the table records.
4. Create the user functions that the table records will call when they need to update their data.

Running the program

Compile the source code and run the executable. You should see the following window on the screen:



Word	Definition
Bad	Not achieving an adequate standard.
Bell	Hollow object which rings.
Benefit	Help received, charity entertainment.
Betray	Reveal wanted information; deceive.
Bilious	Bad-tempered; relating to bile.
Bind	Tie together; unite; wrap; obligate.

The table is a nonfield region so it occupies the entire window. The table has three headers: a column header that contains a label identifying the definition column; a corner header that contains a label identifying the word column; and the row header, which contains the words. The definitions appear as records in the table. Each definition record contains a multi-line text object.

All movement is handled by the table. We can scroll the table up and down using either the scroll bar or the keyboard. Table keystrokes are native to each environment, but typically are the equivalent of <Ctrl+Up Arrow> and <Ctrl+Down Arrow>.

The application retrieves the necessary data from disk whenever a new record scrolls into view.

All fields in this tutorial are view only, so you won't be able to edit any information.

Source code

- **VLIST.CPP**. Contains all the source code for the virtual list. This includes the following functions..

```
LoadRecord()
RecordFunction()
RowHeaderFunction()
UI_APPLICATION::Main()
```

- **VLIST.TXT**. Contains 100 records that are dynamically read from disk when needed by the virtual list.
- ***.DEF**, ***.RC**. The environment specific definition and resource files required when compiling for environments Zinc supports.
- ***.MAK**. The compiler-dependent makefiles used to build **VLIST.CPP**.

Analyzing the source code

The first section in the source file, **VLIST.CPP**, contains some pre-compiler variable definitions and some global variable declarations. *RECORD_LENGTH* is the length of each record in the data file. In our application we are using fixed-length records. *RECORD_LENGTH* is different across environments due to how each environment handles the end-of-line character.

file is the file handle of the data file, **VLIST.TXT**.

maxRecords is the number of records in the data file.

The next section of the source file contains the definitions of the support functions used in our application. The **LoadRecord( )** function loads a record from the data file. It takes three parameters. The first parameter is the record number to load. The second parameter is a text buffer where the function is to place the word. The third parameter is a text buffer where the function is to place the definition.

RecordFunction( ) is a user function associated with the **UIW_TABLE_RECORD** used to display the definitions. This function is called by the **UIW_TABLE_RECORD** just as any user function is, when the object becomes current, is selected, or becomes noncurrent. In addition to being called at these times, however, a table record user function is also called when the record needs to load its data. We will discuss this in further detail below, where we discuss how the table operates.

RowHeaderFunction( ) is a user function associated with the **UIW_TABLE_RECORD** used to display the words.

The last section in the source file contains the **UI_APPLICATION-::Main( )** function definition:

```
int UI_APPLICATION::Main()
{
  UI_APPLICATION::LinkMain();
  maxRecords = 100;
  file = fopen("vlist.txt", "rb");
  UIW_WINDOW *window = UIW_WINDOW::Generic(3, 2, 53, 13,
    "Dictionary");
  UI_WINDOW_OBJECT *rowPrompt, *definition;
  UIW_TABLE *table = new UIW_TABLE(1, 1, 40, 10, 1, 0, 100,
    ZIL_NULLP(void), 100, TBLF_NO_FLAGS, WOF_NON_FIELD_REGION |
    WOF_NO_ALLOCATE_DATA);
  UIW_TABLE_HEADER *cornerHeader = new
    UIW_TABLE_HEADER(THF_CORNER_HEADER);
  UIW_TABLE_HEADER *colHeader = new
    UIW_TABLE_HEADER(THF_COLUMN_HEADER);
  UIW_TABLE_HEADER *rowHeader = new
    UIW_TABLE_HEADER(THF_ROW_HEADER);
  *cornerHeader
    + &(*new UIW_TABLE_RECORD(8, 1)
      + new UIW_PROMPT(1, 0, "Word"));
  *colHeader
    + &(*new UIW_TABLE_RECORD(40, 1)
      + new UIW_PROMPT(1, 0, "Definition"));
  *rowHeader
    + &(*new UIW_TABLE_RECORD(12, 2, RowHeaderFunc)
      + (rowPrompt = new UIW_PROMPT(1, 0, "")));
  *table
    + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_CORNER)
    + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_VERTICAL)
    + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_HORIZONTAL)
    + cornerHeader
    + colHeader
    + rowHeader
    + &(*new UIW_TABLE_RECORD(37, 2, RecordFunction)
      + (definition = new UIW_TEXT(1, 0, 35, 2, "", 80,
        WNF_NO_FLAGS, WOF_VIEW_ONLY)));
  rowPrompt->StringID("ROW_PROMPT");
  definition->StringID("DEFINITION");
  *window
```

```
      + table;
    *windowManager
      + window;
    // Process user responses.
    UI_APPLICATION::Control();
    fclose(file);
    return (0);
  }
```

This is where we set up the application by opening the data file, creating the window, the table, and all the subobjects of the table, and processing the user events.

Program flow

When the application starts, it creates a window, places a table on the window, and adds the window to the Window Manager. As the table is displaying a record for the first time—for example, when the table is first coming up or as a new record is scrolled into view—the table record's user function is called to load the data. All events are handled by the table and its subobjects.

*Using the **UIW_TABLE** object*

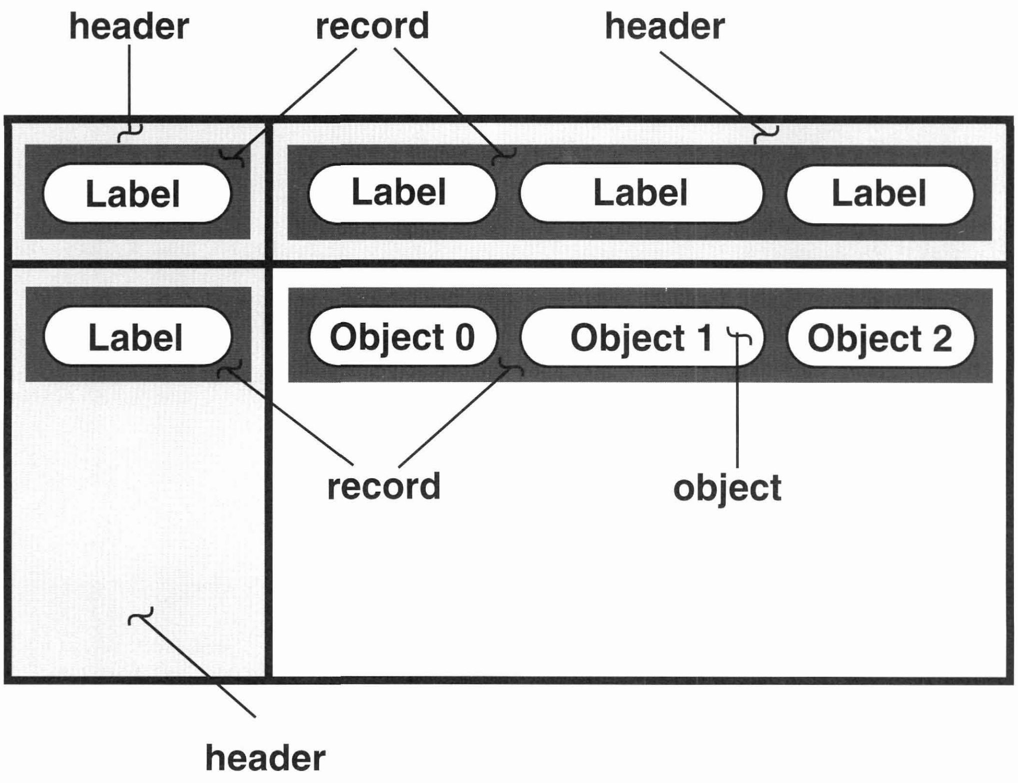
In keeping with the philosophy of Zinc, the **UIW_TABLE** object offers us a good deal of flexibility—a record can consist of a single field, as it does in this application, or it can be made up of many different fields. The table can have a single column, or it can be made up of dozens of columns, as a spreadsheet might be. The table can handle memory allocation for you, or you can take care of it yourself.

Along with all this flexibility, however, comes a certain amount of complexity. So we're going to devote the rest of this chapter to a discussion of the basics of using the **UIW_TABLE**.

Table structure

When we break it down, we find that a table consists of records of data and some labels identifying each field in the data records. The **UIW_TABLE_RECORD** class displays records, and the **UIW_TABLE_-**

HEADER class displays the column and row labels. All data manipulation is handled at the table record level. And lastly, standard Zinc window objects comprise data and label fields. Here's a representation of a table object:



The table record

A table record, similar to a window, is simply a collection of fields that are in some way related. In fact, the **UIW_TABLE_RECORD** class derives from **UIW_WINDOW**, and we create and add fields to the table record just as would add window objects to a window. When creating a table record we specify its height and width and associate a user function with it. We will talk about the user function later when we discuss how we get data into a record.

The table header

The table header is like a small table that appears in a special area of the table. Instead of being used to input and output data, though, the header only displays information, usually describing the contents of the column or row with which it is associated. The header appears down the left edge of the table, in the upper-left corner of the table, or across the top of the table, depending on the table header's flag setting. Often, several fields are needed in a header, typically because the data being described by the header consists of several fields. For this reason, we add each label field to a table record and add the table record to the header. The table header is, in turn, added to the table. Our application only uses one field in the header, but we can see this hierarchy of additions in the code:

```
UIW_TABLE_HEADER *rowHeader = new
  UIW_TABLE_HEADER(THF_ROW_HEADER);
...
*rowHeader
    + &(*new UIW_TABLE_RECORD(12, 2, RowHeaderFunc)
      + (rowPrompt = new UIW_PROMPT(1, 0, "")));
...
*table
    ...
    + rowHeader
```

Adding records to the list

You may have noticed that only one **UIW_TABLE_RECORD** was added to the table and to each of the table headers.

```
*table
    ...
    + &(*new UIW_TABLE_RECORD(37, 2, RecordFunction)
      + (definition = new UIW_TEXT(1, 0, 35, 2, "", 80,
        WNF_NO_FLAGS, WOF_VIEW_ONLY)));
```

If the dictionary we displayed in our application has 100 records, and if there were typically 5 or more records displayed at any given time, how did the one record become 100? The answer lies in one of the most useful features of the table object, its built-in virtual capability. We only add one record, but the table makes it look as if there are many records. The details of how it does this are not relevant to our discussion, but in a nutshell it makes a copy of the record we add and then uses that copy to draw images of all the records except the current one.

Adding fields to the records

Each field of data, whether it is a label on a header or a part of a data record, is created using a window object. If we place the object in a header, using a **UIW_PROMPT** is usually sufficient, since this data can never be edited. The fields in a data record, however, will often both display information and collect information from the user. These fields can be just about any window object.

To set the fields in a record, simply create them and add them to the table record just as you would add them to a window. Their size and position parameters are used to place the object within the region of the table record, and their other flag settings will affect their operation and appearance. Let's look at our definition record:

```
+ &(*new UIW_TABLE_RECORD(37, 2, RecordFunction)
    + (definition = new UIW_TEXT(1, 0, 35, 2, "", 80,
      WNF_NO_FLAGS, WOF_VIEW_ONLY)));
```

The definition record only contains a **UIW_TEXT** object. We can see from the parameters that it is placed one cell from the left of the table record, is 35 cells wide and two cells tall. It has a maximum length of 80 characters, is view only, and has no border.

If we look at the header used to label the definition record we will see how the two are related:

```
*colHeader
    + &(*new UIW_TABLE_RECORD(37, 1)
      + new UIW_PROMPT(1, 0, "Definition"));
```

The label is created using a **UIW_PROMPT** that is placed one cell from the left of the table record, so it aligns with the text of the definition field.

Getting the data into the fields

So, if most of the data we see is actually only an image of the fields, and if we only add one table record to the table or header, how does the data get there?

There are several ways to place data into the table. One way is to pass the data in to the **UIW_TABLE** constructor. This, of course, won't work if there is more data than can fit in memory at one time. This also only provides data for the data in the table, but not for the headers. We wanted all of our data to come from the data file so we didn't give the table any memory and we set its *WOF_NO_ALLOCATE_DATA* flag so that it would not attempt to allocate memory for our data. We can see this in the call to the **UIW_TABLE** constructor:

```
UIW_TABLE *table = new UIW_TABLE(1, 1, 40, 10, 1, 0, 100,
  ZIL_NULLP(void), 100, TBLF_NO_FLAGS, WOF_NON_FIELD_REGION |
  WOF_NO_ALLOCATE_DATA);
```

If we wanted to initialize some data at the beginning, we could have passed in a data block—for example, an array of structures, each containing data for a single record—and indicated how many records of data that block contained.

Another way to get data into the records is by using user functions with the table records. This is the method we used in **VLIST**. Whenever a table record needs to have its data set, it calls the user function. Let's look at the definition field's user function, **RecordFunction( )**:

```
EVENT_TYPE RecordFunction(UI_WINDOW_OBJECT *object,
  UI_EVENT &event, EVENT_TYPE ccode)
{
  if (ccode == S_SET_DATA)
  {
    ZIL_ICHAR definition[80];
    LoadRecord(event.rawCode, ZIL_NULLP(ZIL_ICHAR), definition);
    object->Get("DEFINITION")->Information(I_SET_TEXT,
      definition);
  }
  return (ccode);
}
```

As we mentioned earlier, in addition to the usual times that a user function is called, a user function associated with a table record is called when the table record needs its data set. In our user function we check to see if the *ccode* is *S_SET_DATA*, the message we'll get when we need to set the record's data. If it is, we call **LoadRecord( )** to load the record from disk. The record number is passed in *event.rawCode*. If the table record had any memory allocated for its data—**VLIST** does not, since we neither passed any to the table constructor, nor set the *WOF_NO_ALLOCATE_DATA* flag for the table—the pointer to this data would be passed in *event.data*. After we get the definition back from **LoadRecord( )** we get a pointer to the text object in the table record that displays the definition and set its data with our definition. And the table takes care of the rest. If we wanted to, we could use this user function to save data whenever the object was becoming noncurrent or perform some other action if the object is selected.

A third way of updating a record's data is similar to using the user function. Instead of the user function, however, we could derive our own table record class and trap the *S_SET_DATA* event in its **Event( )** function.

Conclusion

Now that we've learned how to write a virtual list and how to use event map tables, we'll learn about deriving our own custom device classes. This will give us the ability to write programs that respond to user input in ways we can define.

Chapter 15

Deriving a Device

In this chapter, we'll learn how to derive our own device. We'll create a macro device that will watch the events flowing through the system to see if the user presses certain macro keys. If the user does press a macro key, the device will enter some text into a text object.

Key Concepts

how to work with input devices

how to write a simple keyboard macro

how to initialize the macro device class and its base class

What we'll do

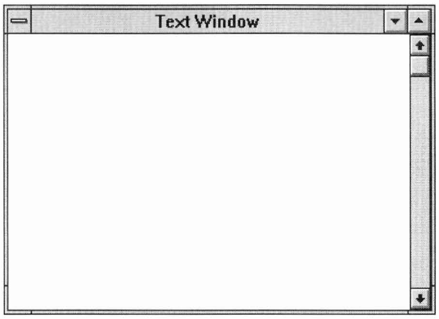
Source code

The source code for this program is located in the **\ZINC\TUTOR\MACRO** subdirectory, and 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 inside **UI_APPLICATION::Main( )**.
- ***.DEF**, ***.RC**. The environment specific definition and resource files.
- ***.MAK.** The compiler-dependent makefiles. See "Appendix A—Compiler Considerations" for information on compiling for each Zinc-supported platform.

Program execution

Let's begin by looking at how the keyboard macro works. 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, which, in this case, is a nonfield region that takes up the entire region within the window. In addition to a text object, this program has four macro keys.

TABLE 12. **Macro keys and their function**

Keys	*Function*
<F5>	Enters the text "Macro #1." into the text window.
<F6>	Enters the text "Macro #2." into the text window.
<F7>	Enters the text "Macro #3." into the text window.
<F8>	Enters the text "Macro #4." into the text window.

Class definitions

The macro device is implemented in a class called **MACRO_HANDLER**. Here's its definition:

```
const EVENT_TYPE E_MACRO = 89;
struct MACRO_PAIR
{
  RAW_CODE rawCode;
  char *macro;
};
class MACRO_HANDLER : public UI_DEVICE
{
public:
  MACRO_HANDLER(MACRO_PAIR *_macroTable);
  EVENT_TYPE Event(const UI_EVENT &event);
private:
  MACRO_PAIR *macroTable;
  MACRO_PAIR *currentMacro;
  int offset;
  void Poll(void);
};
```

MACRO_HANDLER uses the following definitions and member variables:

- *E_MACRO*, a constant value that uniquely identifies the macro device. Zinc predefines the values for the keyboard, mouse, and cursor devices, but leaves other values open for input devices that we design ourselves. We'll discuss later in this chapter the significance of the value 89.
- *MACRO_PAIR,* a structure that allows us to define a keyboard/macro equivalent pair. Below is the definition of the four macro keys we will use in our sample program:

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

The entry { 0, NULL } is an end-of-array indicator. In addition, 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 scan codes defined in **UI_MAP.HPP**.

- *macroTable,* a pointer to the table that contains the *rawCode/macro* pairs to be matched.
- *currentMacro*, a pointer to the current, or active, macro. This value is reset whenever a new macro key is pressed.
- *offset*, a value that gives the position within the *currentMacro->macro* character array. We use this when the macro device places a keyboard event into the Event Manager's event queue.

Program flow

The code sample and the corresponding steps show how the macro device works after we attach it to the Event Manager.

1. When the programmer calls *eventManager*->**Get( )**, it calls the device's **Poll( )** function. The first thing the **Poll( )** function does is get the next event waiting to be processed from the event queue so it can determine if it is a macro key. The code for this step is shown below.

```
void MACRO_HANDLER::Poll(void)
{
  // See if any events are in the event manager's event queue.
  UI_EVENT event;
  static int emptyQueue = TRUE;
      if (emptyQueue)
        emptyQueue = eventManager->Get(event,
           Q_NO_POLL | Q_NO_BLOCK | Q_NO_DESTROY | Q_BEGIN);
```

   When calling *eventManager*->**Get( )**, we need to ensure we don't disrupt normal event handling; we do this by calling **Get( )** with four parameters, *Q_NO_POLL*, *Q_NO_BLOCK*, *Q_NO_DESTROY*, and *Q_BEGIN*.

   The *Q_NO_POLL* flag prevents the Event Manager from polling any other input devices. Since we are receiving user input while in a function of an input device, we must be careful to not poll input devices, causing unwanted recursion.

   The *Q_NO_BLOCK* flag protects against stopping program execution until an event is detected. We set this since we only want to check the event queue to see if an event is available. If there is an event in the queue, the function returns a value of 0. Otherwise, it returns a negative value.

The *Q_NO_DESTROY* flag prevents the **Get( )** function from destroying the contents of the queue merely by looking for special keyboard events. This flag allows us to examine the events without removing them from the queue.

Q_BEGIN lets our function get events from the beginning, rather than the end, of the queue.

2. The second step is to check for events specific to a particular environment. If our program receives these types of events, they are mapped to the generic Zinc event format for processing. Here's an example of how our program maps events for some operating systems.

```
// Check for environment-specific keyboard events.
  #if defined (ZIL_MSWINDOWS)
    if (state == D_OFF && !emptyQueue && event.type == E_MSWINDOWS &&
      event.message.message == WM_KEYDOWN)
    {
      ...
  #elif defined (ZIL_OS2)
    if (!emptyQueue && event.type == E_OS2 &&
      event.message.msg == WM_CHAR)
    {
      ...
  #elif defined (ZIL_MOTIF)
    if (!emptyQueue && event.type == E_MOTIF &&
      event.message.type == KeyPress)
    {
      ...
  #endif
      ...
    }
```

3. This step determines if a macro key was pressed, and if so, which one. The program only executes this step if the device is not already processing a macro key. If the user has pressed a valid macro key, the program shuts off all other input devices, so they won't feed more information into the queue while we are putting into the queue our macro events.

   Next, the original macro key is removed from the Event Manager's event queue and the macro device is enabled.

4. The program only executes the fourth step if the macro device is enabled. Once the macro device is enabled, it feeds one event into the event 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 fourth step from being executed until another macro key is pressed.

```
// Put macro information into the event queue.
if (state == D_ON && emptyQueue)
{
  ...
}
```

5. The main program loop processes all event information, including the macro key expansions, by calling *windowManager*->**Event( )**. The main program loop exits if the *L_EXIT* message is received, or it returns to the first step to get the next event.

Base class initialization

The **MACRO_HANDLER** class constructor is an inline function.

```
class MACRO_HANDLER : public UI_DEVICE
{
public:
  MACRO_HANDLER(MACRO_PAIR *_macroTable) : UI_DEVICE(E_MACRO, D_OFF),
    macroTable(_macroTable) { installed = TRUE; }
```

We call **UI_DEVICE's** class constructor before any we set any class-specific information. 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 list. Input devices are arranged in the device list in ascending type order. The order of the four input devices we attached to the Event Manager is:

- **UID_KEYBOARD**. Its value is 10, the number associated with the constant variable *E_KEY.*
- **UID_MOUSE**. Its value is 30, the number associated with the constant variable *E_MOUSE.*
- **UID_CURSOR**. Its value is 50, the number associated with the constant variable *E_CURSOR.*
- **MACRO_HANDLER**. We assigned it the value 89, so that it would be the last device in the list.

Here's why the macro handler should be the last device in the list. Its **Poll( )** function must review any activity 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 event 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 event queue before the keyboard feeds its information and will never see the <F5> key, and it will be passed to the main control before the macro handler is called again.

The initial state of the macro device needs to be off so that the program doesn't think macro information is being fed into the event 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 either on or off.

1. *D_OFF.* When the macro device is not placing events into the event queue, it sets itself to this state.
2. *D_ON.* When the macro device places events into the event queue, it sets itself to this state.

The Event Manager and **UI_DEVICE** set three other variables:

enabled, a second-level state indicator. **UI_DEVICE** sets this variable to be *TRUE*, but the macro device ignores it.

display, a pointer to the screen display created in the main event loop. Not set until the macro device is attached to the Event Manager. The macro device does not use *display*.

eventManager, a pointer to the Event Manager where the macro device is attached. The macro device uses this pointer to make queries on and place events in the event queue.

Initializing member variables

The class member *macroTable* is initialized to point to the constructor argument *_macroTable*. This variable is the search table for keyboard/macro expansions. The array specified in this argument must not be destroyed until the class is destroyed by the Event Manager.

The last thing the class constructor does is override the base class member *installed*. The value specified is *TRUE*. This value is not used by the Event Manager, but it does provide consistency when checking for device installation.

The class members *currentMacro* and *offset* are not set until the state of the device changes to *D_ON*.

The Poll function

We mentioned **MACRO_HANDLER::Poll( )** function earlier in this chapter. **Poll( )** functions do the following:

1. Feed information to or get information from the Event Manager's event queue. The keyboard, mouse, cursor, and timer devices all have poll routines that feed information into the event queue.
2. Pass control to an object periodically. Some environments Zinc supports don't multitask, and so using a poll routine in those environments ensures the program will poll all devices each time it calls the *eventManager->* **Get( )** function. The cursor device uses a poll routine 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 at regular intervals.

q&a

"How do I install hotkeys?"

Any prompt or selectable object such as a button or menu item can have a hot key. If the object is attached to a window added to the window manager, all you need to do is place the '&' before the desired hotkey in the object's text. If you want the object to respond to special characters, such as '#,' you may need to copy the library's *hotKeyMapTable* and add entries for the special characters. The library's *hotKeyMapTable* is defined in **G_WIN.CPP**.

If you want to place the hot key object in a group, list, or child window, pass *HOT_KEY_SUB_WINDOW* to the parent object's **HotKey( )** function. This tells the parent window to search its subobjects for a match on the hot key. For more details, see **UI_WINDOW_OBJECT::HotKey( )** in the *Programmer's Reference, Volume 1.*

The macro device feeds information to and gets information from the Event Manager. When the device is on, it feeds information into the event queue and checks the input when it is off.

Responding to events

The **MACRO_HANDLER::Event( )** function is defined below:

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

This routine must be declared by the macro device since the base **UI_DEVICE** declares it a virtual function.

```
class UI_DEVICE : public UI_ELEMENT
{
  public:
  virtual EVENT_TYPE Event(const UI_EVENT &event) = 0;
```

Generally, we use **Event( )** functions to change the state of an input device.

Enhancements

Now that we have discussed the design and implementation of a macro device, let's look at some variations we could implement to make the class more powerful.

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 event queue in one step. The benefits of this method are that it simplifies the process of the macro device and that it prevents the need for disabling all other input devices. The problem with this implementation is twofold.

   First, the macro may fill the input buffer, in which case we will have to write code to wait until the buffer is not full. Second, the macro may itself contain a character that is a macro key. This would require modification to our member variables and may cause recursion of macro events.
2. Modify the static variables **UIW_STRING::***pasteBuffer* and **UIW-_STRING::***pasteLength* to contain the macro, then send an *L_PASTE* message through the system. This implementation's only drawbacks would be wiping out the old information in the global paste buffer and that the receiving object may not be a simple text field, like the window created in our application.
3. Extend the macro device to enable the addition or deletion of macro pairs. This could be accomplished by overloading the **+** and – operators for the **MACRO_HANDLER** class.
4. Extend the macro pair to handle logical, system, or normal keyboard information. In this implementation, we would modify the definition of *MACRO_PAIR.macro* to support **UI_EVENT** information, rather than

simple character values. In addition, we would probably want to write an editor so that the macro could be edited and modified easily. This would require setting 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 us add to, delete from, or modify the contents of the list.

Conclusion

Now that we've learned how the keyboard macro device works, we'll learn to derive our own custom display classes. This will give us the ability to write displays built around third-party graphics libraries and will teach us more about the display class.

Chapter 16

Customized Displays

In order to display information on the screen under each of the operating environments Zinc supports, we use a display object to handle drawing chores. Writing display classes from scratch would consume a great amount of time, so Zinc designed **UI_DISPLAY**, which is an abstract class that describes basic behaviors of drawing but leaves the implementation up to us. Here we will use **UI_DISPLAY** to derive a display class for a specific graphics library, **UI_BGI_DISPLAY**.

Key Concepts

- the basics of designing of a display class
- initializing the display class and its base class
- giving a display class custom behavior

What we'll do

All display classes derive from **UI_DISPLAY**, which handles the details of the display. But **UI_DISPLAY** doesn't automatically know what those details are; we need to define those behaviors in our derived display, **UI_BGI_DISPLAY**. To tell **UI_DISPLAY** about those details, we must take three steps.

1. Decide which virtual functions contained in **UI_DISPLAY** we'll implement in our derived display, **UI_BGI_DISPLAY**.
2. Determine the coordinate system. This depends on whether the display is running in text or graphics modes. The coordinate system is always left-top, zero-based, where {0,0} is the coordinate of the left-top corner of the screen, and where the type of display and the mode in which it is running determines the right-bottom coordinates.
3. Define clip regions, or identifying rectangular regions of the screen where windows overlap. For example, if two windows were attached to the screen, the display would contain several rectangular regions with different identifications. Most environments handle drawing routines as well as clipping.

Using the class

A display class defines some methods for drawing on the screen. We begin defining those methods by deciding basic properties of the screen like the types of fonts, the number of columns and lines, and whether the display is color or monochrome. Then we declare the behaviors we want our display class to use, behaviors like starting up the display, and others like drawing lines, polygons, or rectangles.

Source code

The source code for this example is located in **\ZINC\TUTOR\DISPLAY**, and contains:

- **TEST.CPP**, a test program.
- **BORLAND.MAK**, the makefile associated with the test program.
- **D_BGIDSP.CPP,** located in **\ZINC\SOURCE**, contains the BGI class constructor, destructor, and associated display member functions.

To derive the **UI_BGI_DISPLAY** class, we need the graphics display library **GRAPHICS.LIB,** and its BGI files, **EGAVGA.BGI**, **CGA.BGI**, and **HERC.BGI**, provided with the Borland compiler. Even if we're not using the Borland compiler, or even if we won't derive our own display later on, we can still learn the design and implementation of display classes by studying this chapter.

Besides setting up information needed for working with screens, **UI_DISPLAY** initializes the following member variables:

- *installed*, which tells whether the display has been installed. By default, **UI_DISPLAY** sets it to *FALSE*. We need to tell our derived display constructor to set this variable to be *TRUE* if the graphics display installs correctly.
- *isMono,* which tells whether the display is operating in monochrome mode.
- *cellWidth* and *cellHeight,* the width and height values of a cell coordinate. If the program is running in text mode, *cellWidth* and *cellHeight* are 1. Otherwise, the values of *cellWidth* and *cell Height* are determined by the graphics mode and default font size. For example, the **UI_BGI_DISPLAY** class constructor sets *cellWidth* to 7 and *cellHeight* to 23.
- *columns* and *lines*, the columns or lines the display contains. The following table shows BGI's values for *columns* and *lines*:

TABLE 13. BGI display values

Display	*Columns*	*Lines*
Text	80	25
	40	25
	80	43
	80	50
CGA	320	200
MCGA	320	200
EGA	350	480
VGA	640	480

- *preSpace* denotes the size in pixels of the white space between the top border of a string field and the tallest character. By default, *preSpace* is set to 2.
- *postSpace* denotes the size in pixels of the white space between the bottom border of a string field and the lowest character. By default, *postSpace* is set to 2.

- *miniNumeratorX* and *miniDenominatorX* determine the width of a mini-cell. *miniNumeratorX* is set to 1 and *miniDenominatorX* is set to 10. These values default to 1/10th of a cellwidth. Mini-cells provide for more precise positioning of objects and are available in graphics modes only.
- *miniNumeratorY* and *miniDenominatorY* determine the height of a mini-cell. *miniNumeratorY* is set to 1 and *miniDenominatorY* is set to 10. These values default to 1/10th of a cellheight. Mini-cells provide for more precise positioning of objects and are available in graphics modes only.
- b*ackgroundPalette* is a pointer to the background color palette. When initialized, this static pointer points to the **UI_PALETTE** structure, *_backgroundPalette*, contained in **G_DSP.CPP**.
- *xorPalette* is a pointer to the XOR color palette. When initialized, this static pointer points to the **UI_PALETTE** structure, *__xorPalette*, contained in **G_DSP.CPP**.
- *colorMap* is a pointer to the normal color palette. When initialized, this static pointer points to the **UI_PALETTE** structure, *__colorMap*, contained in **G_DSP.CPP**.

*Writing **UI_BGI_DISPLAY***

Initializing the base class

Since our derived display uses **UI_DISPLAY's** methods, we must first initialize **UI_DISPLAY** before we initialize **UI_BGI_DISPLAY**. To initialize it, we call inside of **UI_BGI_DISPLAY** the **UI_DISPLAY** constructor with three arguments, *isText*, *_operatingSystem*, and *_windowingSystem*.

```
UI_DISPLAY(FALSE, _operatingSystem, _windowingSystem)
```

When we call this function, **UI_DISPLAY** sets up then information needed for working with screens. To be able to write display classes, we need not understand what **UI_DISPLAY** does—we can treat **UI_DISPLAY** as a black box.

This black box notion is a benefit of Zinc and of object orientation in general. It allows us to use the functionality of another class *without having to understand how it works*. All we need to know is how to pass parameters and arguments into the class and let it do our work for us. If not for Zinc's true object orientation, writing our own display class would mean duplicating much of the work Zinc has already done.

Here's where we pass parameters and arguments into **UI_DISPLAY**'s constructor. *isText* is the first variable in the constructor, which tells whether a text display will be created—since we're creating a graphics display, this value is *FALSE*. We'll already know the values for the *_operatingSystem* and *_windowingSystem* variables before we write the class.

Initializing UI_BGI_DISPLAY

After initializing **UI_DISPLAY** to use its methods for working with screens, we have to initialize **UI_BGI_DISPLAY's** member variables. Below are the steps **UI_BGI_DISPLAY**'s constructor follows to initialize them.

1. Register the system, dialog, and small fonts contained in the **.CHR** files in **\ZINC\SOURCE**. We can modify these fonts with the Borland font editor, and we must compile them with the Borland utility **BGI2OBJ.EXE**, which translates them to **.OBJ** files. Once translated, the fonts are linked automatically into the program.

```
// Register the system, dialog and small fonts linked in.
  BGIFONT BGIFont = {0, 0, 1, 1, 1, 1, 0, 0 };
  BGIFont.font = registerfarbgifont(SmallFont);
  if (BGIFont.font >= 0)
  {
    BGIFont.charSize = 0;
    BGIFont.maxWidth = 10;
    BGIFont.maxHeight = 11;
    UI_BGI_DISPLAY::fontTable[FNT_SMALL_FONT] = BGIFont;
  }
  BGIFont.font = registerfarbgifont(DialogFont);
  if (BGIFont.font >= 0)
  {
    BGIFont.charSize = 0;
    BGIFont.maxWidth = 11;
    BGIFont.maxHeight = 11;
    UI_BGI_DISPLAY::fontTable[FNT_DIALOG_FONT] = BGIFont;
  }
  BGIFont.font = registerfarbgifont(SystemFont);
  if (BGIFont.font >= 0)
  {
    BGIFont.charSize = 0;
    BGIFont.maxWidth = 11;
    BGIFont.maxHeight = 13;
    UI_BGI_DISPLAY::fontTable[FNT_SYSTEM_FONT] = BGIFont;
  }
```

2. Determine the type of display. In the Borland graphics library we can determine the type of display by calling **detectgraph( )**. The *driver* and *mode* arguments of the constructor allow us to override this default detection.

```
// Find the type of display and initialize the driver.
    if (driver == DETECT)
      detectgraph(&driver, &mode);
    int tDriver, tMode;
```

3. Find the display's graphics driver. The current working directory is the first place we look, and the second is the originating directory of the program. If these fail, we use the **UI_PATH** object to search the directories specified by the environment variable *PATH*. If the driver cannot be found, the *installed* flag remains *FALSE,* and we drop out of the initialization process.

```
// Use temporary path if not installed in main().
int pathInstalled = searchPath ? TRUE : FALSE;
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;

}
```

4. Set up *columns*, *lines,* and *maxColors* variables that we discussed earlier.

```
columns = getmaxx() + 1;
lines = getmaxy() + 1;
maxColors = getmaxcolor() + 1;
```

5. Set up the default font, initialize *cellWidth* and *cellHeight,* fill the background screen, and define the new display region, which is, in our case, the entire screen. Since the display was successfully installed, the constructor sets *installed* to *TRUE.*

```
    // 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;
    }
```

Display destructor

The class destructor for **UI_BGI_DISPLAY** only has to do a small amount of work—it need only restore the display by calling **closegraph( )**, which restores the screen.

```
UI_BGI_DISPLAY::~UI_BGI_DISPLAY(void)
{
  // Restore the display.
  if (installed)
    closegraph();
}
```

The ***Rectangle( )*** *function*

Drawing on the screen

To show how to draw on the screen, let's examine the **UI_BGI_DISPLAY::Rectangle( )** function. All drawing functions, **Rectangle( )** included, work similarly—first we set up a draw region, then we draw inside of it. Here are the steps this function, or a rectangle function for any other display class, will take.

1. Set up the desired draw region. In our **Rectangle( )** function, we've specified two regions. The first region is where we draw the rectangle, otherwise called the fill region. We define this region with four coordinates: *left*, *top*, *right,* and *bottom*. The second region is specified by *clipRegion*, which describes where the drawing should be clipped. The clip region associates the screen identifications of window objects with a window. A window may contain several different window objects, such as buttons, title bar, and borders, but all the objects share the same identification, which ensures that one window object does not draw over another.

   The way we ensure that window objects don't draw over one another is to specify a *clipRegion* that is the *true* coordinates of the object that wants to draw to the screen. The object's true screen coordinates are contained in the public **UI_WINDOW_OBJECT::***true*.

```
// Assign the rectangle to the region structure.
UI_REGION region, tRegion;
if(!RegionInitialize(region, clipRegion, left, top, right, bottom))
  return;

// Draw the rectangle on the display
int changedScreen = FALSE;
```

2. Identify. Determine which areas of the screen have the same identification as that passed down by the *screenID* argument. To do this, our program walks through the list of region elements and checks their identifications with *screenID*'s. If the IDs match, and if the screen region and the region specified overlap, the program executes the third step.

```
    for (UI_REGION_ELEMENT *dRegion = First();
      dRegion;
      dRegion = dRegion->Next())
      if (screenID == ID_DIRECT ||
        (screenID == dRegion->screenID &&
        dRegion->region.Overlap(region, tRegion)))
      {
        if (screenID == ID_DIRECT)
          tRegion = region;
        if (!changedScreen)
        {
          changedScreen = VirtualGet(screenID, region.left,
            region.top, region.right, region.bottom);
          SetPattern(palette, xor);
}
```

3. Clip. The best way would be to set up all the clip regions at once and then draw the image. Unfortunately, the BGI graphics library does not support multiple clip regions, and so we must walk through the list of regions and

display the image each time we find an overlapping region. Note that for operating systems that associate a handle with a window object, *screenID* is set to the window handle.

```
setviewport(tRegion.left, tRegion.top, tRegion.right,
  tRegion.bottom, TRUE);
        if (fill && xor)// Patch for Borland bar() xor bug.
        {
          for (int i = 0; i < tRegion.right - tRegion.left; i++)
            line(i, top - tRegion.top, i, bottom - tRegion.top);
        }
        else if (fill)
          bar(left - tRegion.left, top - tRegion.top,
            right - tRegion.left, bottom - tRegion.top);
        for (int i = 0; i < width; i++)
          rectangle(left - (tRegion.left - i), top -
            (tRegion.top - i), right - (tRegion.left + i),
            bottom - (tRegion.top + i));
        if (screenID == ID_DIRECT)
          break;

      }
```

4. Draw. The low-level display calls depend on the type of function, such as **Rectangle( )**, **Ellipse( )**, **Polygon( )**, and whether the *fill* parameter is *TRUE* or *FALSE*.

```
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)
{
```

5. Update the screen quickly with **VirtualGet( )** and **VirtualPut( )**. Briefly, these functions allow us to optimize repetitive drawing tasks by copying part of the display into a buffer, draw into the buffer, and then copy the modified data out of the buffer and onto the screen. For more details, see ``**UI_BGI_DISPLAY**" in the *Programmer's Reference*.

```
// Update the screen.
  if (changedScreen)
    VirtualPut(screenID);
}
```

Information member functions

The display has two information functions. **TextHeight( )**, gets the maximum height of a string using a specific font. If the font parameter, *logicalFont*, has an entry in the font table, its associated value is returned. Otherwise, the Borland **textheight( )** function is called. **TextWidth( )** gets the width of the text displayed in the current font. Its operation is similar to that of **TextHeight( )**.

```
int UI_BGI_DISPLAY::TextHeight(const char *string, SCREENID,
  LOGICAL_FONT logicalFont)
{
  logicalFont &= 0x0FFF;
  SetFont(logicalFont);
  if (fontTable[logicalFont].maxHeight)
    return (fontTable[logicalFont].maxHeight);
  else if (string && *string)
    return (textheight((char *)string));
  else
    return (textheight("Mq"));
}

int UI_BGI_DISPLAY::TextWidth(const char *string, SCREENID,
  LOGICAL_FONT logicalFont)
{
  if (!string || !(*string))
    return (0);
  SetFont(logicalFont & 0x0FFF);
  int length = textwidth((char *)string);
  ...
  return (length);
}
```

Graphic display information functions must return the width and height of a string in pixel values. In addition, the text width or height should be returned, not the cell height and cell width defined by the *cellWidth* and *cell-Height* values.

Conclusion

In this chapter, we learned how to derive a display class from **UI_DISPLAY**, for a specific graphics library, **UI_BGI_DISPLAY**. If we had had to write **UI_BGI_DISPLAY** from scratch, we would have spent a lot more time. In the next chapter, we'll learn how to use Zinc's ability to detect language and locale at run time and change the locale of an object according to user input.

Chapter 17

Using Locales

In this chapter we will begin our discussion of how to globalize a Zinc application. We start by learning how to work with locales.

In this tutorial, we learn how to write a program for a department of Interpol, which maintains offices in France, Germany, and the United States, and whose responsibility is to track bank robberies in those countries. The Interpol MIS director asks us to write an Incident Report program that allows Interpol agents to record the date of the crime, the institution robbed, and the amount stolen. Since the program might be deployed in any of the Interpol international offices, and since the agents will record robberies in those countries, they must be able to record the type of currency stolen with the appropriate currency symbol.

Key Concepts

detecting the system locale

setting an object's locale

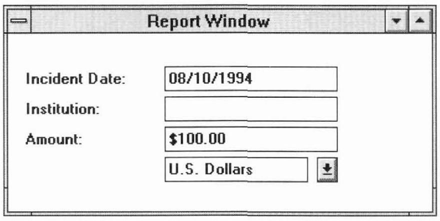
What we'll do

Here are the steps we'll take in writing **INTRPOL1.CPP**.

1. Load the report window from the **.DAT** file.
2. Determine what the system's default locale is and update the window accordingly.
3. Display the window.
4. If the user selects a different locale for the amount field, update the field's locale information and exchange the value for the new setting.

Running the program

Compile the source code and run the executable. You should see the following window on the screen:



Incident Date: 08/10/1994
Institution:
Amount: $100.00
U.S. Dollars

By default, the date and the currency symbol use the system's locale. So if the Interpol agent is running the application in Germany on a computer with a German configuration, the date will appear in the normal German fashion, and the amount will use the deutschemarks currency symbol. But if the German Interpol agent records a robbery that took place in France, the program will allow him to update the amount field with the currency symbol for francs. Note that the date field remains in the format specified by the system's configuration.

Source code

The source code for our tutorial is located in **\ZINC\TUTOR\GLOBAL**, and contains the following files:

- **INTRPOL1.CPP**. Contains the main event loop inside **UI_APPLICATION::Main( )**, as well as the implementation of the **REPORT_WINDOW** class.
- **INTRPOL1.HPP**. Contains the declaration for the **REPORT_-WINDOW** class and application constants and events.
- **IPOLWIN1.CPP**. The object table for the objects we created in the Designer.
- **IPOLWIN1.DAT**. The data file created in the Designer. Contains the data for creating the report window and its fields.
- **IPOLWIN1.HPP**. The header information for the window and its fields that we created in the Designer.
- ***.DEF**, ***.RC**. The definition and resource files when compiling for different environments.
- ***.MAK**. The compiler-dependent makefiles.

Analyzing the source code

The header file has three sections, **INTRPOL1.HPP**. The first section defines the following country identifiers:

```
const int GERMANY       = 0;
const int UNITED_STATES = 1;
const int FRANCE        = 2;
```

These country name constants are used to locate the proper exchange rate data when switching locales. We'll talk more about these constants later.

The next section contains some definitions for events specific to this application:

```
const ZIL_USER_EVENT LOCALE_FIRST       = 10000;
const ZIL_USER_EVENT GERMAN_LOC        = 10000;
const ZIL_USER_EVENT US_LOC            = 10001;
const ZIL_USER_EVENT FRANCE_LOC        = 10002;
const ZIL_USER_EVENT LOCALE_LAST       = 10010;
```

The program places these user-defined events on the event queue when the user changes locales by selecting an option from the combo box. We will trap these events in the **REPORT_WINDOW::Event( )** function.

The third section contains the definition for **REPORT_WINDOW**, the class used to display our reports. **REPORT_WINDOW** maintains a pointer to the amount field and the current locale name, since they are used fairly often. It also contains the **Event( )** function and a **ConvertAmount( )** function which is used to update the amount field when the user selects a new locale. Here's the definition for the **REPORT_WINDOW** class:

```
class REPORT_WINDOW : public UIW_WINDOW
{
public:
  REPORT_WINDOW(ZIL_ICHAR *name);
  ~REPORT_WINDOW(void);
  EVENT_TYPE Event(const UI_EVENT &event);
protected:
  void ConvertAmount(EVENT_TYPE ccode);
private:
  UI_WINDOW_OBJECT *amountField;
  ZIL_ICHAR *currentLocaleName;
};
```

REPORT_WINDOW uses the following member variables:

- *amountField,* a pointer to the **UIW_BIGNUM** used to display the amount.
- *currentLocaleName*, a string pointer that contains the two-letter ISO locale name currently displayed.

The main source file, **INTRPOL1.CPP**, contains four sections. The first section includes the header files:

```
#include <ui_win.hpp>
#include "intrpol1.hpp"
#include "ipolwin1.hpp"
```

Note that we included the header file generated by the Designer as well as the header file that has our application-specific code.

The second section sets up data:

```
// Create static strings used in application.
static ZIL_ICHAR _USLocaleString[] = { 'U','S', 0 };
static ZIL_ICHAR _DELocaleString[] = { 'D','E', 0 };
static ZIL_ICHAR _FRLocaleString[] = { 'F','R', 0 };
static ZIL_ICHAR _amountFieldName[] = {
'A','M','O','U','N','T','_','F','I','E','L','D', 0 };
static ZIL_ICHAR _convertBoxName[] = {
'C','O','N','V','E','R','S','I','O','N','S', 0 };
static ZIL_ICHAR _fileName[] = {
'i','p','o','l','w','i','n','1','.','d','a','t', 0 };
```

```
static ZIL_ICHAR _windowName[] = {
'R','E','P','O','R','T','_','W','I','N','D','O','W', 0 };
// Table for exchange rates and to identify locales.
static struct EXCHANGE
{
  int country;
  ZIL_ICHAR *ISOLocaleName;
  ZIL_RBIGNUM exchangeRate;
}_exchange[] =
{
  { GERMANY,          _DELocaleString,    1.5 },
  { UNITED_STATES,    _USLocaleString,    1.0 },
  { FRANCE,           _FRLocaleString,    0.5 },
  { -1, ZIL_NULLP(ZIL_ICHAR), 1.0 }
};
```

The first part of this data initialization creates Unicode-compatible strings for use in the application. The second part creates a structure that is used to look up exchange rates and identify locales.

The third part of the main source code file contains the definitions for the **REPORT_WINDOW** member functions. We will discuss the important parts of these functions when we look at the interface, below.

The fourth section is the definition of the **UI_APPLICATION::Main( )** function:

```
int UI_APPLICATION::Main(void)
{
  // The UI_APPLICATION constructor automatically initializes the
  // display, eventManager, and windowManager variables.
  // This line fixes linkers that don't look for main in the
  // .LIBs.
  UI_APPLICATION::LinkMain( );
  // Create derived window.
  UI_WINDOW_OBJECT::defaultStorage = new
    ZIL_STORAGE_READ_ONLY(_fileName);
  UIW_WINDOW *window = new REPORT_WINDOW(_windowName);
  // Add window to the window manager.
  *windowManager
    + window;
  // Process user responses.
  UI_APPLICATION::Control( );
  // Clean up.
  delete UI_WINDOW_OBJECT::defaultStorage;
  return (0);
}
```

We create a **UI_STORAGE_READ_ONLY** object to which we assign the **UI_WINDOW_OBJECT::defaultStorage**. This is the **.DAT** file that contains the Report Window. We won't describe the creation of the window using the Designer—if you need to review this process, see "Using the Designer" on page 139. **UI_APPLICATION::Main( )** also creates the Report Window and adds it to the Window Manager. The rest of this function you should be familiar with by now.

Program flow

Using **UI_APPLICATION::Main( )**'s built-in main event loop, our program flow is simple. We load the report window from the **.DAT** file and add it to the Window Manager. The report window determines what the system's locale is and updates its combo box accordingly. If the user selects a different locale from the combo box, the combo box option places a message on the event queue, which the **REPORT_WINDOW::Event( )** function uses. Then the event function updates the amount field. **REPORT_WINDOW::Event( )** passes all other events back to its base class, **UIW_WINDOW::Event( )**.

REPORT_WINDOW

Wiring up the interface

Once we've created the window, the next step is to "wire up" the interface so that we can trap user events and change the amount field's locale when the user requests it. In the constructor for **REPORT_WINDOW** we get a pointer to the amount field so that we can change its locale and value. We then get the initial locale being used by the system by inspecting *localeManager.defaultName*. *localeManager* is a global, static instance of **ZIL_LOCALE_MANAGER**. This object maintains all the application's locales.

Once we determine the system locale, the constructor determines if the application supports the locale by looking for the locale name in the *_exchange* structure. If the application does not support that locale, we set the application's locale to be the first entry in the structure as a default.

After we have a valid locale for the field, we update the combo box so when the application comes up, its selection matches the contents of the amount field by inspecting each object attached to the combo box, comparing its value to the current locale. Once we find the proper selection, we simply re-add it to the combo box. This makes it the current selection.

Changing locales

The **REPORT_WINDOW::Event( )** function is the heart of the application. While it doesn't have much code in it, all our functionality really exists there.

Whenever the user selects a locale option from the combo box, a message is put on the event queue because the combo box options are **UIW_BUTTON**s with the *BTF_SEND_MESSAGE* flag set. The event that is put on the queue is one of the events that we defined in the header file. After that event is pulled off the queue and sent to the Window Manager by the **UI_APPLICATION::Control( )** function, the Window Manager will route the event to the Report Window. We trap for those messages in the **Event( )** function:

```
EVENT_TYPE REPORT_WINDOW::Event(const UI_EVENT &event)
{
  // Get the logical event.
  EVENT_TYPE ccode = LogicalEvent(event);
  // Check to see if the event is one of ours.
  if (ccode >= LOCALE_FIRST && ccode <= LOCALE_LAST)
    ConvertAmount(ccode);
  // If it's not our event, pass it to the UIW_WINDOW base class.
  else
    ccode = UIW_WINDOW::Event(event);
  return (ccode);
}
```

Any other messages are passed to the base class's **Event( )** function so that the object can process them properly.

When we get a message to change the locale, we call **REPORT_WINDOW::ConvertAmount( )**, passing it the message we received. The most important thing **ConvertAmount( )** does is set the locale for the amount field. It does this by getting a pointer to the **ZIL_BIGNUM** used by the **UIW_BIGNUM** object, and then calling the **ZIL_BIGNUM**'s **SetLocale(S)** function:

```
// Set the new locale.
amount->SetLocale(_exchange[newLocale].ISOLocaleName);
```

The rest of the code in **ConvertAmount( )** is related to changing the monetary value using the exchange rates, and so we won't discuss that here.

Conclusion

In this chapter, we learned how to detect which locale the system is using and how to set which locale a particular instance of an object is using. We also learned how to set a combo box entry. In the next chapter we will extend this tutorial and learn how to switch languages at run time.

Chapter 18

Using Languages

In the last chapter we began a discussion of globalizing applications by learning how to use locales. In this chapter we will continue by learning how to work with languages in our application. We will continue with the Interpol example we began in the last chapter and expand it to allow switching of languages at run time.

Key Concepts

- detecting the system language
- setting the application language

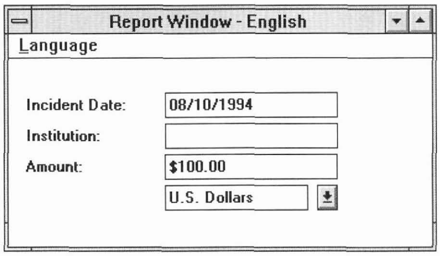
What we'll do

Here are the steps we'll take in writing **INTRPOL2.CPP**.

1. Determine what the system's default language is and load the proper window.
2. Display the window.
3. If the user selects a different language for the application, load the new window and delete the old window.

Running the program

Compile the source code and run the executable. You should see this window:



Incident Date:	08/10/1994
Institution:	
Amount:	$100.00
	U.S. Dollars

Notice that this window is the same as the one we saw in the last chapter, except that this window has a pull-down menu. By default, the window uses the language used by the system if our program supports that language. So if the Interpol agent happens to work in Germany on a computer with a German configuration, the program will detect that and bring up a German window will appear. If the user selects a new language from the pull-down menu, a the program will load a new window in that language and the discard the old window.

Source code

The source code for our tutorial is located in **\ZINC\TUTOR\GLOBAL**, and contains the following files:

- **INTRPOL2.CPP**. Contains the main event loop inside **UI_APPLICATION::Main( )**, as well as the implementation of the **REPORT_WINDOW** class.
- **INTRPOL2.HPP**. Contains the declaration for the **REPORT_-WINDOW** class and application constants and events..
- **IPOLWIN2.CPP**. The object table for the objects we created in the Designer.
- **IPOLWIN2.EN, IPOLWIN2.DE, IPOLWIN2.FR**. The data files created in the Designer. Each contains the data for creating the report window and its fields for the language identified by the file's extension.
- **IPOLWIN2.HPP**. The header information for the window and its fields that we created in the Designer.
- ***.DEF, *.RC**. The definition and resource files when compiling for different environments.
- ***.MAK**. The compiler-dependent makefiles.

Analyzing the source code

We've defined several new events to allow **INTRPOL2.HPP** to changing languages:

```
const ZIL_USER_EVENT LANGUAGE_FIRST = 10020;
const ZIL_USER_EVENT GERMAN_LANG    = 10020;
const ZIL_USER_EVENT ENGLISH_LANG   = 10021;
const ZIL_USER_EVENT FRENCH_LANG    = 10022;
const ZIL_USER_EVENT LANGUAGE_LAST  = 10030;

const ZIL_USER_EVENT DELETE_OBJECT  = 10040;
```

The first five user-defined events are those the program places on the event queue when the user changes languages by selecting an option from the pull-down menu. We will trap these events in the **REPORT_WINDOW::Event( )** function.

The last event is used to delete the old window when a new language is selected. We trap this event in the **REPORT_WINDOW::Event( )** function, as well.

We added several new strings to **INTRPOL2.CPP** to accommodate different languages:

```
static ZIL_ICHAR _enLanguageString[] = { 'e','n', 0 };
static ZIL_ICHAR _deLanguageString[] = { 'd','e', 0 };
static ZIL_ICHAR _frLanguageString[] = { 'f','r', 0 };
```

The *_fileName* string changed slightly to reflect the different **.DAT** files being used.

The *_exchange* structure expanded to include an entry for the language:

```
// Table for exchange rates and to identify locales.
static struct EXCHANGE
{
    int country;
  ZIL_ICHAR *ISOLocaleName;
  ZIL_ICHAR *ISOLanguageName;
  ZIL_RBIGNUM exchangeRate;
}_exchange[] =
{
  { GERMANY, _DELocaleString, _deLanguageString, 1.5 },
  { UNITED_STATES, _USLocaleString, _enLanguageString, 1.0 },
  { FRANCE, _FRLocaleString, _frLanguageString, 0.5 },
  { -1, ZIL_NULLP(ZIL_ICHAR), ZIL_NULLP(ZIL_ICHAR), 1.0 }
};
```

A new global function, **CreateWindow( )**, was added to the application. This function takes an identifier which specifies which entry in the *_exchange* table corresponds to the language in use. The function then obtains the language name from the table, creates a new default storage, and loads the proper Report Window.

The **REPORT_WINDOW::Event( )** function is the only member function that changed for this application. We added two sections to the function: one to change languages and the other to handle the deletion of the old Report Window. We will discuss how these are accomplished when we talk about the interface below.

The last section that changed in **INTRPOL2.CPP** is the **UI_APPLICATION::Main( )** function, which we updated to check for the system's language and then to load an appropriate window:

```
  // Get default system language name.  languageManager is a
  // global library variable that contains all the ZIL_LANGUAGE
  // objects.
  ZIL_ICHAR *currentLanguageName = languageManager.defaultName;
  // Locate the entry in the EXCHANGE structure for the default
  // language.
  int currentLanguage = -1;
  for (int i = 0; _exchange[i].ISOLocaleName; ++i)
  {
    if (strcmp(_exchange[i].ISOLanguageName,
        currentLanguageName) == 0)
        currentLanguage = i;
    }

    // If system language doesn't correspond to one supported by the
    // application, then use a default language.
    if (currentLanguage == -1)
      currentLanguage = 0;

    // Add window to the window manager.
    *windowManager
      + CreateWindow(currentLanguage);
```

If the system's language is not supported by the application, we assign a default language and load the window. We will discuss this later on in the chapter.

Program flow

Using **UI_APPLICATION::Main( )**'s built-in main event loop, our program flow is simple. We first determine the system's language and load the proper Report Window from the **.DAT** file and add it to the Window Manager. If the user selects a different language from the pull-down menu, the menu item places a message on the event queue which is routed to the **REPORT_WINDOW::Event( )** function. The proper window is then loaded and displayed and the old window deleted. All other events that we don't handle are passed by **REPORT_WINDOW::Event( )** back to the base class **UIW_WINDOW::Event( )**.

REPORT_WINDOW

Wiring up the interface

Once we've created the window, the next step is to "wire up" the interface so that we can trap user events and change the application's language when the user requests it In **UI_APPLICATION::Main( )**, we look at the system's language and determine if it is one that the application supports. To get the language we simply inspect *languageManager.defaultName*. *languageManager* is a global, static instance of **ZIL_LANGUAGE_MANAGER**. All languages used by the application are maintained by this object. We determine if the application supports the language by looking for the language name in the *_exchange* structure. If the application dos not support that language, we set the language to be the first entry in the structure as a default. We then load the proper window.

Changing languages

Whenever the user selects a language option from the pull-down menu, a message is put on the event queue because the pop-up items options have the *MNIF_SEND_MESSAGE* flag set. The event that is put on the queue is one of the events that we defined in the header file. After that event is pulled off the queue and sent to the Window Manager by the **UI_APPLICATION::Control( )** function, the Window Manager will route the event to the Report Window. We trap those messages in the **Event( )** function:

```
// Change language.
else if (ccode >= LANGUAGE_FIRST && ccode <= LANGUAGE_LAST)
{
  // Delete old default storage.
  delete UI_WINDOW_OBJECT::defaultStorage;
  // Determine language to load.
  int currentLanguage = -1;
  for (int i = 0; _exchange[i].ISOLocaleName; ++i)
  {
    if (_exchange[i].country + LANGUAGE_FIRST == ccode)
      currentLanguage = i;
  }
   // Change the application's default language.
  languageManager.LoadDefaultLanguage(
    _exchange[currentLanguage].ISOLanguageName);
  // Create new window.
  *windowManager
    + CreateWindow(currentLanguage);

  // Cause current window to be subtracted.
  UI_EVENT tEvent;
  tEvent.type = S_SUBTRACT_OBJECT;
  tEvent.data = this;
  eventManager->Put(tEvent);
  // Cause current window to be deleted.
  tEvent.type = DELETE_OBJECT;
  tEvent.windowObject = this;
  eventManager->Put(tEvent);
}
```

When a message to change languages arrives, the first thing the function does is delete the old default storage. It then locates the proper entry in the *_exchange* table for the new language. To set the application's language it calls *languageManager*.**LoadDefaultLanguage( )**. This will cause all library strings to be displayed in the new language. After setting the application's language we call **CreateWindow( )**, which loads the new window.

The **Event( )** function then puts two messages on the event queue to remove the old language window. We can't simply delete the window because the program is running in an instance of the window. Nor can we simply place an *S_CLOSE* message on the event queue, because by the time the program will processed it, the current window will be the new language window. So we have to subtract and delete the window ourselves.

The first event we place on the event queue is *S_SUBTRACT_OBJECT*. This event is processed by the Window Manager when it receives it from the **UI_APPLICATION::Control( )** function. The second message placed on the queue is *DELETE_OBJECT*, which is one that we defined for this application. It will be handled by the new Report Window's **Event( )** function.

The section that handles the *DELETE_OBJECT* message is the second new part of the **Event( )** function:

```
// Delete old window.
else if (ccode == DELETE_OBJECT)
  delete event.windowObject;
```

As usual, any events that we don't handle are passed to the base class **Event( )** function.

Conclusion

In this chapter, we learned how to detect which language the system is using and how to set which language the application is using. We also learned one technique for switching windows at run time. In the next chapter, we'll learn about the design of a large, complex Zinc application.

Chapter 19

Program Design

In this chapter, we'll learn how to write a complex program using Zinc. Our program, called ZincApp, contains several objects that perform specialized tasks, and communicate with the main control window by sending messages. The main control window then responds to these messages by calling certain member functions.

Key Concepts

design of a large application

using event map tables

using accelerator keys

What we'll do

Here's what we'll do in this chapter.

1. Discuss ZincApp's design and implementation.
2. Examine what happens when the user selects each option.

Source code

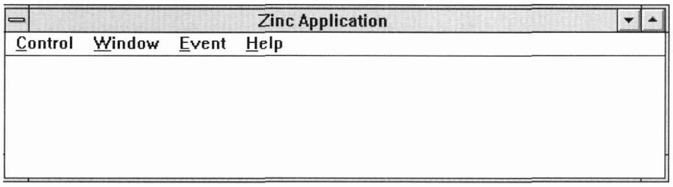
ZincApp source is located in **ZINC\TUTOR\ZINCAPP**. Here's a list of ZincApp's source code components and what each contains:

- **ZINCAPP.CPP**. The main program loop, and the **main( )** or **WinMain( )** function.
- **ZINCAPP.HPP**. Definition of the display, window, event, and help messages that pass through the system when the user selects a pop-up item from the main control window. Also contains the declarations for the **ZINCAPP_WINDOW_MANAGER**, **CONTROL_WINDOW**, and **EVENT_MONITOR** classes.
- **CONTROL.CPP**. Contains member functions which we'll use to create the main control menu and to handle all main control throughout the program. Here are those member functions:
  CONTROL_WINDOW::CONTROL_WINDOW( ), **CONTROL_WINDOW::Event( ), CONTROL_WINDOW::Message( ), ZINCAPP_WINDOW_MANAGER::Event( ), ZINCAPP_WINDOW_MANAGER::ExitFunction( )**
- **SUPPORT.CPP**. The object table that must be compiled with the program since persistent window objects are to be used.
- **SUPPORT.DAT**. The binary data file created by Zinc Designer, which contains the help context and persistent window object information.
- **SUPPORT.HPP.** The help context constant information used to associate a help context with a window. It also contains the persistent object identification values entered as the *stringID* field for each object in the **.DAT** file.
- **DISPLAY.CPP.** Contains the **CONTROL_WINDOW::Option_Display( )** member function. Changes the type of display.

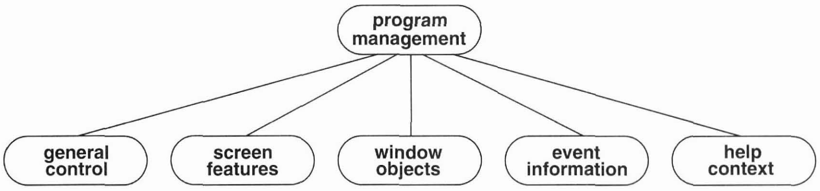
- **EVENT.CPP.** Contains the **CONTROL_WINDOW::Option_Event( )** and **EVENT_MONITOR( )** member functions. Process all the messages that are produced when an **Event** menu item is selected from the main control window.
- **HELP.CPP.** Contains the **CONTROL_WINDOW::OptionHelp( )** member function. It processes all of the messages that are produced when a **Help** menu item is selected from the main control window.
- **WINDOW.CPP.** Contains the **CONTROL_WINDOW::OptionWindow( )** member function. This function invokes the proper window that was selected from the main control window by processing all the messages that are produced when a menu item is selected.
- ***.DEF, *.RC**. The environment-specific definition and resource files required when compiling for other environments..
- ***.MAK**. The compiler-dependent makefiles associated with ZincApp.

Program specification

ZincApp provides a single control window, with pull-down items in a pull-down menu displaying selections. This control window gives the user easy access to all functions. We could write ZincApp with multiple windows, but then it might suffer from a common malady of graphical user interfaces called "windowitis," where the application's functionality is spread over too many windows.



The control window controls the pull-down items, which in turn control items within their scope. For example, the control window may pass control to a pull-down item that handles screen functionality. In turn, this pop-up item may send a message through the system, requesting that some other object perform some action.



each item has a call function or a send message function

Design and implementation

ZincApp consists of several parts:

- the Event Manager, which contains the event queue;
- the ZincApp window manager class, derived from the Zinc Window Manager;
- the event monitor, which receives events from the ZincApp Window Manager and displays them;
- various pull-down menu options, which place a message on the queue when selected;
- and the Control Window, which contains the member functions that allow ZincApp to respond to user input.

Here's what happens when we launch ZincApp.

1. The **CONTROL_WINDOW** constructor sets up the window and menu items. Here's a partial listing of the constructor:

```
CONTROL_WINDOW::CONTROL_WINDOW(void) : UIW_WINDOW(0, 0, 76, 6,
  WOF_NO_FLAGS, WOAF_LOCKED)
{
        // Control menu items.
        static UI_ITEM controlItems[] =
        {
          { S_REDISPLAY,VOIDF(CONTROL_WINDOW::Message),
            "&Refresh\tShift+F6", MNIF_NO_FLAGS },
          { 0,VOIDF(0),"",MNIF_NO_FLAGS },// item separator
          { L_EXIT_FUNCTION,VOIDF(CONTROL_WINDOW::Message),
            "E&xit\tAlt+F4",MNIF_NO_FLAGS },
          { 0, 0, 0, 0 }// End of array.
        };
        ...
        // Attach the sub-window objects to the control window.
        *this
          + new UIW_BORDER
          + new UIW_MAXIMIZE_BUTTON
          + new UIW_MINIMIZE_BUTTON
          + new UIW_SYSTEM_BUTTON(SYF_GENERIC)
          + new UIW_TITLE("Zinc Application")
          + &(*new UIW_PULL_DOWN_MENU
            + new UIW_PULL_DOWN_ITEM("&Control", WNF_NO_FLAGS,
              controlItems)
            + new UIW_PULL_DOWN_ITEM("&Display", WNF_NO_FLAGS,
              displayItems)
            + &(*new UIW_PULL_DOWN_ITEM("&Window", WNF_NO_FLAGS)
              + controlObjects
              + inputObjects
              + selectObjects)
            + new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS, eventItems)
            + new UIW_PULL_DOWN_ITEM("&Help", WNF_NO_FLAGS, helpItems));
      }
```

Part of the task of the control window's constructor is to initialize the **UI_ITEM** array contained in each pull-down item in the control window's menu bar. This array contains:

The message. The first field in the **UI_ITEM** structure. For example, the first **Control** menu item, **Refresh**, contains the message *S_REDISPLAY,* which will pass through the system whenever the user selects the **Control | Refresh** menu item.

The user function. Called when the user selects a menu item. All menu items specify **CONTROL_WINDOW::Message( )** as their user function.

The string information. The text displayed on the screen. The string for the **Refresh** menu item is "&Refresh\tShift+F6." We'll discuss the "Shift+F6" portion of the string later in this chapter. Note that this hotkey only works in DOS.

The menu item flags. These control how the menu items look and act. For example, *MNIF_CHECK_MARK* tells the menu item to display a check mark to the left of the menu item's text when selected.

2. When the user selects an option, two events are generated. The first is the system event, handled top down or bottom up, according to the type of operating environment. The second is the event that a pull-down menu will place on the event queue for retrieval by the control window.

   The control window responds to events by overriding the **Event( )** virtual function in the base **UIW_WINDOW** class. Here it is:

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

   Then member functions inside the control window then call the appropriate member function, passing the *event.type* of the event as a parameter:

   ```
   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);
   };
   ```

   Depending on the circumstances, however, one member function of the control window will send a message through the system, whereas another may call another member function. For example, **OptionDisplay( )** doesn't reset the display, but sends a message through the system instead. Conversely, **OptionEvent( )** creates an event monitor object with a member function without creating an additional message.

   The control window will receive four types of messages:

   - *Display option messages.* 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.* Generated when a **Window** menu item has been selected from the main control window. Processed by the **OptionWindow( )** member function.
   - *Event option messages.* Generated when an **Event** menu item has been selected from the main control window. Processed by the **OptionEvent( )** member function.

- *Help option messages.* Generated when a **Help** menu item has been selected from the main control window. Processed by the **OptionHelp( )** member function.

The **UIW_WINDOW::Event( )** member function processes all other messages. Note that the Window Manager automatically processes the control option messages, since they represent operations handled by the Window Manager.

Accelerator keys

ZincApp uses two accelerator keys:

<*Shift+F6*>. Causes the Window Manager to clear the screen and to redisplay each window attached to the Window Manager's list of window objects.

<*Alt+F4*>. Causes the exit application window to appear on the screen.

The **CONTROL_WINDOW::Event( )** function contains the implementation of the accelerator keys.

```
EVENT_TYPE CONTROL_WINDOW::Event(const UI_EVENT &event)
{
  // Check for an accelerator key.
  EVENT_TYPE ccode = event.type;
  if (ccode == L_EXIT_FUNCTION)
    eventManager->Put(UI_EVENT(L_EXIT_FUNCTION));
  if (ccode == E_KEY)
  {
    // Define the set of accelerator keys.
    static struct ACCELERATOR_PAIR
    {
      RAW_CODE rawCode;
      LOGICAL_EVENT logicalType;
    } acceleratorTable[] =
    {
      { SHIFT_F6,S_REDISPLAY },
      { ALT_F4,L_EXIT_FUNCTION },
      { 0, 0 }// End of array.
    };
    for (int i = 0; acceleratorTable[i].rawCode; i++)
      if (event.rawCode == acceleratorTable[i].rawCode)
      {
        UI_EVENT tEvent(acceleratorTable[i].logicalType);
        eventManager->Put(tEvent);// Put the accelerator key
        return (ccode);// into the system.
      }
  }
  // Process the event according to its type.
  if (ccode >= MSG_HELP)
```

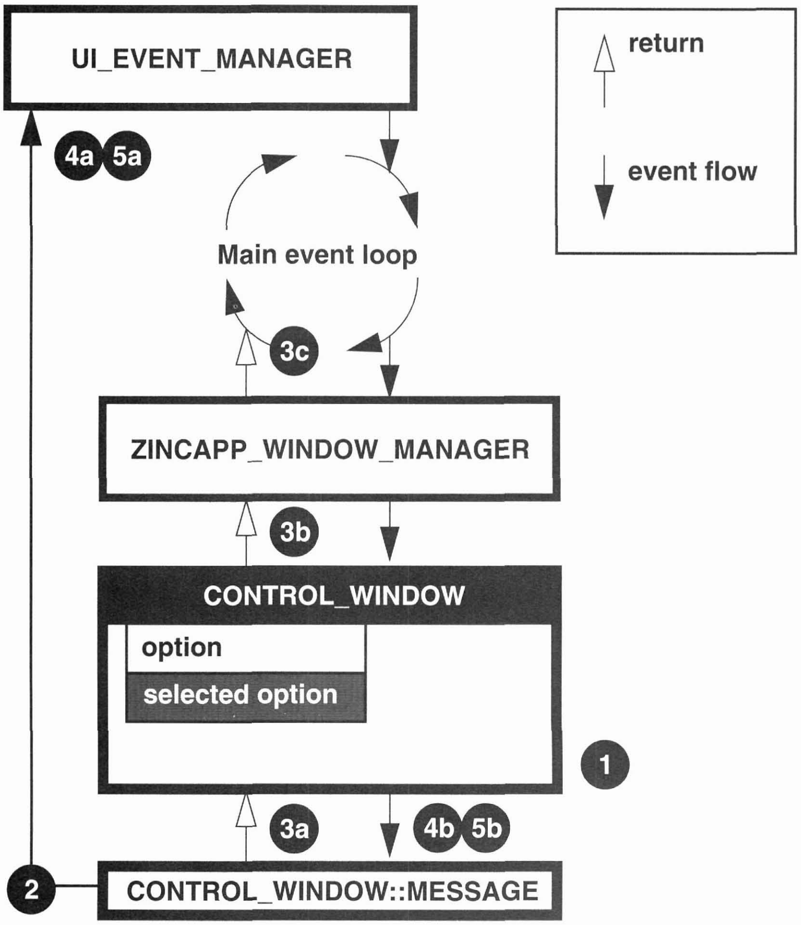
```
        OptionHelp(event.type);// Help menu option selected.
      else if (ccode >= MSG_EVENT)
        OptionEvent(event.type);// Event menu option selected.
      else if (ccode >= MSG_WINDOW)
        OptionWindow(event.type);// Window menu option selected.
      else if (ccode >= MSG_DISPLAY)
        OptionDisplay(event.type); // Display menu option selected.
      else if (ccode >= MSG_CONTROL)
      {
          UI_EVENT tEvent(event.type);
          eventManager->Put(tEvent);// Put the accelerator key
      }
      else
        ccode = UIW_WINDOW::Event(event);// Unknown event.
      // Return the control code.
      return (ccode);
    }
```

Here's what happens when the user presses an accelerator key:

1. **CONTROL_WINDOW::Event( )** receives the event from the Window Manager.
2. If the event is a normal key, the control window searches its list of raw code/logical type pairs.
3. If an accelerator key is detected, its logical value is placed into the Event Manager. The Window Manager interprets its value when the main program loop gets the next key using *eventManager*->**Get( )**. The definition of the two accelerator keys is given by the *acceleratorTable* static array shown above. Note that the accelerator keys are available only when the main control window is the front window.

General program flow

What happens when the user selects one of the options in the menu bar? Though the Zinc window manager handles the **Control** option, the control window handles the others in steps one through four. At the fifth step, however, the control window calls a different member function associated with the option that the user selected.



1. After the user selects a menu item, the **UIW_POP_UP_ITEM::Event( )** function calls the **CONTROL_WINDOW::Message( )** function.

   ```
   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 pop-up item's **Event( )** function passes some arguments to **Message( )**. Those arguments are

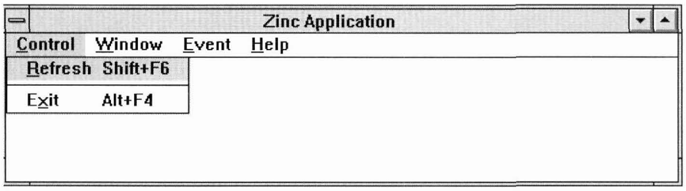
   - a pointer to the selected display option, *this*;
   - a copy of the event that caused the user function to be called, *tEvent*; and
   - the logical interpretation, *ccode*, of the event that caused **Event( )** to be called. Notice the variable *tEvent* needs to be a copy of *event,* since it's a 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 to Event Manager and thereby through the system. It then sends the display request through the system by setting *event.type* to be the menu item's value, for example, to 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 event loop, first by exiting **CONTROL_WINDOW::Message( )**, and then by exiting the **Event( )** virtual functions of the **UIW_POP_UP_ITEM**, **CONTROL_WINDOW**, and **ZINCAPP_WINDOW_MANAGER** classes.
4. *eventManager*->**Get( )** gets two messages that the program generates from the event queue. The first message is *S_CLOSE_TEMPORARY.* Responding to this message, the Window Manager removes the display options menu from the screen.
5. The second message tells the control window which menu option the user selected. In the following parts of this chapter, we'll examine what the control window does when it receives one of these messages.

Control



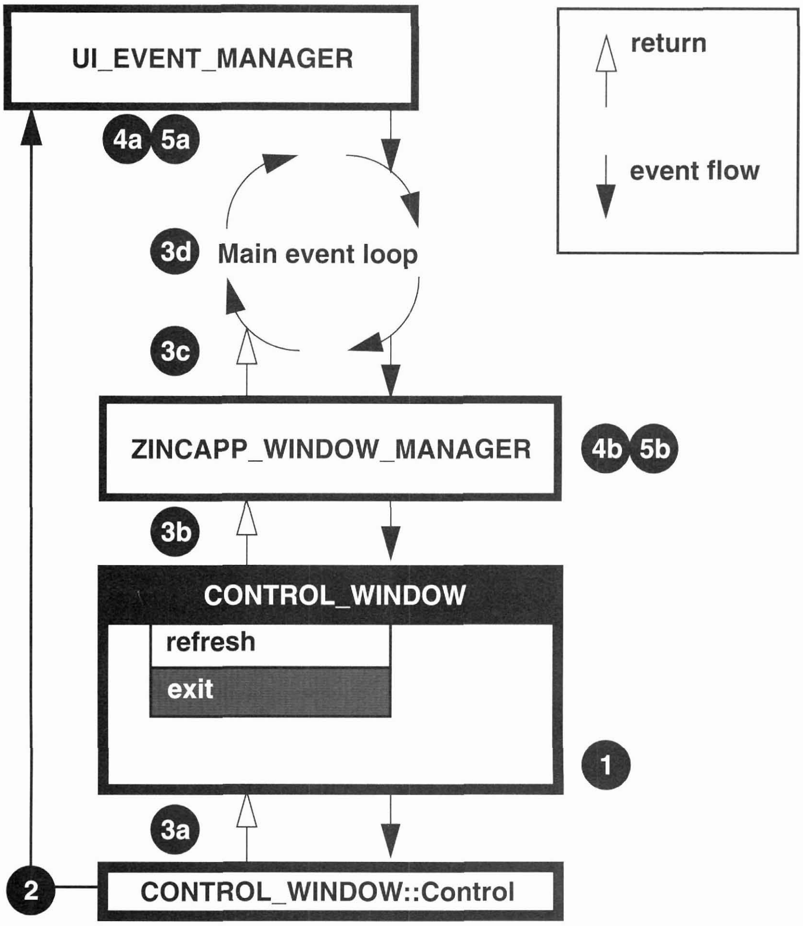
This item contains ZincApp's control options, **Refresh** and **Exit**, which refresh the screen and allow the user to exit the application. The **CONTROL_WINDOW** constructor initializes these options here:

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

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

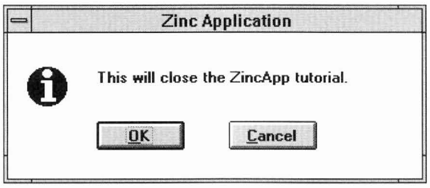
Control program flow

What happens when the user selects the **Control** option? First, the window executes steps one through four of the general program flow. Then it executes a fifth step.



1. The **UIW_POP_UP_ITEM::Event( )** function calls the **CONTROL_WINDOW::Message( )** function.
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.
3. Control returns to the main event loop.

4. *eventManager*->**Get( )** gets two messages that the program generates. The first message it gets is *S_CLOSE_TEMPORARY.*
5. The second message it gets is the value of the menu item from *event.type*, which it passes to the Window Manager by calling *windowManager*->**Event( )**. When the Window Manager receives the following messages, it performs the corresponding actions:
   - *S_REDISPLAY*—Clears the screen and redisplays each window in the Window Manager's list of window objects.
   - *L_EXIT_FUNCTION*—The Window Manager calls the **CONTROL_WINDOW::ExitFunction( )** function, which displays an exit window on the screen.

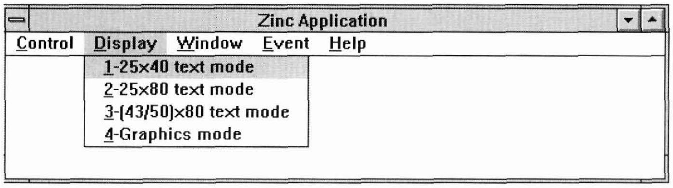

Zinc Application

This will close the ZincApp tutorial.

If the user selects **OK**, the Window Manager sends an *L_EXIT* message through the system. The main program breaks from the main loop and exit the application.

Note that in the **Control** option, the Window Manager and not the control window responds to the message.

Display options

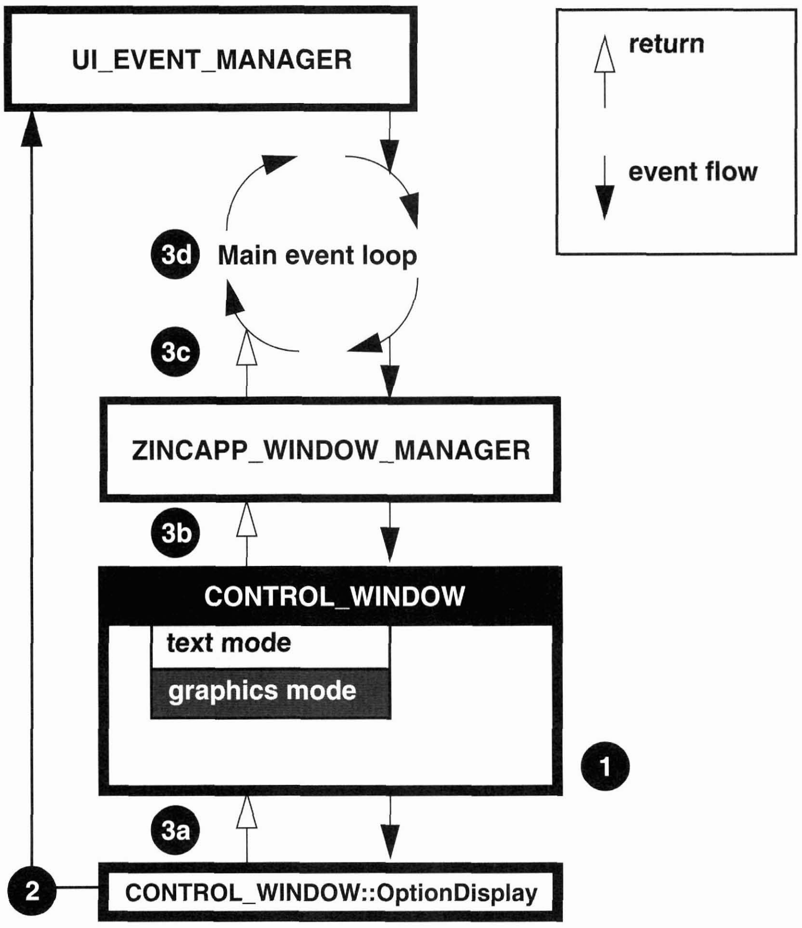


This menu item, available only under DOS, contains ZincApp's display options, initialized by the **CONTROL_WINDOW** constructor. Here's that part of the constructor.

```
static UI_ITEM displayItems[] =
  {
  #if defined (ZIL_MSDOS)
    { MSG_25x40_MODE,Message,"&1-25x40 text mode",
      MNIF_NO_FLAGS },
    { MSG_25x80_MODE,Message,"&2-25x80 text mode",
      MNIF_NO_FLAGS },
    { MSG_43x80_MODE,Message,"&3-(43/50)x80 text mode",
      MNIF_NO_FLAGS },
    { MSG_GRAPHICS_MODE,Message,"&4-Graphics mode",MNIF_NO_FLAGS },
    { MSG_WINDOWS_MODE,Message,"&5-Windows 3.X mode",
      MNIF_NON_SELECTABLE },
  #endif
    { 0, 0, 0, 0 }// End of array.
  };
  ...
  // Attach the sub-window objects to the control window.
  *this
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON(SYF_GENERIC)
    + new UIW_TITLE("Zinc Application")
    + &(*new UIW_PULL_DOWN_MENU
      + new UIW_PULL_DOWN_ITEM("&Control", WNF_NO_FLAGS, controlItems)
      + new UIW_PULL_DOWN_ITEM("&Display", WNF_NO_FLAGS, displayItems)
      + &(*new UIW_PULL_DOWN_ITEM("&Window", WNF_NO_FLAGS)
        + controlObjects
        + inputObjects
        + selectObjects)
      + new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS, eventItems)
      + new UIW_PULL_DOWN_ITEM("&Help", WNF_NO_FLAGS, helpItems));
```

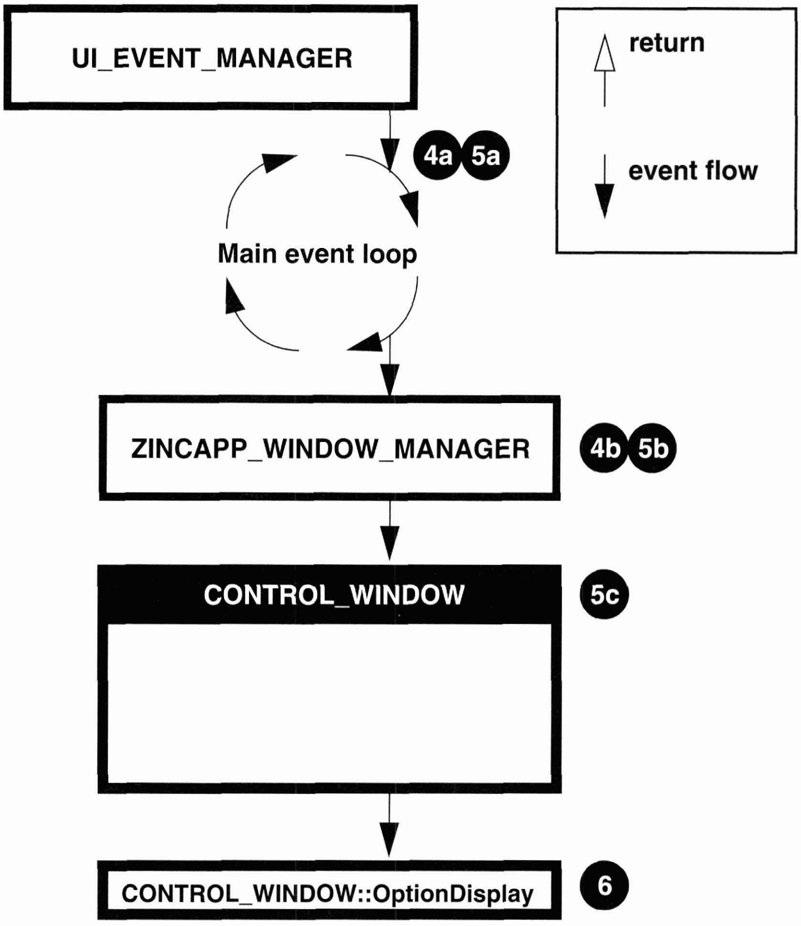
Display program flow

What happens when the user selects the **D̲isplay** option? First, the control window executes steps one through four of the general program flow. At the fifth step, however, it calls the **OptionsDisplay( )** member function.



1. The **UIW_POP_UP_ITEM::Event( )** function calls the **CONTROL_WINDOW::Message( )** function.
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.
3. Control returns to the main event loop.

4. *eventManager*->**Get( )** gets two messages that the program generates. The first message it gets is *S_CLOSE_TEMPORARY.*

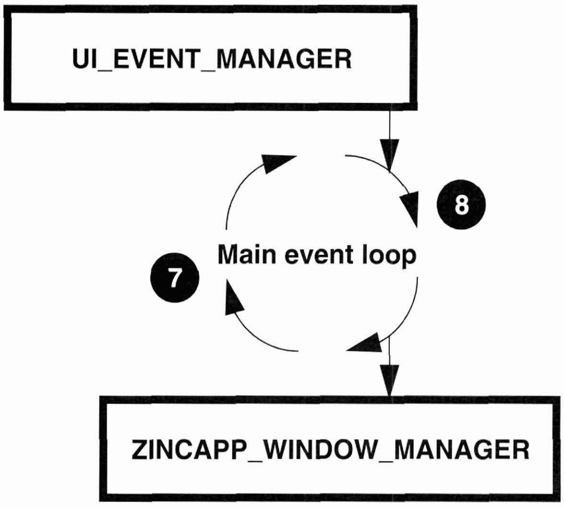


5. The second message it receives is the display message determined by the selected menu item. This message is passed by the main loop to the Window Manager, then is sent by the Window Manager to **CONTROL_WINDOW::Event( )** since the control window is the front window on the screen. The control window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

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

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

6. The **OptionDisplay( )** member function evaluates the item's value, which was passed down through the *item* argument, to determine which type of display has been requested. At this stage, however, no display is recreated. Instead, an *S_RESET_DISPLAY* is generated and passed through the system. We must create and delete displays at the highest level of the program, since that is where we initialized the *display* object, and since that is where the program destroys the display when it goes out of scope. The following code shows how this message is sent:

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

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

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

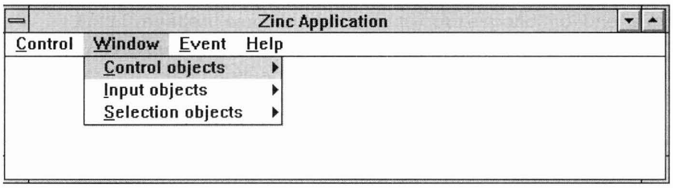


7. Control returns once again to the main event loop by exiting the associated **Event( )** functions.
8. The main loop picks up the *S_RESET_DISPLAY* message by calling *eventManager*->**Get( )**. This message causes the program to
   - tell the Event and window managers that the old display is about to be deleted. This allows them to uninitialize any display dependent information they may have.
   - construct the new display, the type of which is determined by *event.rawCode*.
   - 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 reinitialize themselves using the new display and coordinate system.

```
// Wait for user response.
EVENT_TYPE ccode;
UI_EVENT event;
do
{
  // Get input from the user.
  eventManager->Get(event);
  // Check for a screen reset message.
  if (event.type == S_RESET_DISPLAY)
  {
    #if defined(ZIL_MSDOS)
      event.data = NULL;
      // Tell the managers we changed the display.
      windowManager->Event(event);
      eventManager->Event(event);
      delete display;
      if (event.rawCode == TDM_NONE)
      {
        display = new UI_GRAPHICS_DISPLAY;
        if (!display->installed)
        {
          delete display;
          display = new UI_TEXT_DISPLAY;
        }
      }
      else
        display = new UI_TEXT_DISPLAY(event.rawCode);

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

If we examine the **CONTROL_WINDOW::OptionDisplay( )** member function and the code in the main event loop, we'll find 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.

Window options

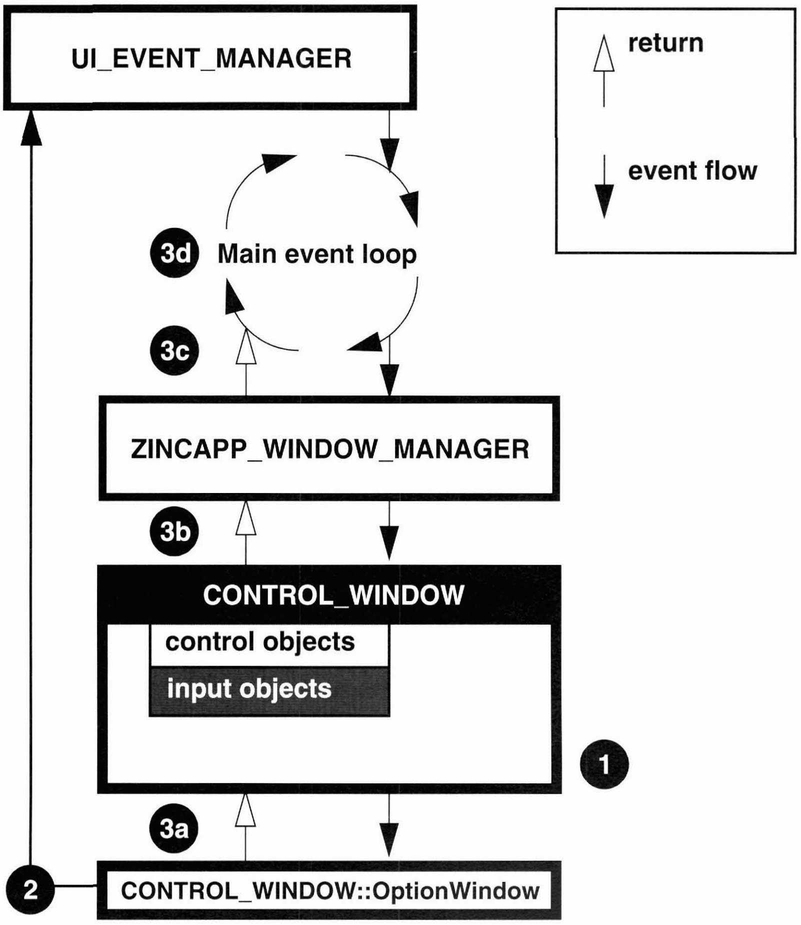


This item contains ZincApp's window options, initialized by the **CONTROL_WINDOW** constructor. Here's that part of the constructor.

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

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

UIW_POP_UP_ITEM *selectObjects = new UIW_POP_UP_ITEM("&Selection objects");
*selectObjects
   + new UIW_POP_UP_ITEM("&Combo Box window...", MNIF_NO_FLAGS,
     BTF_NO_FLAGS, WOF_NO_FLAGS, CONTROL_WINDOW::Message,
     MSG_COMBO_BOX_WINDOW)
   + new UIW_POP_UP_ITEM("&List window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
     WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_LIST_WINDOW)
   + new UIW_POP_UP_ITEM("&Menu window...", MNIF_NO_FLAGS, BTF_NO_FLAGS,
     WOF_NO_FLAGS, CONTROL_WINDOW::Message, MSG_MENU_WINDOW)
   + new UIW_POP_UP_ITEM("&Tool Bar window...", MNIF_NO_FLAGS,
    BTF_NO_FLAGS, WOF_NO_FLAGS, CONTROL_WINDOW::Message,
    MSG_TOOL_BAR_WINDOW);
...
// Attach the sub-window objects to the control window.
*this
+ new UIW_BORDER
+ new UIW_MAXIMIZE_BUTTON
+ new UIW_MINIMIZE_BUTTON
+ new UIW_SYSTEM_BUTTON(SYF_GENERIC)
+ new UIW_TITLE("Zinc Application")
+ &(*new UIW_PULL_DOWN_MENU
  + new UIW_PULL_DOWN_ITEM("&Control", WNF_NO_FLAGS, controlItems)
  + new UIW_PULL_DOWN_ITEM("&Display", WNF_NO_FLAGS, displayItems)
  + &(*new UIW_PULL_DOWN_ITEM("&Window", WNF_NO_FLAGS)
    + controlObjects
    + inputObjects
    + selectionObjects)
  + new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS, eventItems)
  + new UIW_PULL_DOWN_ITEM("&Help", WNF_NO_FLAGS, helpItems));
```

Window program flow

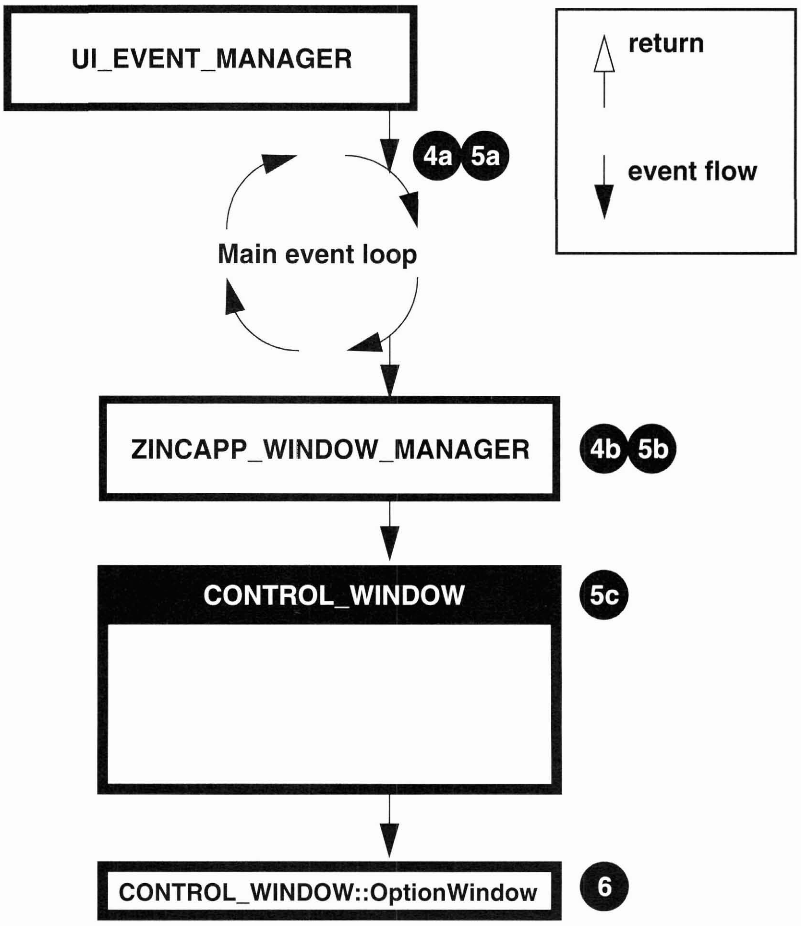
What happens when the user selects the **W̲indow** option? First, the control window executes steps one through four of the general program flow. At the fifth step, however, it calls the **OptionsWindow( )** member function.



1. The **UIW_POP_UP_ITEM::Event( )** function calls the **CONTROL_WINDOW::Message( )** function.
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.
3. Control returns to the main event loop.

4. *eventManager*->**Get( )** gets two messages that the program generates. The first message it gets is *S_CLOSE_TEMPORARY.*
5. The second message it gets is the window request of the selected menu item. This message is passed by the main loop to the Window Manager and is then dispatched by the Window Manager to **CONTROL_WINDOW::Event( )** since the control window is the front window on the screen. The control window evaluates *event.type*, which is, in this case a *MSG_WINDOW* message—resulting in the **OptionWindow( )** member function being called.

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



6. The **OptionWindow( )** member function evaluates the item's value, passed down through the *item* argument, to determine which type of window the user has requested. Then it calls the member function that constructs the appropriate window. Finally, it attaches the window to the Window Manager using the overloaded + operator. The following code shows how:

```
void CONTROL_WINDOW::OptionWindow(EVENT_TYPE item)
{
  // Get the specified window.
  UI_WINDOW_OBJECT *object = NULL;
  switch(item)
  {
  case MSG_DATE_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_DATE");
    break;
  case MSG_GENERIC_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_GENERIC");
    break;
  case MSG_ICON_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_ICON");
    break;
  case MSG_LIST_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_LIST");
    break;
  case MSG_COMBO_BOX_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_COMBO_BOX");
    break;
  case MSG_MENU_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_MENU");
    break;
  case MSG_NUMBER_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_NUMBER");
    break;
  case MSG_STRING_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_STRING");
    break;
  case MSG_TEXT_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_TEXT");
    break;
  case MSG_TIME_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_TIME");
    break;
  case MSG_BUTTON_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_BUTTON");
    break;
  case MSG_TOOL_BAR_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_TOOL_BAR");
    break;
  case MSG_MDI_WINDOW:
    object = UIW_WINDOW::New("support.dat~WINDOW_MDI");
    break;
  }
  // Add the window object to the window manager.
  if (object)
    *windowManager + object;
}
```

The *object* variable is a **UI_WINDOW_OBJECT** pointer, not a **UIW_WINDOW** pointer. This generic declaration allows us to expand the program to attach other nonwindow objects, for example, an icon.

Now the new window becomes the front window, which processes all subsequent events until the user requests a change. A description of the types of windows presented in this menu item follows:

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

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

Button. Shows standard buttons, radio buttons, check boxes, and bitmapped buttons.

Combo box. Shows two combo box objects, one of which was implemented with string objects, and the other with bitmapped buttons.

Date. Shows the many variations of the date class.

Icon. Shows several types of icons that we can attach either to a parent window or to the screen.

List. Shows a horizontal and vertical list.

Menu. Shows pull-down menus. The source code shows you how to create and attach pull-down and pop-up items into pull-down menus.

Number. This window shows several **UIW_BIGNUM** objects.

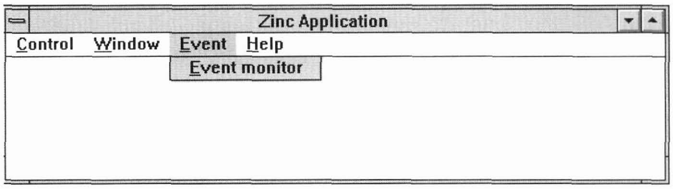
String. This window shows several types of string objects that can be created with Zinc Application Framework. These objects include the basic **UIW_STRING** class, two types of **UIW_FORMATTED_STRING** class objects, and a multi-line text field, **UIW_TEXT**, that only occupies part of its parent window.

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 **ZIL_TIME** class.

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

Event options

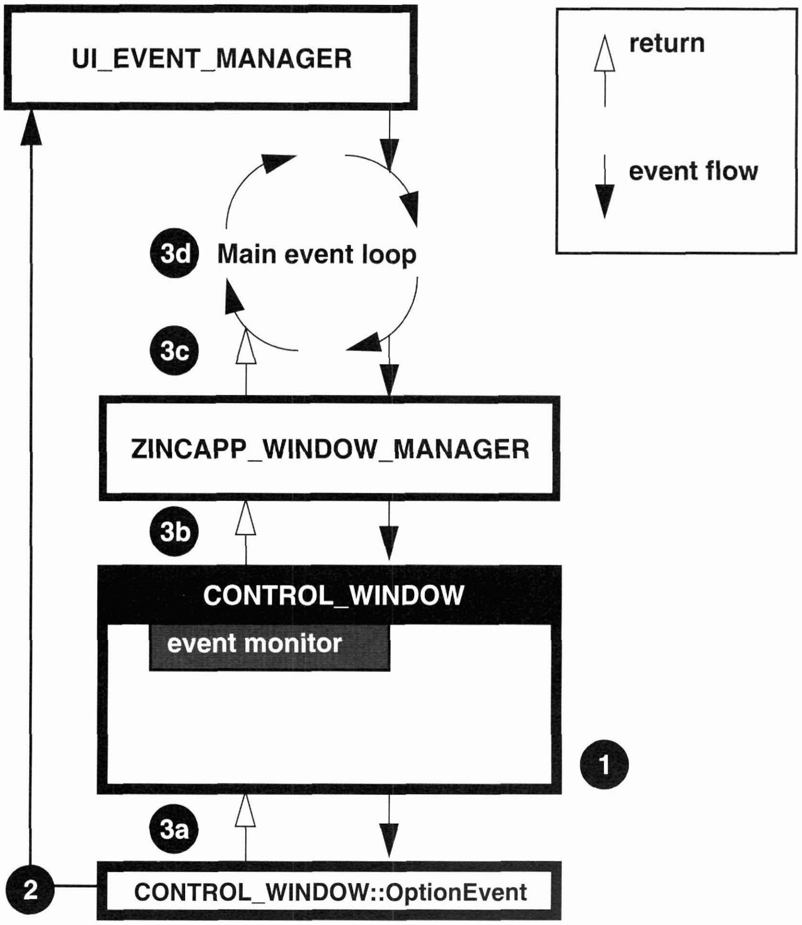


This item contains ZincApp's event options, initialized by the **CONTROL_WINDOW** constructor. Here's that part of the constructor.

```
static UI_ITEM eventItems[] =
{
  { MSG_EVENT_MONITOR,VOIDF(CONTROL_WINDOW::Message),
    "&Event monitor"MNIF_NO_FLAGS },
  { 0, 0, 0 }// end of array
};
...
// Attach the sub-window objects to the control window.
*this
  + new UIW_BORDER
  + new UIW_MAXIMIZE_BUTTON
  + new UIW_MINIMIZE_BUTTON
  + new UIW_SYSTEM_BUTTON(SYF_GENERIC)
  + new UIW_TITLE("Zinc Application")
  + &(*new UIW_PULL_DOWN_MENU
    + new UIW_PULL_DOWN_ITEM("&Control", WNF_NO_FLAGS, controlItems)
    + new UIW_PULL_DOWN_ITEM("&Display", WNF_NO_FLAGS, displayItems)
    + &(*new UIW_PULL_DOWN_ITEM("&Window", WNF_NO_FLAGS)
      + controlItems
      + inputItems
      + selectItems)
    + new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS, eventItems)
    + new UIW_PULL_DOWN_ITEM("&Help", WNF_NO_FLAGS, helpItems));
```

Event program flow

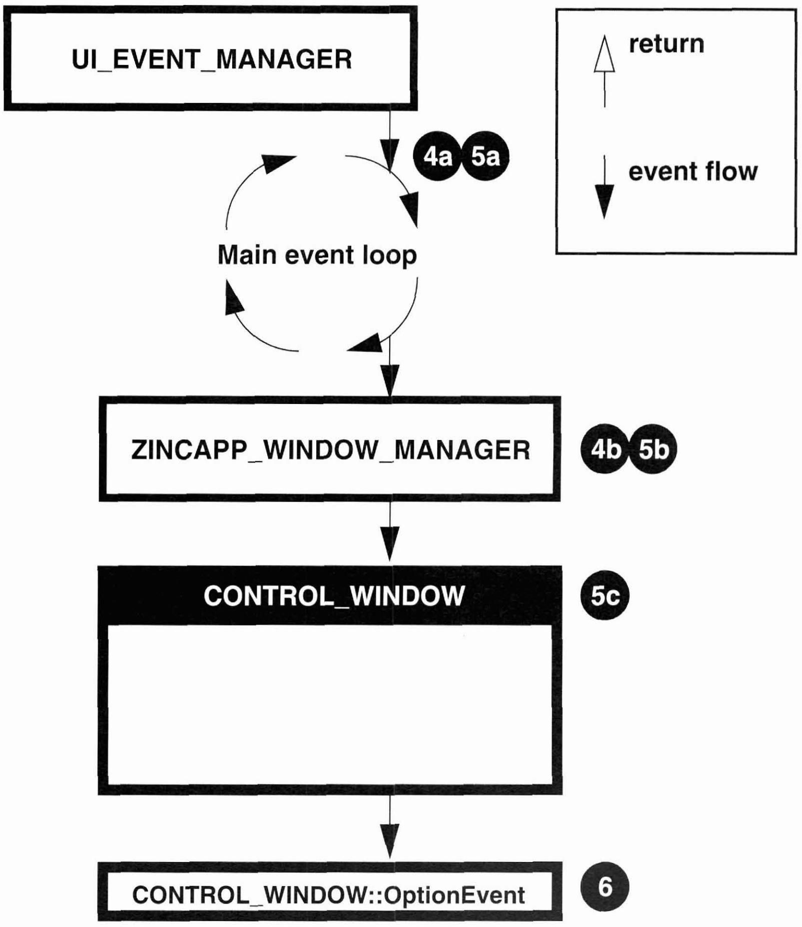
What happens when the user selects the **Event** option? First, the control window executes steps one through four of the general program flow. At the fifth step, however, it calls the **OptionsEvent( )** member function.



1. The **UIW_POP_UP_ITEM::Event( )** function calls the **CONTROL_WINDOW::Message( )** function.
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.
3. Control returns to the main event loop.

4. *eventManager*->**Get( )** gets two messages that the program generates. The first message it gets is *S_CLOSE_TEMPORARY.*
5. The second message received is *MSG_EVENT,* which the main loop passes to the Window Manager, which in turn passes it to **CONTROL_WINDOW::Event( ),** since the control window is the front window on the screen. Then control window evaluates *event.type*—in this case the *MSG_EVENT* message—and calls the **OptionEvent( )** member function.

```
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);
}
```



The **OptionEvent( )** member function creates the event monitor window and attaches it to the window manager. 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, which we encounter in the next section, becomes the front window of the application, and will process all subsequent events directly or indirectly.

Monitoring library events

In order to monitor events, we derived two classes, **EVENT_MONITOR** and **ZINCAPP_WINDOW_MANAGER**.

Event Monitor. The event monitor shows which messages the library is processing. The Windows version of the event monitor window has five sections:

- *Message.* The hex value of the Windows message. We could have implemented a translation table that displayed the message in human-readable form.
- *wParam.* The event's *wParam* value.
- *lParam.* The event's *lParam* value.
- *Position.* The event's *Position* value.
- *Last event.* The interpreted value of the last event. This can be any Zinc event or logical event, or it could be a keyboard or mouse code.

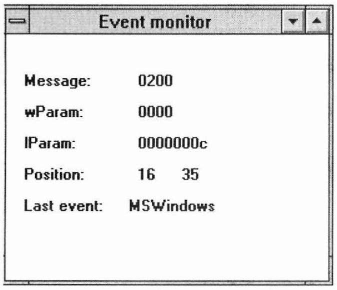
The class **EVENT_MONITOR** contains the implementation of this window, and **ZINCAPP.HPP** contains the definition of **EVENT_MONITOR**. Its members are shown below:

```
class EVENT_MONITOR : public UIW_WINDOW
{
public:
  EVENT_MONITOR(void);
  EVENT_TYPE Event(const UI_EVENT &event);
private:
#if defined(ZIL_MSDOS)
  UIW_PROMPT *keyboard[3];
  UI_EVENT kEvent;
  UIW_PROMPT *mouse[3];
  UI_EVENT mEvent;
#elif defined(ZIL_MSWINDOWS)
  UIW_PROMPT *windowsMessage[5];
  MSG wMsg;
#elif defined(ZIL_OS2)
  UIW_PROMPT *windowsMessage[5];
  QMSG oMsg;
#elif defined(ZIL_MOTIF)
  UIW_PROMPT *motifMessage[3];
  XEvent xEvt;
#elif defined(ZIL_MACINTOSH)
  UIW_PROMPT *macintoshMessage[5];
  EventRecord mEvent;
#endif
  UIW_PROMPT *system;
  UI_EVENT sEvent;
};
```

The event monitor

The **EVENT_MONITOR** derives from the base class **UIW_WINDOW**, therefore inheriting the ability to receive message information, and giving us the ability to remove easily the event monitor window from the screen. When we attach the event monitor window to ZincApp's window manager, it receives all events that pass through the system—after the front window has processed the event, allowing the front window to process the event normally.

If we were to derive the event monitor from **UI_DEVICE** as we did in the **MACRO_HANDLER** tutorial, it would receive only raw input information. By positioning ourselves in the window manager, we are able to see, not only raw events, but how an object interprets raw events. This allows us to see firsthand one of the benefits of Zinc, how Zinc objects handle events in the context of what the object knows how to do.



```
Message:      0200
wParam:       0000
lParam:       0000000c
Position:     16    35
Last event:  MSWindows
```

For example, pressing the mouse button on the title bar produces a series of messages ending in “Move.” Pressing the mouse button in a text field, however, produces the message “Begin mark.” If we had derived **EVENT_MONITOR** from **UI_DEVICE**, we would see only a “mouse down” message.

The **EVENT_MONITOR::Event( )** function can receive two types of events . The first type is messages passed to the window during executioN. 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 messages are sent to the event monitor *after* they have been processed by the window manager. In addition, these special events are packaged by the window manager into a new event, and in turn passed to the member function. The window manager packages these events this way:

- *event.type* is the logical event returned by the receiving object.
- *event.rawCode* is always 0xFFFF if the event has already been passed to the front window. This special value lets us determine whether the original message was intended for the event monitor window (if it is front window on the screen) or whether the event has already been passed through the system.
- *event.data* is the original event that was passed through the system.

EVENT_MONITOR::Event( ) has four parts that check for normal, keyboard, mouse, and logical events, for all the environments ZincApp supports.

1. The first part of **EVENT_MONITOR::Event( )** sets up the event information and determines whether the event window should interpret the event, or whether it should pass the event to **UIW_WINDOW**.

```
EVENT_TYPE EVENT_MONITOR::Event(const UI_EVENT &event)
  {
    ...
    // See if it is a normal event.
    if (event.rawCode != 0xFFFF)
      return (UIW_WINDOW::Event(event));
```

2. In the second part, *keyboard* and *kEvent*, available only in DOS, contain information about the last key that was pressed. *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 program calls **EVENT_MONITOR::Event( )** routine, it changes these variables to reflect the new event, which it passes 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;
      }
```

3. In the third part, *_mouse* and *mEvent*, also available only in DOS, contain information about the last mouse event. They work just like the keyboard variables *keyboard* and *kEvent*, except that they maintain mouse information. For optimization, *mEvent* keeps track of the last event, so that **EVENT_MONITOR::Event( )** will update only those parts of the mouse event that have changed. When the program calls **EVENT_MONITOR::-Event( )**, it passes as an argument the changes in the event. Below is the code that does this:

```
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;
      }
```

4. The fourth part of **EVENT_MONITOR::Event( )** contains variables that keep track of events that the event monitor window receives. The difference between this part and the other parts is that this part can keep track of events for each operating environment Zinc supports, whatever that might be. For example, if the native operating environment is Windows, it keeps track of Windows events; if the native operating environment is Macintosh, it keeps track of Macintosh events; and so forth. Here are those variables:
   - *windowsMessage* and *wMsg* contain the information from the last event that was received by the event monitor in the Windows environment.
   - *windowsMessage* and *oMsg* contain the information from the last event that was received by the event monitor in the OS/2 environment.
   - *motifMessage* and *xEvt* contain the information from the last event that was received by the event monitor in the Motif environment.
   - *macintoshMessage* and *mEvent* contain the information from the last event that was received by the event monitor in the Motif environment.

   For optimization reasons, still other variables, *wMsg*, *oMsg, xEvt,* and *mEvent*, keep track of the last event for optimization so that only those parts of the event that have changed will be updated. When the program calls the **EVENT_MONITOR::Event( )** routine, it changes these variables to reflect the new event, which it passes as an argument to the event monitor's **Event( )** function. Below is the code responsible for this change in Windows:

```
EVENT_TYPE EVENT_MONITOR::Event(const UI_EVENT &event)
  #elif defined(ZIL_MSWINDOWS)
  if (tEvent->type == E_MSWINDOWS)
  {
    MSG msg = tEvent->message;
    char string[32];
    if (wMsg.message != msg.message)
    {
      sprintf(string, "%04x", msg.message);
      windowsMessage[0]->Information(I_SET_TEXT, string);
    }
    if (wMsg.wParam != msg.wParam)
```

```
	{
	  sprintf(string, "%04x", msg.wParam);
	  windowsMessage[1]->Information(I_SET_TEXT, string);
	}
	if (wMsg.lParam != msg.lParam)
	{
	  sprintf(string, "%08x", msg.lParam);
	  windowsMessage[2]->Information(I_SET_TEXT, string);
	}
	if (wMsg.pt.x != msg.pt.x)_
	{
	  sprintf(string, "%d", msg.pt.x);
	  windowsMessage[3]->Information(I_SET_TEXT, string);
	}
	if (wMsg.pt.y != msg.pt.y)
	{
	  sprintf(string, "%d", msg.pt.y);
	  windowsMessage[4]->Information(I_SET_TEXT, string);
	}
	wMsg = msg;
      }
```

5. *_system* and *sEvent* contain information about the last interpreted event that was returned by the window object. These variables work just like the mouse variables *mouse* and *mEvent* except that the information is maintained for the logical or system event. The variable *sEvent* keeps track of the last event for optimization so that only changes in the event cause the event field to be updated. When program calls **EVENT_MONITOR::Event( )**, it changes these variables to reflect the new event, by passing it as an argument to the event monitor's **Event( )** function. Below is a partial list of the event/string pair table:

```
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
  {
    ZIL_LOGICAL_EVENT type;
    char *name;
  } eventTable[] =
  {
    // Raw events.
    { E_MSWINDOWS,"MSWindows" },
    { E_OS2,"OS/2" },
    { E_MOTIF,"Motif" },
    { E_MACINTOSH,"Macintosh" },
    { E_KEY,"Key" },
{ E_MOUSE,"Mouse" },
{ E_CURSOR,"Cursor" },
// System events.
{ S_ERROR,"Error" },
{ S_UNKNOWN,"Unmapped Event" },
{ S_NO_OBJECT,"No object" },
...

// Logical events.
{ L_EXIT,"Exit" },
{ L_VIEW,"View" },
{ L_SELECT,"Select" },
```

The ZincApp window manager

The event monitor window we just described receives all interpreted messages by attaching itself to the Zinc Window Manager class, **ZINCAPP_-WINDOW_MANAGER**. This class is the second part of what makes it possible for us to intercept events without disrupting their normal flow. **ZINCAPP.HPP** contains the definition of the **ZINCAPP-_WINDOW_MANAGER** class, 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);
};
```

Here some information about **ZINCAPP_WINDOW_MANAGER**.

- **UI_WINDOW_MANAGER** is the base class. This allows us to get all interpreted messages before they pass to the main control loop, and to send the event information to the event monitor window, if it exists.
- **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. The ZincApp window man-

ager 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:

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

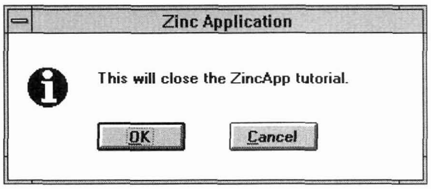
- **Event( )**, which processes the event information, contains two major parts. The first calls **UI_WINDOW_MANAGER::Event( ),** so that it can dispatch the message to the proper window.

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

The second parts sends the interpreted message to the event monitor window, if it exists. It determines if it should by looking at the object's *userFlags*. If **EVENT_MONITOR::Event( )** has set the flag to *MSG_EVENT_MONITOR,* and if the event type is not *S_RESET_DISPLAY,* it modifies the event. When modified, *event.type* contains the logical code, *event.rawCode* contains the value 0xFFFF, and *event.data* points to the raw event. Then the event function sends the message to the device.

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

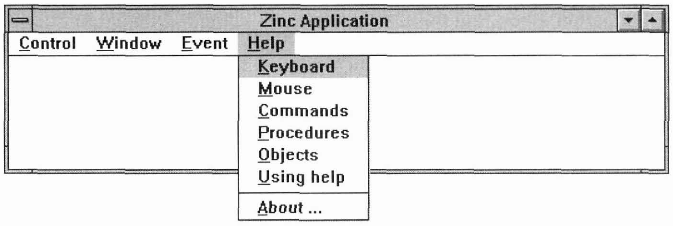
ZINCAPP_WINDOW_MANAGER also provides a way to exit the program through the static member function **ExitFunction( )**, which displays the modal exit window we saw earlier in the chapter.



This will close the ZincApp tutorial.

If the user selects **<u>O</u>K**, an *L_EXIT* message passes through the system, and program stops. Otherwise, the window manager removes the window from the screen, and program flow continues normally.

Help options

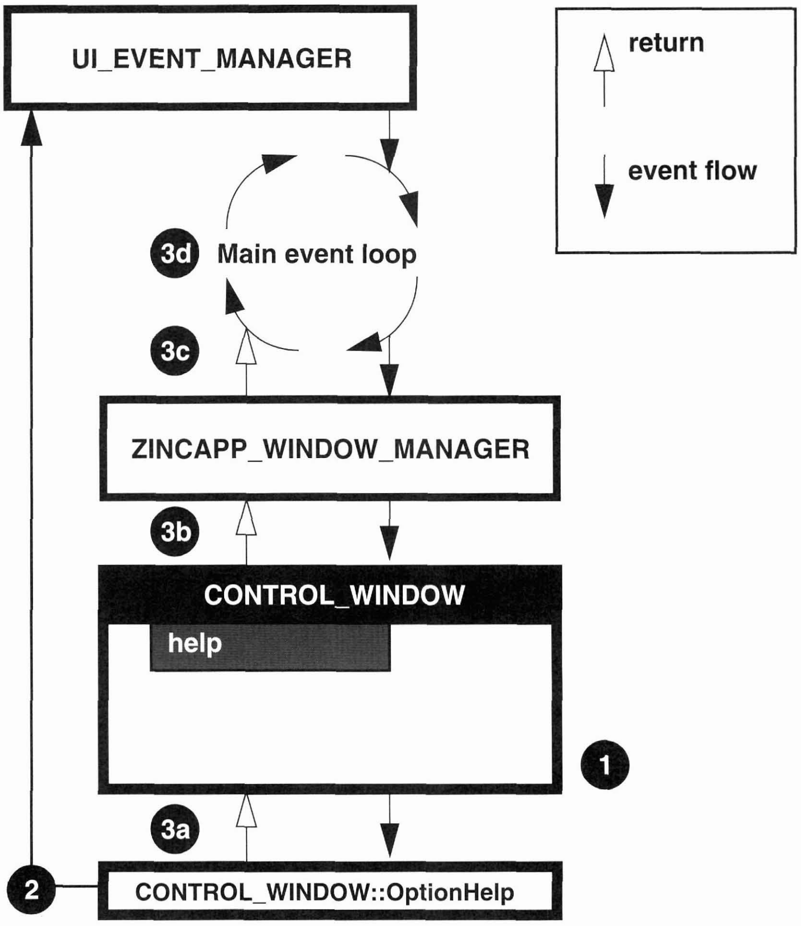


This item contains ZincApp's help options, initialized by the **CONTROL_WINDOW** constructor. Here's that part of the constructor.:

```
    // Help menu items.
static UI_ITEM helpItems[] =
{
    { MSG_HELP_KEYBOARD,ZIL_VOIDF(Message),"&Keyboard",
      MNIF_NO_FLAGS },
    { MSG_HELP_MOUSE,ZIL_VOIDF(Message),"&Mouse",
      MNIF_NO_FLAGS },
    { MSG_HELP_COMMANDS,ZIL_VOIDF(Message),"&Commands",
      MNIF_NO_FLAGS },
    { MSG_HELP_PROCEDURES,ZIL_VOIDF(Message),"&Procedures",
      MNIF_NO_FLAGS },
    { MSG_HELP_OBJECTS,ZIL_VOIDF(Message),"&Objects",
        MNIF_NO_FLAGS },
      { MSG_HELP_HELP,ZIL_VOIDF(Message),"&Using help",
        MNIF_NO_FLAGS },
      { 0, ZIL_VOIDF(0),"",MNIF_SEPARATOR },
      { MSG_HELP_ZINCAPP,ZIL_VOIDF(About),"&About ...",
        MNIF_NO_FLAGS },
      { 0, 0, 0, 0 }// End of array.
    };
    ...
    // Attach the menu and support 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_POP_UP_ITEM("About ZincApp...", MNIF_NO_FLAGS,
          BTF_NO_TOGGLE | BTF_NO_3D,
          WOF_SUPPORT_OBJECT, About, MSG_HELP_ZINCAPP))
      + new UIW_TITLE("Zinc Application")
      + &(*new UIW_PULL_DOWN_MENU
        + new UIW_PULL_DOWN_ITEM("&Control", WNF_NO_FLAGS,
          controlItems)
  #if defined(ZIL_MSDOS)
        + new UIW_PULL_DOWN_ITEM("&Display", WNF_NO_FLAGS,
          displayItems)
  #endif
        + &(*new UIW_PULL_DOWN_ITEM("&Window", WNF_NO_FLAGS)
          + controlObjects
          + inputObjects
          + selectObjects)
        + new UIW_PULL_DOWN_ITEM("&Event", WNF_NO_FLAGS,
          eventItems)
        + new UIW_PULL_DOWN_ITEM("&Help", WNF_NO_FLAGS,
          helpItems))
      + new UIW_ICON(0, 0, "minIcon", "Zincapp",
          ICF_MINIMIZE_OBJECT);
  }
```

Help program flow

What happens when the user selects the **Help** option? First, the control window executes steps one through four of the general program flow. At the fifth step, however, it calls the **OptionHelp( )** member function.

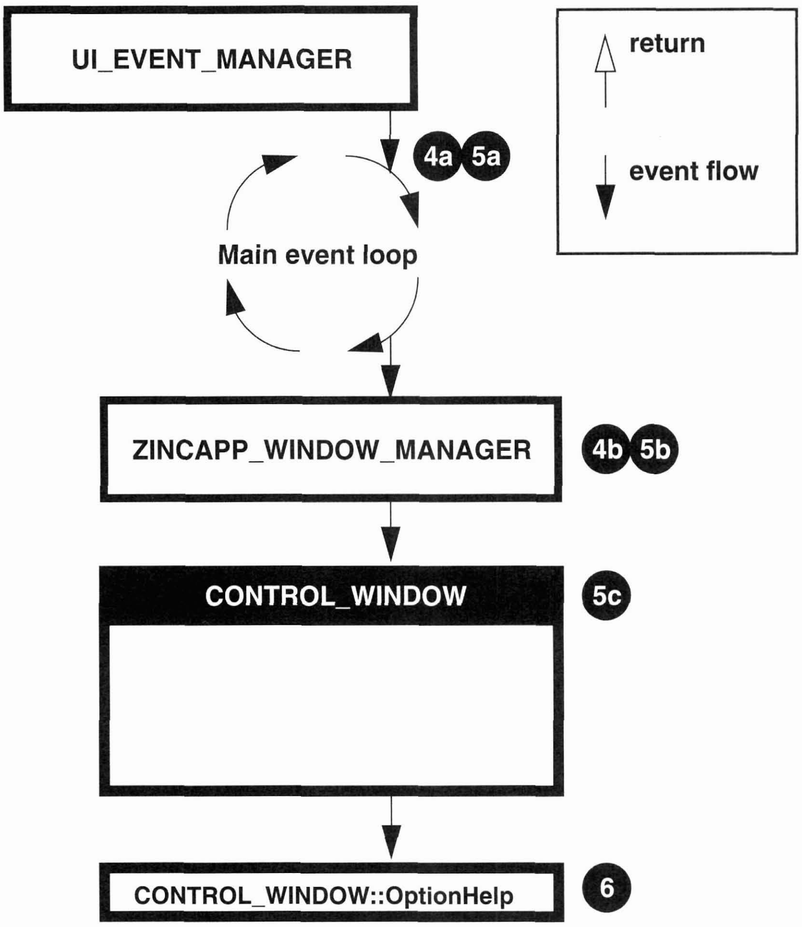


1. The **UIW_POP_UP_ITEM::Event( )** function calls the **CONTROL_WINDOW::Message( )** function.
2. The **CONTROL_WINDOW::Message( )** function sends a request to remove the temporary display options menu by sending an *S_CLOSE_TEMPORARY* message through the system.
3. Control returns to the main event loop.

4. *eventManager->***Get( )** gets two messages that the program generates. The first message it gets is *S_CLOSE_TEMPORARY.*

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—which results 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);
}
```

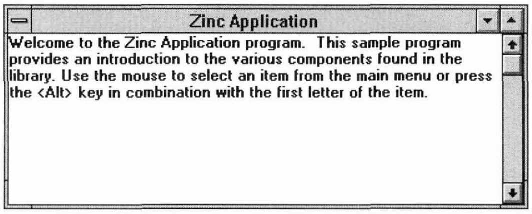


6. The **OptionHelp( )** member function evaluates the item's value (passed down through the *item* argument) to determine which type of help context has been requested. It then sends the help request to the help system by calling **DisplayHelp( )**. The following code shows how this is done:

```
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 **DisplayHelp( )** is called, it attaches the help window to the Window Manager. For example, the help request *MSG_HELP_ZINCAPP* brings up a help window:



Welcome to the Zinc Application program. This sample program provides an introduction to the various components found in the library. Use the mouse to select an item from the main menu or press the <Alt> key in combination with the first letter of the item.

Here the help window becomes the front window of the application, and processes events until the user requests a new window.

The help window is a normal, not modal, window, and so the user can select other windows while the help window is up. In addition, Zinc defines only one help window for an application. If the help window is already present, or if it has been moved and sized by a previous help request, Zinc presents the window 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, the user can access context sensitive help by pressing a help key during the application. Each ZincApp window has a predefined help context, specified when the window is constructed. For example, the help context of the main control window is *HELP_MAIN_CONTROL*. The code below shows how:

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

Generally, **UI_WINDOW_OBJECT::Event( )** also provides access to the help system in the same way. After the user presses the <F1> key, the Window Manager sends the message to the front window. If the window has a help context, the Window Manager calls the help system with that help context. If the user presses the <F1> key when the control window is active, the Window Manager would request the *HELP_MAIN_CONTROL* help context. Otherwise, the Window Manager can request general help by sending *NO_-HELP_CONTEXT* to the *helpSystem*->**DisplayHelp( )** function. The help system receives this message and replaces it with the general help specified at the time when the help system was constructed. In ZincApp, general help context is *HELP_GENERAL*.

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

Structured programming—in a word, don't

Some Zinc programmers use structured programming techniques. If we rewrote ZincApp using those techniques, we would assign each menu item a function, which the program would execute when the user selected an item. This is a cumbersome and inefficient technique for writing programs in a event-driven framework, for reasons we will learn in a moment.

In order to demonstrate that structured programming in an event-driven environment has serious drawbacks, let's hypothetically revise ZincApp. We could rewrite the **Help** options in the **CONTROL_WINDOW** constructor in a structured manner, so that each option would call specific help functions, rather than pass an event to an object that contained a help member function. Remember, ZincApp contains none of this code—this is merely a conceptual alternative, designed to demonstrate a concept.

```
CONTROL_WINDOW::CONTROL_WINDOW(void) :
  UIW_WINDOW(0, 0, 52, 13, WOF_NO_FLAGS, WOAF_LOCKED)
{
  extern EVENT_TYPE HelpKeyboard(UI_WINDOW_OBJECT *item, UI_EVENT &event,
    EVENT_TYPE ccode);
  extern EVENT_TYPE HelpMouse(UI_WINDOW_OBJECT *item, UI_EVENT &event,
    EVENT_TYPE ccode);
  extern EVENT_TYPE HelpCommands(UI_WINDOW_OBJECT *item, UI_EVENT &event,
    EVENT_TYPE ccode);
  extern EVENT_TYPE HelpProcedures(UI_WINDOW_OBJECT *item,
    UI_EVENT &event, EVENT_TYPE ccode);
  extern EVENT_TYPE HelpHelp(UI_WINDOW_OBJECT *item, UI_EVENT &event,
    EVENT_TYPE ccode);
  extern EVENT_TYPE HelpZincApp(UI_WINDOW_OBJECT *item, UI_EVENT &event,
    EVENT_TYPE ccode);
  static UI_ITEM helpItems[] =
  {
    { MSG_HELP_KEYBOARD, VOIDF(CONTROL_WINDOW::Message),
        "&Keyboard", MNIF_NO_FLAGS },
        { MSG_HELP_MOUSE, VOIDF(CONTROL_WINDOW::Message),
          "&Mouse", MNIF_NO_FLAGS },
    { MSG_HELP_COMMANDS, VOIDF(CONTROL_WINDOW::Message),
        "&Commands", MNIF_NO_FLAGS },
    { MSG_HELP_PROCEDURES, VOIDF(CONTROL_WINDOW::Message),
      "&Procedures", MNIF_NO_FLAGS },
    { MSG_HELP_OBJECTS, VOIDF(CONTROL_WINDOW::Message),
      "&Objects", MNIF_NO_FLAGS },
    { MSG_HELP_HELP, VOIDF(CONTROL_WINDOW::Message),
      "&Using help", MNIF_NO_FLAGS },
    { 0, VOIDF(0), "", MNIF_SEPARATOR },
    { MSG_HELP_ZINCAPP, VOIDF(About), "&About ...",
      MNIF_NO_FLAGS }, { 0, 0, 0, 0 }// End of array.
  };
  ...
}
```

In our hypothetical revision, each menu item would have a function that performed a particular operation. To do this, we would define functions for each of the menu items specified in the main control window. Here's an example of how we could write the **HelpKeyboard( )** function.

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

While our hypothetical revision works, it has serious drawbacks.

1. Using structured programming techniques results in inefficiency. In the help example, it took seven functions to do the work that the **CONTROL_WINDOW::OptionHelp( )** function does in one. This wastes compiler time and executable space, making our applications perform more slowly.
2. Using structured programming techniques in an event-driven architecture results in confusing code. Since event-driven architecture works best with object-oriented programming techniques, we should stick to writing object-oriented programs.
3. Using structured programming techniques makes us duplicate much of what Zinc has already accomplished. Since Zinc has an extensive library of objects and event-handling routines, embedding functions like we've done negates the advantages of object-oriented structure, among which are elegant design and smaller code size. Using structured techniques increases the amount of time and effort involved in creating and debugging programs.

Because of these reasons, Zinc recommends that we eschew structured programming techniques in writing our programs.

Conclusion

We've reached the end of *Getting Started with Zinc Programming*. We now know enough about Zinc to begin writing complex applications that run on nearly every operating environment in the world—all with one source code file, using objects native to each of those operating environments. Zinc is sure you'll enjoy using the Application Framework—after all, we have as much flexibility as possible, full use of the advanced features of C++, and can use in our interfaces any modern language used anywhere in the world today.

Have fun!

Appendix A

Compiler Considerations

This appendix describes how to compile your applications with Zinc Application Framework.

When building your applications, we recommend using the same switch settings that were used to compile the Zinc Application Framework libraries. These settings are found in the appropriate library makefiles in the **ZINC\SOURCE** directory.

Here is a complete list of all libraries and what they contain. Libraries for a particular compiler are located in the **ZINC\LIB\** compiler directory (except for Motif, Macintosh, and NEXTSTEP).

- **DOS_ZIL.LIB**. Real-mode DOS library.
- **D16_ZIL.LIB**. 16-bit DOS library.
- **D32_ZIL.LIB**. 32-bit DOS library.
- **DOS_GFX.LIB**. DOS real-mode **UI_GRAPHICS_DISPLAY**.
- **D16_GFX.LIB**. DOS **UI_GRAPHICS_DISPLAY** for 16-bit DOS extender.
- **D32_GFX.LIB**. DOS **UI_GRAPHICS_DISPLAY** for 32-bit DOS extender.
- **WIN_ZIL.LIB**. MS Windows library.
- **WNT_ZIL.LIB**. MS Windows NT library.
- **OS2_ZIL.LIB**. IBM OS/2 library.
- **BC_LGFX.LIB**. Borland-specific GFX graphics library.
- **BC_16GFX.LIB**. Borland-specific GFX graphics library for 16-bit DOS extender.
- **DOS_BGI.LIB**. DOS **UI_BGI_DISPLAY**.
- **DOS_ZILO.LIB**. Borland DOS overlay library.
- **MS_LGFX.LIB**. Microsoft-specific GFX graphics library.
- **MS_16GFX.LIB**. Microsoft-specific GFX graphics library for 16-bit DOS extender.
- **MS_32GFX.LI**B. Microsoft-specific GFX graphics library for 32-bit DOS extender.
- **DOS_MSC.LIB**. DOS **UI_MSC_DISPLAY**.
- **D16_MSC.LIB**. DOS **UI_MSC_DISPLAY** for 16-bit DOS extender.
- **D32_MSC.LIB**. DOS **UI_MSC_DISPLAY** for 32-bit DOS extender.
- **SC_LGFX.LIB**. Symantec-specific GFX graphics library.
- **SC_16GFX.LIB**. Symantec-specific GFX graphics library for 16-bit DOS extender.
- **SC_32GFX.LIB**. Symantec-specific GFX graphics library for 32-bit DOS extender.
- **SC_LGFXV.LIB**. Symantec-specific DOS overlay GFX graphics

library.

- **DOS_ZILV.LIB**. Symantec DOS overlay library.
- **DOS_GFXV.LIB**. Symantec DOS overlay **UI_GRAPHICS_DISPLAY**.
- **D32_WCC.LIB**. DOS **UI_WCC_DISPLAY** for 32-bit DOS extender.
- **WC_32GFX.LIB**. Watcom-specific GFX graphics library for 32-bit DOS extender.
- **lib_mtf_zil.a**. OSF/Motif library.
- **lib_crs_zil.a**. Curses library.
- **lib_nxt_zil.a**. NEXTSTEP library.

Borland

This section describes how to use Borland compilers with Zinc. For more complete details on the Borland compilers, see your Borland *User's Guide*.

Makefiles—DOS, Windows, OS/2

When building applications using a makefile, your **TURBOC.CFG** and **TLINK.CFG** files must be set to include paths to both the Borland and the Zinc libraries and include files. A typical **TURBOC.CFG** file might look like this:

```
-I.;C:\ZINC\INCLUDE;C:\BORLANDC\INCLUDE
-L.;C:\ZINC\LIB\BTCPP400;C:\BORLANDC\LIB
```

A typical **TLINK.CFG** might look like this:

```
-L.;C:\ZINC\LIB\BTCPP400;C:\BORLANDC\LIB
```

Any of the example or tutorial makefiles can be used as a skeleton for creating your own makefiles. It is important that the switches used to compile the Zinc libraries be used when compiling your applications. Of particular importance are the **-x** and -**RT** switches. These control the enabling of exception handling and the enabling of run-time type checking, respectively. The Zinc libraries are compiled with these options turned off. If any modules in your application, including the Zinc libraries, don't match the other modules in your application with regard to these options, or your application will likely crash.

Borland 4.0 IDE—DOS, Windows

To compile DOS or Windows applications in the IDE, do the following:

1. Select **Project | New** project.
2. Enter the project directory and name.
3. Choose the target platform.
4. Choose the large memory model.
5. Make sure the **Runtime** library is selected. If you're building a DOS application using BGI, select the BGI library, as well. Zinc does not require the other libraries.
6. If you're building a Windows application, select the **Static** option.
7. Select **Options | Project**.
8. Select the **Directories** topic. Enter the directories for the Borland and Zinc include and library directories.
9. Select the **C++ Options** topic and open the Exception handling/RTTI sub-topic. Turn off the **Enable exceptions** option and the **Enable run-time type information** option.
10. Place the necessary source and library files in the project.
11. Select **Project | Build** all.

Borland 1.5 IDE—OS/2

To compile an OS/2 application in the IDE, do the following:

1. Select **Project | New** project.
2. Enter the project directory and name.
3. Select **Project | View** Settings.
4. Select the **Directories** page. Enter the directories for the Borland and Zinc include and library directories.
5. Select **Project | Add** item.
6. Place the necessary source and library files in the project.
7. Select **Compile | Make**.

Microsoft

This section describes how to use Microsoft compilers with Zinc. For more complete details on the Microsoft compilers, see your Microsoft *User's Guides*.

Makefiles—DOS, Windows

When building applications using a makefile, your *LIB* and *INCLUDE* environment variables must be set to include paths to both the Microsoft and the Zinc libraries and include files. A typical *LIB* environment variable might look like this:

```
LIB=.;C:\ZINC\LIB\MVCPP150;C:\VISUALC\LIB
```

A typical *INCLUDE* environment variable might look like this:

```
INCLUDE=.;C:\ZINC\LIB\MVCPP150;C:\VISUALC\INCLUDE
```

The easiest way to set theses environment variables is in your **AUTOEXEC.BAT** file.

Any of the example or tutorial makefiles can be used as a skeleton for creating your own makefiles. It is important that the switches used to compile the Zinc libraries be used when compiling your applications.

Visual Workbench—DOS, Windows

To compile DOS or Windows applications in the Visual Workbench, do the following:

1. Select **Project | New**.
2. Enter the executable name.
3. Set the **Project Type**.
4. Add files to the project.
5. Select **Options | Project**
6. Choose the **Compiler** button
7. Under **Code Generation**, set **CPU** to **8086/8088**.
8. Under **Memory Model**, set **Model** to **Large**.
9. Choose the **Linker** button.
10. Under **Input**, turn on **Prevent Use of Extended Dictionary**.
11. Under **Memory Image**, set **Max. Number of Segments** to **256**.
12. If compiling a DOS application, under Input, add *graphics* to **Libraries**.
13. If compiling a Windows application and the **.DEF** file has a *STACK* entry, then under Memory Image remove the entry in **Stack Size**.

Due to the way Microsoft handles dependencies in the Visual Workbench, it may be necessary to comment out some #include directives in the Zinc header files even though these lines should be pre-compiled out. If you get errors that the compiler cannot open some NEXTSTEP or OSF/Motif header files, then comment out the offending line(s).

Symantec

This section describes how to use Symantec compilers with Zinc. For more complete details on the Symantec compilers, see your Symantec *Compiler and Tools Guide*.

Makefiles—DOS, Windows

When building applications using a makefile, your *LIB* and *INCLUDE* environment variables must be set to include paths to both the Symantec and the Zinc libraries and include files. A typical LIB environment variable might look like this:

```
LIB=.;C:\ZINC\LIB\SCCPP610;C:\SC\LIB
```

A typical *INCLUDE* environment variable might look like this:

```
INCLUDE=.;C:\ZINC\LIB\SCCPP610;C:\SC\INCLUDE
```

The easiest way to set theses environment variables is in your **AUTOEXEC.BAT** file.

Any of the example or tutorial makefiles can be used as a skeleton for creating your own makefiles. It is important that the switches used to compile the Zinc libraries be used when compiling your applications.

Symantec 6.1 IDDE—DOS, Windows

To compile DOS or Windows applications in the IDDE, do the following:

1. Select **Project | New**
2. Place the necessary source and library files in the directory.
3. Select **Options | Project**
4. Select the target platform.
5. Select **Options | Compiler | Memory Model**
6. Select **Large**.
7. Select **Options | Directories**.
8. Enter the directories for the Symantec and Zinc include and library directories.

Watcom

This section describes how to use Watcom compilers with Zinc. For more complete details on the Watcom compilers, see your Watcom *C/C++ User's Guide*.

Makefiles—DOS, Windows, OS/2

When building applications using a makefile, your *LIB* and *INCLUDE* environment variables must be set to include paths to both the Watcom and the Zinc libraries and include files. A typical *LIB* environment variable might look like this:

```
LIB=.;C:\ZINC\LIB\WCCPP;C:\WATCOM\LIB386;C:\WATCOM\LIB386\WIN
```

A typical *INCLUDE* environment variable might look like this:

```
INCLUDE=.;C:\ZINC\LIB\WCCPP;C:\WATCOM\H;C:\WATCOM\H\WIN
```

The easiest way to set these environment variables is in your **AUTOEXEC.BAT** file.

Any of the example or tutorial makefiles can be used as a skeleton for creating your own makefiles. It is important that you use the switches for compiling the Zinc libraries when compiling your applications.

Watcom 10.0 IDE—DOS, Windows, OS/2

To compile DOS, Windows, or OS/2 applications in the IDE, do the following:

1. Select **Project** | **New** project.
2. Enter the project directory and name.
3. Select **Open**.
4. Choose the target platform from the dialog.
5. Select **Sources** | **New** source. Place the necessary source and library files in the project.
6. Select **Options** | **C++** compiler switches.
7. Select the **File option** switches. Enter the directories for the Watcom and Zinc include and library directories.
8. Select **Targets** | **Make**.

IBM

This section describes how to use IBM compilers with Zinc. For more complete details on the IBM compilers, see your IBM manuals.

Makefiles—OS/2

When building applications using a makefile, your *LIB* and *INCLUDE* environment variables must be set to include paths to both the IBM and the Zinc libraries and include files. A typical *LIB* environment variable might look like this:

```
LIB=.;C:\ZINC\LIB\SCCPP610;C:\IBM\LIB;C:\TOOLKT21\OS2LIB
```

A typical *INCLUDE* environment variable might look like this:

```
INCLUDE=.;C:\ZINC\LIB\SCCPP610;C:\IBM\INCLUDE;C:\TOOLKT21\OS2H
```

The easiest way to set theses environment variables is in your **AUTOEXEC.BAT** file.

Any of the example or tutorial makefiles can be used as a skeleton for creating your own makefiles. It is important that the switches used to compile the Zinc libraries be used when compiling your applications.

WorkFrame/2

For details on using IBM's WorkFrame/2 development environment, see the **READ.ME** file or your IBM manuals.

Macintosh

These Symantec projects are for compiling Zinc applications on Macintosh:

- **Mac_ZIL1**
- **Mac_ZIL2**
- **Mac_ZIL3**
- **Mac_ZIL4**
- **Mac_ZIL5**
- **Mac_ZIL6**
- **Mac_ZIL7**
- **Mac_ZIL8**
- **Mac_ZIL9**
- **Mac_ZIL10**
- **UI_JumpTables**
- **UI_Application**
- **ZIL_Storage**
- **Mac_ZIL.rsrc**

Apple's universal headers must be used with Zinc applications. These headers are included in the Universal Headers folder in the **THINK** tree. Consult your Symantec documentation for instructions.

THINK Project Manager (TPM)

Each tutorial program has a sample project file that may be used as a template for other programs. Before using the project file, you must make aliases of the SCCPP700 **Include** folder and the SCCPP700 **Library** folder. These aliases must be placed in the **Alias** folder within the TPM tree. Consult your Symantec *User's Guide* for help.

To compile applications for Macintosh with TPM, do the following:

1. Select **Edit | Options | THINK Project Manager**.

2. Select **Preferences**.
3. Choose **Optimize monomorphic methods**.
4. Select **Extensions**.
5. Make the following entries in the table.

```
.cpp => Symantec C++
.rsrc => Resource Copier
```

6. Select **Edit | Options | Symantec C++**.
7. Select **Language Settings**.
8. Choose **Relaxed ANSI conformance**.
9. Choose **Read each header file once**.
10. Select **Compiler Settings**.
11. Choose **Align to 2 byte boundary**.
12. Select **Debugging**.
13. Choose **Use function calls for inlines**.
14. Select **Prefix**.
15. Type the following line.

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

Each of the Mac_ZIL* projects must be added to its own segment within your project, since the Macintosh limits code segments to 32K. You must also include within your project the following Symantec projects:

- **CPlusLib**
- **MacTraps**
- **ANSI++**
- **unix++.**

Motif

After installing Zinc on your system, run the INSTALL utility. It will ask you a number of questions concerning your system's configuration and will then configure your Zinc installation for your environment. Once this is complete all your Zinc makefiles will be ready for use in your environment. See the **MOTIF.TXT** file if you need more details on compiling Zinc for Motif.

Curses

After installing Zinc on your system, run the INSTALL utility. It will ask you a number of questions concerning your system's configuration and will then configure your Zinc installation for your environment. Once this is complete all your Zinc makefiles will be ready for use in your environment. See the **CURSES.TXT** file if you need more details on compiling Zinc for Curses.

NEXTSTEP

Zinc provides Unix makefiles for all tutorial and example programs. These can be run from a terminal window to build an application. Use one of these makefiles as a template for your own makefiles. See the **NEXTSTEP.TXT** file for information on using **ProjectBuilder.app** to build your applications.

Appendix B Example Programs

This appendix lists Zinc's example programs, and explains what each program does as well as its design principles.

Callbacks

VALIDT

Creates two windows with number fields. If the user inputs a number outside the range 0-999, the program alerts the user that the number is out of range.

Concepts.

- Using nonstatic methods.
- Using range-checking to validate with **UIW_BIGNUM**.
- Defining a user function to validate with **UIW_BIGNUM**.
- Deriving a window.

Design principles. The **VALIDATE** class uses the nonstatic **MemberValidate( )** method to validate its **UIW_BIGNUM** fields. To accomplish this, **VALIDATE** defines the **Validation( )** method, which is static, to call the **MemberValidate( )** method. When constructing a **UIW_BIGNUM** object, **VALIDATE** passes the **Validation( )** method to the **UIW_BIGNUM** constructor, since it requires a static method.

The **Input A** field is a **UIW_BIGNUM** that uses range-checking for validation. When the **VALIDATE** constructor constructs this field, it passes the range "0..999" to the constructor to allow normal **UIW_BIGNUM** range validation. Notice that a user function is not passed to the constructor.

The **Input B** and **Input C** fields are **UIW_BIGNUM** objects that call a user function to perform validation. When the **VALIDATE** constructor constructs these fields, it passes the **Validation( )** method to the constructor. These **UIW_BIGNUM** objects will perform validation by calling the **Validation** method. Notice that a range is not passed to the constructor. **VALIDATE** derives from **UIW_WINDOW**.

The **VALIDATE** window does not define an **Event( )** method, so the base **UIW_WINDOW** class handles all events. This way, **VALIDATE** retains all normal window functionality, while defining other private methods and members to provide additional functionality.

DrawItem

ANALOG

What it does. **ANALOG** creates an analog clock with the current date at the bottom. The operating system notifies the clock when each second has passed through a timer device.

Concepts.

- Using **UID_TIMER** with *TMR_QUEUE_EVENTS*.
- Using *I_INCREMENT_VALUE* with a **UIW_DATE** field.
- Using **DataSet( )** with a **UIW_TIME** field.
- Using the *WOS_OWNERDRAW* status flag.
- Drawing an "owner-draw" object. Using display primitives with XOR.

Design principles. The **UID_TIMER** device in **ANALOG** uses the *TMR_QUEUE_EVENTS* flag, which causes **UID_TIMER** to post an *E_TIMER* event on the event manager's queue every second. Since the **UID_TIMER** posts the *E_TIMER* event on the event manager's queue, only the current window on the window manager receives the *E_TIMER* event. **ANALOG** only has one window, so this method is sufficient. **UID_TIMER** may also notify an object directly with an *E_TIMER* event.

When the **CLOCK** window receives an *E_TIMER* event at midnight, it notifies the **UIW_DATE** field at the bottom of the window by calling **UIW_DATE::Information( )** with *I_INCREMENT_VALUE*. The **UIW-_DATE** field therefore increments itself by one day, changing its day, month, or year as it needs to.

When **ANALOG** runs in text mode, the analog clock is replaced by a **UIW_TIME** field that updates every second. To accomplish the update, the **CLOCK** window calls the **UIW_TIME** field's **DataSet( )** with the new system time every time the **CLOCK** window receives an **E_TIMER** event. The **CLOCK** window constructs a **ZIL_TIME** object and passes it to the **UIW_TIME** field's **DataSet( )** to achieve the update.

The **CLOCK** class derives from **UIW_WINDOW**, and its window passes most of the events it receives down to the **UIW_WINDOW** class. The **CLOCK** window's event method handles only the **E_TIMER** event. This way, the **CLOCK** class retains all normal window functionality and responds to the *E_TIMER* event by updating its children as appropriate.

The **ANALOG_FACE** class is derived from **UI_WINDOW_OBJECT** so that the **CLOCK** may consider it an updateable child object. The **ANALOG_FACE** passes most of the events it receives down to the **UI_WINDOW_OBJECT** class. The **ANALOG_FACE** event method handles only the *S_CREATE*, *S_CHANGED*, and *S_MOVE* events. This way, the **ANALOG_FACE** class retains all normal window object functionality and responds to the *S_CREATE*, *S_CHANGED*, and *S_MOVE* events by recalculating the center of the analog face.

The **ANALOG_FACE** class has private members to store the center of the analog face and the positions of the clock hands for updating the display. The **ANALOG_FACE** class also defines a **DrawItem( )** to display the analog face on the **CLOCK** window. The **ANALOG_FACE** sets its *WOS_OWNERDRAW* status flag in the event method when it receives an

S_CREATE, *S_CHANGED*, or *S_MOVE* event so that the **UI_WINDOW_OBJECT** base class will call **ANALOG_FACE::DrawItem( )** to update the analog face.

The "owner-draw" **ANALOG_FACE** DrawItem method draws the **ANALOG_FACE** object by getting the system time from the parent **CLOCK** object's time field, which is a **ZIL_UTIME** object. Then the DrawItem method virtualizes the display by calling **VirtualGet( )**, followed by calls to the **Face( )**, **UpdateMinutes( )**, and **UpdateSeconds( )** methods. The **Face( )** method draws the background of the analog face. The **UpdateMinutes( )** method updates the hour and minute hands. The **UpdateSeconds( )** method updates the second hand.

ANALOG_FACE uses of the XOR mode of the display primitives in the **UpdateMinutes( )** and **UpdateSeconds( )** methods when drawing the hands on the analog face. The display's **Polygon** method draws the hour and minute hands in XOR mode, and the display's Line method draws the second hand in XOR mode. When the **ANALOG_FACE** updates the clock hands, it draws the old hands in XOR mode to erase them, then redraws the new hands in XOR mode.

GRID

Uses the **DrawItem( )** function to display a tic-tac-toe grid. It shows how to use palettes for drawing and how to reference graphical drawing so the drawn object stays centered.

Concepts.

- **DrawItem( )** function
- Registering an object with a native operating environment
- *WOS_OWNERDRAW* flag
- **VirtualGet( )**, **VirtualPut( )**
- *lastPalette*
- Clipping region parameter of display functions
- Using palettes for drawing

Design principles. **GRID** contains a derived class, **DRAW_OBJECT**, that derives from **UI_WINDOW_OBJECT**. **DRAW_OBJECT**'s **DrawItem( )** function overrides the base class's **DrawItem( )** function to draw a tic-tac-toe grid. **DRAW_OBJECT**'s **Event( )** function handles resizing and registers itself with the native operating environment.

GRAPH

Uses the **DrawItem( )** function to display four windows, each with a different type of chart. The first window draws a line chart. The second window draws a sizeable pie chart. The third window draws a nonsizeable pie chart. The fourth window draws a bar chart.

Concepts.

- **DrawItem( )** function
- *WOS_OWNERDRAW* flag.
- **VirtualGet( )**, **VirtualPut( )**

Design principles. **GRAPH** draws directly on the client area of the screen. **LINE_CHART**, **BAR_CHART**, and **PIE_CHART** derive from **UIW_WINDOW**. Each of their **DrawItem( )** functions draw the graphs.

DSPLAY

Uses the drawing primitives to draw a picture with several objects and bitmaps.

Concepts.

- Creating palettes.
- Using the graphics primitives.
- The **DrawItem( )** function.

Design principles. **DISPLAY_WINDOW** is a derived window that is added as a child window. Its virtual DrawItem function calls the drawing functions to create the image. These drawing functions make use of the drawing primitives provided in Zinc's libraries to create the actual image.

A **UI_PALETTE** is defined for each of the drawing functions except **DrawBitmap( )**. The bitmap is defined in the code. The **UI_PALETTE** has ten fields of information. The first three are used for text mode and the last seven are for graphics mode and grey scale monitors.

DrawRectangle( ) draws five rectangles, one in each corner and one in the middle of the window. **DrawEllipse( )** draws four arcs that can be seen at each corner of the window. **DrawPolygon( )** draws the triangular pieces at each corner. **DrawBitmap( )** draws smiley faces, one at each corner and one in the middle. **DrawAlphabet( )** draws the letters of the alphabet in four places in the window. **DrawLines( )** draws the remaining lines that are seen in the window.

LSTITM

Puts a list of derived objects on a windows. The programmer is responsible for the drawing of the objects. The items in the list have multiple entries and lines them up in columns.

Concepts demonstrated.

- Deriving an object from **UIW_BUTTON**
- WOS_OWNERDRAW
- Providing a **DrawItem( )** function for a derived class
- Using drawing primitives from a derived **UI_DISPLAY** class
- Sorting by using a **Compare( )** function

Design principles. To draw an object by hand we must derive a class. For this example we will derive from **UIW_BUTTON**. We must give the class a constructor. The constructor must call the base class's constructor. Inside of the constructor we initialize any member data and since we want to do the drawing we will also set the *WOS_OWNERDRAW* flag. We also declare a destructor to handle clean-up of the initialized member data.

Since the object is *WOS_OWNERDRAW* we will also provide a **DrawItem( )** function. A **DrawItem( )** function takes two parameters, a the contents of the memory address of *UI_EVENT*, and a *EVENT_TYPE*, and returns an *EVENT_TYPE*. In the **DrawItem( )** function we use *LogicalPalette* to determine what colors we should use for painting. Before we do any painting we must call *display*->**VirtualGet( )**. This sets up the regions and notifies the environment we are going to begin painting. Next we can paint by calling the basic display primitives—**Bitmap( )**, **Ellipse( )**, **Line( )**, **Polygon( )**, **Rectangle( )**, and **Text( )**; as well as **UI_WINDOW_OBJECT** drawing functions—**DrawBorder( )**, **DrawShadow( )**, **DrawText( )**, and **DrawFocus( )**.. We will paint the background first then paint several entries at fixed distances so that the columns will line up. When painting is done we must call *display*->**VirtualPut( )** to notify the environment we are done painting. Last, we return *TRUE* to say we drew the object.

We write our compare function as a static member function, so that when we pass the function to a constructor that takes a compare function, the compiler doesn't return an error that the function is of the wrong type.

Compare functions take two void pointers which point to the two objects to compare and return an *int*. If the first object is greater than the second, the compare function must return a positive value; if the first one is less than the second, the compare function must return a negative value; and if they are equal, the compare function must return a zero. Since Zinc only passes us a void pointer, this is one of the few places we must use a typecast. Last, we declare any member variables necessary to hold onto our data.

Inside of the program, we create a list, give it the compare function, and fill it with our derived objects. Then, as with other programs, we put our field on a window, add the window to the Window Manager, and go into our control loop.

Event and palette mapping

CALC

A simple calculator program that replaces the *hotKeyMapTable* defined in the library to show how to change the mapping of events.

Concepts demonstrated:

- Event mapping.
- User function as a class member.

Design principles: The *hotKeyMapTable* has four fields per entry. The first entry is the objectID. The second is the event that should be returned. The third is the event type. The fourth is the raw code. The main function replaces the *hotKeyMapTable* defined in the library. The main function remaps the *_hotKeyMapTable* variable declared in **G_WIN.CPP** so that it points to the new *hotKeyMapTable*.

Notice the *hotKeyMapTable* and the *eventMapTable* look the same, but the *hotKeyMapTable* does not use the *objectID* or the event type. Both of these fields could contain 0 without affecting the program, and so the example program can use *E_KEY* as the event type for all environments. When changing the *eventMapTable*, the *objectID* field should correspond to the object that should receive the event, such as *ID_BUTTON* for a **UIW_BUTTON**.

The event type must correspond to the operating system. For example, a keyboard event in DOS would have the *E_KEY* event type. However, in Windows, the keyboard event type would be *WM_CHAR* or *WM_SYSCHAR*.

A user function must be a static function. The constructor of an object requires the address of a user function as a parameter. Only a static function has an address at all times. Static member function do not have access to nonstatic members. Nonstatic members are accessed through the static user function calling a nonstatic member function.

CALNDR

Implements a **DrawItem( )** function to draw a calendar grid on a window. It also uses its own event map table to map events.

Concepts demonstrated.

- **DrawItem()** function
- Event mapping
- Help system
- Derived window with an overloaded **Event( )** function.
- Pop-up items with both a user function and value.

Design principles. The **DrawItem( )** function is implemented in the derived class **DAYS_OF_MONTH**. This function determines which of program's palettes it should use when drawing the calendar. The *WOS_OWNERDRAW* flag must be set the object that uses the **DrawItem( )** function. In this program the flag is set in the constructor of *DAYS_OF_MONTH*. This flag tells Zinc to call the **DrawItem( )** function, and does not let the operating system or Zinc draw the object.

CALENDR creates an event map table. An event map table has four fields per entry. The first is the objectID. The second is the event that is returned. The third is the event type. The fourth is the raw code. The new event map table has entries for a new object **ID_CALENDAR**. For these entries to be used the *windowID* array must have a match for **ID_CALENDAR**. The *windowID* array is changed in the constructor for the **CALENDAR** class. The *eventMapTable* pointer, a public member of **UI_WINDOW_OBJECT**, points to the default event map table. For the new event map table to be used, this pointer must be changed to point to the new table or the **Event( )** function of the object must call **MapEvent( )** directly and pass in a pointer to the new event map table.

Using a derived window allows us to overload the **Event( )** function, which can trap user-defined messages or other predefined messages. All messages not handled by the **Event( )** function should be passed down to the base class for processing.

Assigning a value to the pop-up item allows us to assign the same user function to each pop up item. The user function tests the value of the selected pop-up item to determine what it should do. We can use this same method on any object that has a value field.

Get/set data

PHONBK

Uses persistence to load and store phone numbers from a data file. The user has the options of loading, saving, or deleting phone numbers to their data file.

Concepts demonstrated.

- User functions.
- Getting and setting text on a window.
- Loading and saving data through persistence
- **UI_STORAGE_OBJECT_READ_ONLY**
- **UI_STORAGE_OBJECT**

Design principles. In **PHONEBK**, each button on the window uses a user function, each of which uses the **Information( )** function to get and set text from the fields on the window. The **UI_STORAGE_OBJECT_READ-_ONLY** and **UI_STORAGE_OBJECT** member functions make the data persistent.

WINDOW

Creates two windows, a **UIW_WINDOW** with name, address, telephone number entry fields; and a modal dialog window with an **OK** button. Then the user can see the data and press the **OK** button to dismiss the dialog window.

Concepts demonstrated.

- **DataSet( )**
- **DataGet( )**
- Message passing
- Modal dialog windows
- Persistence

Design principles. This program takes the data out of the fields in the first window and uses **DataGet( )** to retrieve the data from the entry window and uses **DataSet( )** to pass them to the dialog window.

POSTWN

At launch time, a window with some data entry fields appear on the screen. The user types in some data, which the program saves in a **.DAT** file when the user presses the save button. At the next launch, the program displays the data the user saved during the last session.

Concepts demonstrated.

- Persistent object storage and retrieval

Design principles. The window is an instance of **UIW_WINDOW**, which contains string fields and an instance of **UIW_BUTTON**. The button instance has a user function that calls the **UI_WINDOW** persistence members to save the data to the **.DAT** file.

NOTEBK

Creates a **UIW_NOTEBOOK** object with four pages. Each page shows different information from an imaginary employee record.

Concepts demonstrated.

- Using the **UIW_NOTEBOOK** object.

STATUS

Demonstrates how to check the status flags of a group of checkboxes. Shows how to set the status of a group of checkboxes in one group based on the settings in another group. Also shows how to create and add an object to a vertical list. When run, the main window comes up as a minimized icon, which the user can then maximize.

Concepts demonstrated.

- Adding a window to the windowManager in minimized or maximized state.
- **FlagSet( )** macro
- Setting the *woStatus* flags of an object
- Using a **for( )** loop to get a pointer to a window's subobjects
- **Destroy( )** function
- **Index( )** function
- **Get( )** function
- *S_REDISPLAY*
- *BTF_DOUBLE_CLICK* flag
- Using bitmaps and icons from a **.DAT** file
- Sizing a button

Design principles. **STATUS** uses an **Event( )** function to process user requests. It uses a simple **for( )** loop to go through group 1's checkbox child objects, and uses the **FlagSet( )** macro to check if a checkbox is checked.

When the user clicks on the **Set group2** button, a for() loop checks group 1's checkboxes, and sets group 2's checkboxes to match. When the user clicks the **Copy to vt list** button, entries reflecting group 1's settings are added to the vertical list. The reset button has a user function that demonstrates the *BTF_DOUBLE_CLICK* flag.

MENUS

Changes the status of menu items when different buttons and menu items are selected.

Concepts demonstrated.

- The *WOF_NON_SELECTABLE* flag.
- The *WOS_SELECTED* flag.
- The **Information( )** function.
- The **Get( )** function.

Design principles. **MENUS** uses the **Get( )** function to get pointers to the different menu items. It then modifies the flags associated with those menu items and uses the **Information( )** function to modify their appearance.

SPREAD

*D*emonstrates the usage of the **UIW_TABLE**, **UIW_TABLE_RECORD**, and the **UIW_TABLE_HEADER** classes by creating a simple spreadsheet.

Concepts demonstrated.

- Deriving a class from the **UIW_TABLE** class.
- Deriving a class from the **UIW_TABLE_RECORD** class.
- Getting and setting table record data.
- Loading and storing table data with the **UI_STORAGE** class.

Design principles. Derives a class called **SPREAD_SHEET** from the **UIW_TABLE** class. Creates a multiple-column table, and provides an **Event**( ) function for processing special spreadsheet events. The *TBLF-_GRID* flag causes the table to draw lines separating the rows and columns of the table, giving it a spreadsheet appearance. The **SPREAD_SHEET** class also performs the calculations required for maintaining the spreadsheet.

A class named **SPREAD_SHEET_CELL** is derived from the **UIW_TABLE_RECORD** classes. This class processes the *S_SET_DATA*, *S_CURRENT*, *S_NON_CURRENT*, and selection events. It communicates with its parent spreadsheet class by sending events. The spreadsheet cell contains one **UIW_STRING** child object used for displaying the cells contents.

The spreadsheet data consists of an array of pointers initialized to *NULL*, and memory is allocated only when data is entered into a cell. At this time, the corresponding pointer gives the address of the new data.

SPREAD_SHEET uses the **UI_STORAGE** class to load and store its data in an environment-independent manner.

AGENCY

Shows a list of entries like what you might see for a travel agency. You can add new objects to the list and delete old ones. There is also a save menu option which allows you to save the window and its contents. Entries in the list are saved as persistent objects.

Concepts demonstrated.

- Providing an **Event( )** function for a derived class
- Creating and using user-defined events
- Providing a **DrawItem( )** function for a derived class
- Using drawing primitives from a derived **UI_DISPLAY** class
- Loading and saving data through persistence.

Design principles. We derive **AGENCY_ENTRY** from **UIW_BUTTON** and give it two constructors, one for creating an instance without persistence and one for creating one with persistence. In our case the first constructor will call the standard **UIW_BUTTON** constructor and the second one the persistent object constructor.

Persistent object constructors take three parameters, pointers to **ZIL_ICHAR**, **ZIL_STORAGE_READ_ONLY**, and **ZIL_STORAGE_OBJECT_READ_ONLY**, which we must pass to the base class constructor. Inside the constructor we call the virtual **Load( )** function to load the data for our derived class.

The virtual **Load( )** function takes the same three parameters as the persistent constructor, but we are only going to use the third parameter, a pointer to **ZIL_STORAGE_OBJECT_READ_ONLY**, to load data from the persistent object file. We can use any **Load( )** function documented in the **ZIL_STORAGE_OBJECT_READ_ONLY** chapter of the reference guide.

When using the **Load( )** function across multiple operating environments, we should be sure to take into account different byte-ordering schemes. On some platforms the type *int* is of different sizes and different byte orders. We should instead use specific types such as *ZIL_INT16* or *ZIL_INT32*. Zinc will take care of byte-ordering differences automatically if we use these types.

Our class also has a virtual **Store( )** function, which also takes the same three parameters as the persistent constructor. We must ensure that our **Store( )** function loads the same data, and in the same order. Since the constructor must first call the base class, we must first call the base class's **Store( )** function. Then we store the data by calling the **Store( )** function from the **ZIL_STORAGE_OBJECT_READ_ONLY** parameter.

We must also provide an object ID and a **New( )** function for our class. The unique object ID is a variable of type *OBJECTID*, which we will use in the object table to look up objects for a **New( )** function. The **New( )** function takes the same three parameters as the persistent constructor and returns a pointer to a **UI_WINDOW_OBJECT**. It merely creates a new instance of our class, using those parameters and returns a pointer to the object. The address of the **New( )** function is placed in the *objectTable*. When a window is loading it children it does a lookup in the *objectTable* for the appropriate **New( )** function. These are used because of the difficulties with trying to put constructors directly into the table.

Now that we have persistence in our class we can save our window and the contents of the window will be saved automatically. When the user requests a save we construct a **ZIL_STORAGE**. This is the class that accesses the persistent object file for us. We specify the file name and *UIS_READWRITE* to allow us to write in the persistent object file.

Then we call the window's *Store* function to save it. We pass in the name of the window and the **ZIL_STORAGE** we just created, but let the third parameter default out. This stores all of the data to a temporary file. Then we call **ZIL_STORAGE**'s *Save* function to resolve our data out to the permanent file. Finally, we delete **ZIL_STORAGE** to flush any buffers and close the persistent object file. The next time we run our program, the window will be loaded with the contents saved during the previous session.

I18N

I18N

Shows how to decouple strings used in a window from the creation and manipulation of the window, allowing easy internationalization of the window. Also shows how to override the strings compiled into the library without editing and recompiling the library source.

Concepts demonstrated.

- Using **ZIL_LANGUAGE_MANAGER** and **ZIL_LANGUAGE** to manage two different languages.

Design principles. We declare a new class called **I18N_WINDOW**, which consists of a window; title; system, minimize, and maximize buttons; and three prompts. We want to change the language on the title and prompts so that we can easily add other languages and so that we don't have to modify **I18N_WINDOW**'s source code. We do this using the **ZIL_LANGUAGE_-**

MANAGER. First we set up the set of strings that we want to use and bind them to the **I18N_WINDOW::_className**. Then **I18N_WINDOW** in the constructor fetches the strings by using its class name and gets the strings it needs to use. For completeness, we also replace the system button strings. (Notice that some operating systems may not use the replaced system button strings.)

DELTA

Shows how to load an object that was stored by the Designer as a **DELTA** object. It also shows how multiple different language/locale resources may be stored in the same file with a minimal amount of overhead. Using this method, an application built with Zinc Application Framework may support multiple languages and locales from one executable and one **POST** file. Only the changes (the delta) between different versions of each resource will be stored, reducing the amount of disk space used.

Concepts demonstrated.

- Using **ZIL_DELTA_STORAGE** and **ZIL_DELTA_STORAGE_OBJECT** to manage two different languages.

Design principles. A resource is loaded from the file **P_DELTA.DAT**. It contains a date and a time. The first resource uses the simple numeric formats to display all the objects. The second resource displays the date with named months and days of the week, while the time uses twelve-hour formatting ("am"/"pm"). The field sizes have been adjusted for the larger field sizes needed. Both windows are stored in the same **POST** file, but only the changes made to the second are really stored. The size of the first window is 161 bytes, while the size of the second window is only 36 bytes, including the overhead of the delta object information, which in this case is 12 bytes.

Messages

MESSGS

Uses the event structure to create an event that can be placed on the queue and trapped by a derived window's **Event( )** function.

Concepts demonstrated.

- Creating events
- Derived window with an overloaded **Event( )** function.

Design principles: An event is created in the **LeaveMessage( )** user function by setting *event.type* to the desired event. The *event.data* part of the structure is a void pointer used to pass more information with the event.

The overloaded **Event( )** function traps user-defined events or predefined Zinc or operating system events. This can be used to override existing functionality or when implementing new functionality. Any events not handled by the overloaded **Event( )** function should be passed on to the base class for processing.

MATCH

The user matches two buttons with the same bitmaps in a simple game of "concentration." Through persistence, the program loads a bitmap from disk and assigns the bitmap to a button.

Concepts demonstrated.

- Deriving a window.
- Deriving a button.
- Loading a bitmap image through persistence.
- Assigning a bitmap to a button.
- Message passing.

Design principles. **MATCH_WINDOW** is the control window for the program. **Event( )** handles the calls to switch bitmaps for selected buttons, to create a new set of bitmapped buttons, and to take the matching buttons off the display. **Jumble( )** randomizes the bitmaps used for each button displayed. The *buttonWindow* member actually contains the bitmapped buttons.

MATCH_BUTTON is the bitmapped button being added to the **MATCH_WINDOW** object's *buttonWindow.* Its **SetBitmap( )** switches the bitmap for the button by loading the desired bitmap from disk and setting it on the button. Each button sends a *TOGGLE_BITMAP* message to the **MATCH_WINDOW** object via the event queue when the button is selected.

One of the menu items used sends a *NEW_GAME* message to the **MATCH_WINDOW** by placing it on the event queue when selected. This message causes a new set of bitmapped buttons to be created.

WORLD

Shows how to broadcast messages using a derived window manager as well as updating a bitmap of a rotating world.

Concepts demonstrated.

- Deriving a window manager.
- Deriving a device.
- Broadcasting messages to all windows.

Design principles. **WINDOW_MANAGER** is the derived window manager that broadcasts the messages to all the attached windows.

WORLD_WINDOW is a derived window that receives the message to update its own rotating world bitmap. Each window updates it's own independently rotating world.

WORLD_DEVICE is the derived device that places an update message on the queue when an appropriate amount of time has passed.

Miscellaneous

FRESTR

Implements a free store exception handler by installing a new handler. When the **new( )** operator fails to allocate memory, the new handler will be called allowing the application to recover gracefully.

Concepts.

- Implementing and installing a new handler.
- Performing a compiler independent task.

Design principles. **MEMORY_ALLOCATION_ERROR_SYSTEM** is a class that handles a free store exception. It will contain a constructor, a destructor, and a routine to handle the exception.

The running example will add a window to the Window Manager with this message:

```
This is a test of the Free Store Exception Handler.
```

When the new operator fails, a *NULL* is usually returned. For this example a "new handler" is called instead. This handler takes control, notifies the user, and cleans up and exits. It may take a few hundred windows to use up all of memory. Be patient and watch.

The class will then go through a loop that continually creates windows and adds them to the Window Manager. When memory is exhausted, and the new handler invoked, a dialog window will appear explaining the system is out of memory and allow a graceful exit back to DOS.

This program is a DOS-only example. Since there is no standard for this exception, and each compiler implements doing a new handler differently, there will be multiple **#if defined(..)** statements throughout the code. This will be an example for programmers on how to go about other compiler-independent tasks, such as Critical Error handlers and Interrupt Service Routines.

DRAG

Simulates a file manipulation dialog window. The user can pretend to move files from one location to another by dragging them from list to list.

Concepts demonstrated.

- Using the *WOAF_ACCEPTS_DROP* flag.
- Using the *WOAF_DRAG_OBJECT* flag.

Design principles. In order to drag an object it must have the *WOAF_DRAG_OBJECT* flag set. We set this flag on the items in the lists so they can be moved.

In order for an object to accept a dragged object it must have the *WOAF_ACCEPTS_DROP* flag set. We set this flag on the list objects so they can accept objects that are dragged to them.

The receiving object determines if an object can be copied or moved. Since we are dragging from list to list, the objects can either be copied or moved.

SPY

Reports the events that go through the system on a scrolling **TTY** window.

Concepts demonstrated:

- Deriving a device.
- Implementing a **Poll( )** member function for a derived device.
- Deriving a prompt.
- Deriving a **TTY** window.
- Implementing a **Printf( )** member function for a **TTY** window.

Design principles. The **SPY** class is derived from **UI_DEVICE**. The **SPY** class passes most of the events it receives down to the **UI_DEVICE** class. The **SPY** device's event method handles only the *S_INITIALIZE*, *D_ON*, and *D_OFF* events. This way, **SPY** retains all normal device functionality and responds to the *S_INITIALIZE* and *D_ON* events by adding its spy window to the Window Manager, and responds to the *D_OFF* event by subtracting its spy window from the window manager. **SPY** includes the spy window, which is a normal **UIW_WINDOW** that provides the messages to be monitored in the **TTY** window. **SPY** also includes the **TTY** window itself, so that the **SPY** device may call the **TTY** window's **Printf( )** member function to display the events monitored.

The **SPY** class defines a **Poll( )** member function that is called instead of the base **UI_DEVICE Poll( )** member function. The **SPY** device's **Poll( )** member function uses a lookup table to map each event monitored to a message displayed in the **TTY** window. Only messages that flood the system, such as mouse-move messages, or those that do not provide interesting information, are filtered out.

The **TTY_ELEMENT** class derives from **UIW_PROMPT**. The **TTY_ELEMENT** class passes most of the events it receives down to the **UIW_PROMPT** class. The **TTY_ELEMENT** event method handles only the *S_INITIALIZE* event. In this way, **TTY_ELEMENT** retains all normal prompt functionality and responds to the *S_INITIALIZE* event by adjusting its relative region according to the height passed to the constructor by the **TTY** window when the **TTY** window creates the **TTY_ELEMENT** object. The **TTY** window calls the **DataSet( )** member function of the **TTY_ELEMENT** class, which the **TTY_ELEMENT** class does not define, and so it inherits the **DataSet( )** member function defined by the **UIW_PROMPT** class.

The **TTY** class derives from **UIW_WINDOW**. The **TTY** window passes most of the events it receives to the **UIW_WINDOW** class. The **TTY** window's event method handles only the *S_CREATE* and *S_CHANGED* events. This way, **TTY** retains all normal window functionality and responds to the *S_CREATE* and *S_CHANGED* events by updating its children and class members.

The **TTY** class defines a **Printf( )** member function that provides functionality similar to the **printf** function in the standard C library. **Printf( )** provides formatting using the **vsprintf** function in the standard C library. The Printf member function makes use of the **TTY_ELEMENT** class to display and scroll as many lines of text as will fit in the TTY window. **Printf( )** provides scrolling by calling the **DataSet( )** member function of each **TTY_ELEMENT** object on the **TTY** window.

COORDS

Demonstrates the differences between *cell*, *mini-cell*, and *graphics* coordinates.

Concepts demonstrated.

- Use of the *WOF_MINICELL* flag.
- Use of the *WOS_GRAPHICS* flag.

Design principles. **COORDS** uses the *WOF_MINICELL* and *WOS-_GRAPHICS* flags to change the way the size parameters of each window are interpreted as they are being added to the window manager.

FONTS

Shows how to add and use a new font for each Zinc-supported platform.

Concepts demonstrated.

- Creating a font in each supported platform.
- How to use the newly created font.

Design principles. **FONT** displays four **UIW_STRING** objects that use the three default fonts an one newly created one for each supported platform.

LoadFont( ) is the function that creates and loads the new font into the **UI_XXX_DISPLAY::***fontTable* for use.

COLORS

Creates a window containing 256 boxes, where each box is drawn in one of 256 colors. The program will run only in VGA or SuperVGA graphics modes that support 256 or more colors.

Concepts demonstrated.

- Deriving a window.
- Using the *WOS_OWNERDRAW* flag.
- Using display primitives within a **DrawItem( )** function.
- Using 256-color palettes.

Design principles. **COLOR_WINDOW** is a derived window with its own **DrawItem( )** function. This function draws 256 filled boxes within the window, each box displaying one of 256 colors. The **DrawItem( )** function also displays the corresponding palette number (0..256) above each colored box. Also, the **RGBConvert( )** function is used to convert the RGB values, stored in the **.HPP** file, to the Windows RGB format.

The **.HPP** file contains the color value for each of the colors numbered 16 through 255. RGB color values are also used for those environments in which they are supported.

New objects

GMGR

Displays five Zinc windows, each of which shows some uses of Zinc's geometry management.

Concepts demonstrated.

- Constructing instances of the **UI_GEOMETRY_MANAGER** class, and adding these instances to windows
- Use of the following **UI_CONSTRAINTS—UI_ATTACHMENT**, **UI-_RELATIVE_CONSTRAINT**, **UI_DIMENSION_CONSTRAINT**
- Use of the constraint flags

Design principles. **Gmgr** creates the following example windows:

- **BasicGMWindow**. Shows basic geometry management with buttons being attached to the sides of the parent window. As the parent is resized, the buttons all stay in the corners. This window also shows a text object that resizes with the parent window, with maximum and minimum limits.
- **TieGMWindow**. Shows how to tie child buttons to other child buttons. When a button moves because its parent is resized, those buttons that are tied to it move also.
- **OppositeGMWindow**. Shows how to tie one edge of a child to the opposite edge of its parent.
- **RelativeGMWindow**. Shows how to tie a child to a relative position on its parent. For example, we attach the top left corner of a button to the point on the parent window that is 10% from the left edge and 10% from the top. We also attach the bottom right corner of the button to a point 40% from the left and 40% from the top of the parent. This causes the button to retain its relative position and size on the parent regardless of the parent's size.
- **CenterGMWindow**. Shows how to attach the center of a child to a relative point on the parent. This lets a child recalculate its position as its parent is sized, while retaining its original size.

All of these example windows follow the same procedure for implementing geometry management:

- Each window is created in a separate function.
- A pointer to each child object that will have its geometry managed is created. We use the pointer twice: once to add the child to its parent, and again to create a constraint for that child.
- A **UI_GEOMETRY_MANAGER** is created and added to the parent window before any children that it will manage.
- **UI_CONSTRAINTS**, such as **UI_ATTACHMENT**, **UI_RELATIVE_CONSTRAINT**, and **UI_DIMENSION_CONSTRAINT**, are created and added to the *geometryManager*. Each constraint is associated with one or two child objects derived from **UI_WINDOW_OBJECT**. These **UI_CONSTRAINTS** are created after their associated objects are created.

PRINTR

Demonstrates a **UI_PRINTER** object by printing text and graphics.

Concepts.

- Printing from within an application.
- Using an object's **DrawItem( )** for both drawing and printing.
- Sending a single-page print job to the printer.
- Sending a multiple-page print job to the printer.
- Sending a print job to a PostScript file.
- Screen dumps
- Deriving a new **UI_HELP_SYSTEM** capable of printing.

Design principles. The main application window is a generic **UIW_WINDOW**, with a menu of print options. A nonfield window for drawing display primitives occupies the client area of the window.

We print from the application using the **Print** menu option that uses a call-back routine. When the user selects this option, the program presents a dialog window that allows setting up the print job. The user has a choice of printing graphics or dumping the contents of the screen to the printer or to a PostScript file.

The **DRAW_WINDOW** class derives from **UIW_WINDOW**. It contains its own **DrawItem( )** routine for drawing display primitives such as **Bitmap( )**, **Line( )**, **Rectangle( )**, **Ellipse( )**, and **Polygon( )** for both the display and the printer. This object is the nonfield window added to the main application window.

Sending a single-page print job to the printer occurs when doing screen dumps and printing the graphics from **DRAW_WINDOW**. Sending a multiple page print job happens when printing a text document that spans multiple pages. A class is derived from **UI_HELP_SYSTEM** to do this.

This program creates a **PRINTABLE_HELP_SYSTEM** class that derives from **UI_HELP_SYSTEM** in order to add a **Print** option to the help window, and to start the print jobs that format and output the pages of help text to the printer.

Some notes on using **UI_PRINTER**:

- All print jobs must begin with a call to **UI_PRINTER::BeginPrintJob( )**.
- All print jobs must end with a call to **UI_PRINTER::EndPrintJob( )**.
- All pages must begin with a call to **UI_PRINTER::BeginPage( )**.
- All pages must end with a call to **UI_PRINTER::EndPage( )**.
- All drawing to be done on a page must occur between a **BeginPage( )** / **EndPage( )** pair.
- All **Pages** to be printed must be done between the **BeginPrintJob( )** / **EndPrintJob( )** pair.
- **UI_PRINTER::ScreenDump( )** makes the calls to **BeginPrintJob( )**, **EndPrintJob( )**, **BeginPage( )**, and **EndPage( )**. The **ScreenDump( )** routine is the only exception to the above rules.

In DOS, we must have an environment variable *ZINC_PRINTER* and the *lpt* identifier set to a string that identifies the type of default printing and the printer port. If an environment variable doesn't exist, the class will write printer output to a PostScript file.

Types of default printing:

- PS
- PCL
- DM9
- DM24

Types of *lpt* values:

- LPT1
- LPT2
- LPT3

Examples:

- **SET** *ZINC_PRINTER=PS,LPT1*

  Sets defaults to PostScript output, port 1.
- **SET** *ZINC_PRINTER=PCL,LPT2*

  Sets defaults to PCL output, port 2.
- **SET** *ZINC_PRINTER=DM9,LPT3*

Sets defaults to nine-pin dot matrix, port 3.

- **SET** *ZINC_PRINTER=DM24,LPT1*

  Sets defaults to 24-pin dot matrix, port 1.

When printing output to a PostScript file, the *ZINC_PRINTER* environment variable is ignored.

SPIN

Simulates a simple video editor interface using spin controls and sliders to set various values.

Concepts demonstrated.

- Deriving your own object for use with a spin control.
- Setting ranges on a spin control.
- Setting ranges on a slider.
- Using user functions with a slider.

Design principles. A spin control by default only works with five Zinc objects: **UIW_BIGNUM**, **UIW_DATE**, **UIW_INTEGER**, **UIW_REAL**, and **UIW_TIME**. But it can be used with any window object that handles the *I_DECREMENT_VALUE* and *I_INCREMENT_VALUE* information requests.

The object that is to be spun is passed to the spin control in the spin control constructor. You set a range of values for a spin control by passing an appropriate range in the controlled object's constructor.

To create a slider, set the *SBF_SLIDER* flag in the **UIW_SCROLL_BAR** constructor. This flag causes the slider to draw differently than a scroll bar in most environments. It also causes a vertical slider to move its thumb button up as the value increases, as opposed to a scroll bar, which moves its thumb button down when it is increasing.

To set the range on a slider, set up a **UI_SCROLL_INFORMATION** structure with the desired values. This structure is passed into the **UIW-_SCROLL_BAR** constructor.

Other than its appearance and the reverse thumb button motion on vertical sliders, sliders operate the same as scroll bars do. Typically, you use a slider by itself by associating it with a user function in the constructor, as a user function can be associated with a scroll bar. The user function is called when the slider is manipulated.

Notice that we used positional parameters on the slider. This allows us to position the slider where we want it. If the *WOF_NON_FIELD_REGION* flag is set, these parameters are ignored. Scroll bars are usually nonfield regions but sliders often are not.

MDI

Demonstrates creating a Multiple Document Interface (MDI) application. One MDI frame window is created that contains multiple MDI child windows. Scrolling window capabilities are added to the MDI frame window and to all the MDI child windows.

Concepts.

- Creating an MDI frame **UIW_WINDOW**.
- Creating MDI child **UIW_WINDOW**s.
- Deriving a persistent **UIW_WINDOW**.
- Scrolling an MDI frame window.
- Scrolling the MDI child windows.
- Adding MDI children at run-time.
- Removing MDI children at run-time.
- Activating MDI children through the MDI frame menu.
- Using **UIW_ICON**s as selectable objects.

Design principles. The MDI application window is created with Zinc Designer, which contains a pull-down menu and three MDI child windows.

An **MDI_FRAME_WINDOW** class is derived from **UIW_WINDOW**. It contains a persistent constructor to load the window. **MDI_FRAME_WINDOW** also has its own **Event( )** routine to handle events generated by the menu and selectable MDI children icons.

The menu will contain options so the user can add, remove, and activate the child windows. The **File** pull down item option contains a **New** submenu item to add a new MDI child window, and **Delete** to delete the active MDI child. The window also contains a **Window** pull-down item that identifies the MDI children. When the user selects a certain window in the menu, that window will become current.

MDIWIN adds scroll bars to the MDI frame window as well as the MDI children. The MDI child windows contain selectable **UIW_ICON** objects.

PERIOD

Shows many of the basic Zinc objects on a single window to show the objects available in Zinc.

Design. **Periodic** loads a window from a data file and adds it to the window manager. Its purpose is to display the Zinc objects, rather than to teach Zinc programming principles.

TABLE

Demonstrates the **UIW_TABLE**, **UIW_TABLE_RECORD,** and the **UIW_TABLE_HEADER** classes. Shows a table of sales figures and a **UIW_STATUS_BAR** to show totals. Also includes menu options for adding and deleting records.

Concepts demonstrated.

- Creating a table, table headers, and table records.
- Controlling data and status presentation.
- Adding and deleting records.
- Accessing record data.
- Using a status bar to show table information.
- Processing menu events in a derived class's **Event( )** function.

Design Principles. This example derives a class named **TABLE_WINDOW** from the **UIW_WINDOW** base class. The **TABLE_WINDOW** class provides a constructor which creates the child **UIW_TABLE** object, as well as a pull-down menu and other support objects. The derived class also provides an **Event( )** function which processes menu-generated events.

This example leaves out the left, top, width, and height parameters required by the **UIW_TABLE** class constructor, because the table object uses the *WOF_NON_FIELD_REGION* flag, which causes the table to occupy the entire client area of the window. This example sets the *columns* parameter to one because the table contains only one column of records. Each record, however, contains multiple fields, which gives the table an appearance of multiple columns, but with each row grouped into one record.

The table displays the data contained in up to 100 structures of type **DATA_RECORD**. Initially, however, it contains only 10 records. To accomplish this, *recordSize* is set to **sizeof(***DATA_RECORD*), *maxRecords* to 100, *records* to 10, and the data for the initial 10 records is passed in the data parameter as an array of *DATA_RECORD* structures.

If the table will display large amounts of data, we can set *recordSize* and *maxRecords* to 0 or the size of a database key, and access the data during the processing of the *S_SET_DATA* event. Using these techniques, the table can display an unlimited number of records.

At any time, we only add one **UIW_TABLE_RECORD** object to a table or table header. Several fields are added to the table record, however, in order to display the data contained one **DATA_RECORD** structure.

One table record displays all records in the table. A user function assigned to the table record object associates the data with the fields in the table record. Whenever the user function is called with a control code of *S_SET_DATA* the function sets the data from the appropriate record into the fields on the table record. It also sets the selected status of the table record according to the status in the *DATA_RECORD* structure. By sending the *S_SET_DATA* event multiple times, the table can use one table record to display all of the data in the table.

The menu items in this example are all flagged with the *MNIF_SEND_MESSAGE* flag. This causes the menu options to generate an event each time they are selected. Two special events have been defined in this example: *S_ADD_RECORD*, and *S_DELETE_RECORD*. These events are processed in the table window's event function, and use the table's **InsertRecord( )** and **DeleteRecord( )** functions to perform the required actions.

Whenever a table record is selected or loses focus, the user function generates a special event of type *S_CALCULATE_TOTALS*. The table window processes this event by using the table's **GetRecord( )** event to get data from the table, calculating totals, and setting the results into the table's status bar.

MSGWIN

Displays a window with two data entry fields, one for dates and the other for strings. Giving a date to the date field and then hitting <Enter> displays a message window telling what day the date typed in fell on. If the user types a string into the string field that does not begin with 'A,' and then tries to leave the field, a notification window comes up. If you choose the **Ignore** option on this window then the invalid string is left. If you choose the **Cancel** option then the text is blanked out. When the user tries to close the window, the program verifies the user intends to exit by displaying a modal message window.

Concepts demonstrated.

- Exit functions.
- Using **ZAF_MESSAGE_WINDOW**.
- Validation using user functions.

Design principles. The Window Manager calls the exit function any time a user attempts to exit the application. Exit functions take three parameters, pointers to **UI_DISPLAY**, **UI_EVENT_MANAGER**, and **UI_WINDOW_MANAGER**; and return an *EVENT_TYPE*.

For this application we a supply an exit function to verify the user intends to exit. by creating a function called **ExitFunction( )** with the parameters and return type described above.

Inside of **ExitFunction( )** we display a message window my creating a **ZAF_MESSAGE_WINDOW** and giving it the messages we want displayed and that we want the **YES** and **NO** buttons. Then we call the messages window's **Control**( ) function.

If the return value from the **Control( )** function indicates that the **NO** button was pressed then the user doesn't want to exit and we return *S_CONTINUE* to say we wish to continue with the program. Otherwise the **YES** button was pressed and we return *L_EXIT* saying we wish to exit.

In **UI_APPLICATION::Main( )** we say *windowManager->exitFunction = ExitFunction* and we are all hooked up.

Also in **UI_APPLICATION::Main( )** we create a window with a date field, a string field and corresponding prompts.

One of the parameters for the **UIW_DATE** field is a user function. User functions take three parameters, a pointer to a **UI_WINDOW_OBJECT**, the contents of **UI_EVENT**, and an *EVENT_TYPE*; and return an *EVENT_TYPE*. The user function **ValidateDate**( ) of the **UIW_DATE** field checks to see why we call the user function. If the function was called because the field was selected, meaning the user hit <Enter>, it gets the data from the date field that was passed in as the first parameter to **ValidateDate**( ); it finds out what day of the week it falls on, and then creates a **ZAF_MESSAGE_WINDOW**. In order for **ZAF_MESSAGE_WINDOW** to find the icon assigned to it, we must set up **UI_WINDOW_OBJECT::***defaultStorage* in **UI_APPLICATION::Main( ).** The message window has an **OK** button so the user can dismiss it. Then the program calls the its **Control( )** function and then exits when it returns.

We also create a user function for the string field called **ValidateString( )**, which checks to see if the field is noncurrent. If the field is noncurrent, the function checks to see if the user changed the text, and if the user did, so it performs the validation. If the text is invalid, it will create a **ZAF_MESSAGE_WINDOW** with a message, call its **Control( )** function, and get the response. If the user chooses **CANCEL,** the function blanks out the field. Otherwise it leaves the bad text.

After the window is created and added to the Window Manager, the program sets *windowManager->screenID* to the window's screenID. This tells the Window Manager that the window is the control window. Any time the user tries to close that window, the Window Manager will assume the user is trying to exit, so it will call the exit function.

Once this is finished, we merely go into our control loop by calling **UI_APPLICATION::Main( )**, and let our program run.

Appendix C

Zinc Coding Standards

Zinc Software has an internal document that specifies standards for all code written for internal, as well as external, distribution. The purpose of these standards is to improve the readability, organization and maintenance of source code and header files. This document is printed in this appendix so that you can understand the coding standards we use when writing the example programs, tutorial programs and source code modules you receive when you purchase this product.

Naming

Classes and structures

Class names should be self-explanatory and should be in upper-case lettering, with underscores used to separate words. Some example class and structure names are shown below.

```
struct UI_EVENT
struct UI_PALETTE_MAP
class UI_ELEMENT
class UI_EVENT_MANAGER : public UI_LIST
class UIW_BUTTON : UI_WINDOW_OBJECT
```

In addition, Zinc class names use the following prefixes

- **UI_** denotes a general user-interface class or structure.
- **UID_** denotes a device class or structure.
- **UIW_** denotes a window interface class or structure.

Functions

Functions should be self explanatory and should be in name-case format, such as first letter, uppercase lettering, all remaining character in lowercase lettering, with no underscores used to separate words. In addition, the function name should describe what the function does.

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 use lowercase lettering for the first word, then name case for each word thereafter. Underscores should precede global variables. 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, and 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 uppercase 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 should be aligned to the right of the value.

Organization

Class scopes

The class declaration 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. (Be sure to list the constructor and destructor 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, static member variables, and member functions, described in alphabetical order. An example of **BORDER.CPP's** file layout is shown below:

```
//Zinc Application Framework - BORDER.CPP
//COPYRIGHT (C) 1990-1993.  All Rights Reserved.
//Zinc Software Incorporated.  Pleasant Grove, Utah  USA
#include "ui_win.hpp"
#include <string.h>
int UIW_BORDER::width = 4;

UIW_BORDER::UIW_BORDER(void) :
  UI_WINDOW_OBJECT(0, 0, 0, 0, WOF_NON_FIELD_REGION,
WOAF_NON_CURRENT)
{
...
}

UIW_BORDER::~UIW_BORDER(void)
{
...
}

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

Comments

Files

Each source file (**.CPP** or **.HPP**) should contain a three-line comment that contains the library or program name, the name of the file and copyright information. A sample header is shown below:

```
//Zinc Application Framework - BUTTON.CPP
//COPYRIGHT (C) 1990-1993.  All Rights Reserved.
//Zinc Software Incorporated.  Pleasant Grove, Utah  USA
```

The copyright information should be copied as shown above. The copyright year should include the original year when the product was created and all subsequent years when major revisions were made.

Functions

Each routine may be preceded by a short description giving the routine's purpose and any related algorithms. If the routine name intuitively describes the routine, no comment is needed. The example below shows the use of a function comment:

```
// This member function displays the biorhythm information in
// the window. As the size of the window object changes (by
// changing the parent window)
// the size of the biorhythm chart also changes. A horizontal
// change results in a change in the number of days displayed. A
// vertical change results in a dynamic change in the height of
// the biorhythm curve.
void BIORHYTHM::UpdateBiorhythm()
  {
    ...
  }
```

Variables

Function arguments and local variables should only have descriptive comments if their names are not descriptive. These comments should be lined up on a right tab region. In addition, all comments should start with a capital letter and be followed by a period. An example of two variable declarations is shown below.

```
EVENT_TYPE ccode;// The control code for an event.
int cardFile;// File handle for the disk file.
```

Blocks

Block comments are used to describe a group of related code. Most block comments should be one line, if possible, and reside immediately above the block being commented. If more than a one line comment is needed, the extra lines should each begin with the double slash.

Block comments should be indented to match the indentation of the line of code following it. A single blank line should precede the comment and the block of code should follow immediately after. Small blocks of code that do a specific job should be commented but not individual lines, unless the line is complex or not intuitive). Some example block comments are shown below.

```
// Destroy all of the items within the list.
Destroy();

// When the user selects a button from the current window, ccode

// is checked to see what type of event was received.
switch (ccode)
{
...
}
```

Indentation

Classes and structures

Structures and classes should have all members listed on individual lines and should be indented with one tab from the left margin. Several sample indentations are shown below:

```
class EXPORT UI_DEVICE : public UI_ELEMENT
{
  friend class EXPORT UI_EVENT_MANAGER;
public:
  static ALT_STATE altState;
  static UI_DISPLAY *display;
  static UI_EVENT_MANAGER *eventManager;

  int installed;
  DEVICE_TYPE type;
  DEVICE_STATE state;

  virtual ~UI_DEVICE(void);
  virtual EVENT_TYPE Event(const UI_EVENT &event) = 0;

  // List members.
  UI_DEVICE *Next(void);
  UI_DEVICE *Previous(void);
protected:
  UI_DEVICE(DEVICE_TYPE _type, DEVICE_STATE _state);
  static int CompareDevices(void *device1, void *device2);
  virtual void Poll(void) = 0;
};
```

Functions

The main body of routines should have braces below the function declaration. All function code should be indented one tab. An example of this indentation is shown below:

```
void UIW_BUTTON::DataSet(const char *string)
{
   // Reset the button's string information.
...
}
```

Function calls

Parameters in a function call should be listed with each argument, followed by a comma and one space. If a routine call cannot fit on one line on the screen, it should be broken with the next half of the call indented one tab farther over. It should be split after a comma or logic symbol if possible. Several examples of this calling convention are shown below:

```
UIW_WINDOW *UIW_WINDOW::Generic(int left, int top, int width,
int height,
  char *title, UI_WINDOW_OBJECT *minObject, WOF_FLAGS woFlags,
  WOAF_FLAGS woAdvancedFlags, UI_HELP_CONTEXT helpContext)
{
  // Create the window and add default window objects.
  UIW_WINDOW *window = new UIW_WINDOW(left, top, width, height,
    woFlags, woAdvancedFlags, helpContext, minObject);
  ...
}

UIW_WINDOW *window = UIW_WINDOW::Generic(2, 2, 40, 6,
  "Hello World Window", NULL, WOF_NO_FLAGS, WOAF_NO_FLAGS,
  HELP_HELLO_WORLD);

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

Case statements

The reserved word **case** should be aligned with the switch statement, but all code information should be indented an additional tab. Each additional level of logic should be indented one tab. The colon should immediately follow each case and the statement(s) should start on a new line. The break should also be on a separate line. An example of this organization is shown below:

```
EVENT_TYPE UIW_PROMPT::Event(const UI_EVENT &event)
{
  // Switch on the event type.
  EVENT_TYPE ccode = event.type;
  switch (ccode)
  { case S_CREATE:
    case S_SIZE:
      ...
    break;

  case S_CURRENT:
  case S_NON_CURRENT:
  case S_DISPLAY_ACTIVE:
  case S_DISPLAY_INACTIVE:
    if (UI_WINDOW_OBJECT::NeedsUpdate(event, ccode))
      UI_WINDOW_OBJECT::Text(prompt, 0, ccode, lastPalette);
  break;
  default:
  ccode = UI_WINDOW_OBJECT::Event(event);
  break;
}

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

If and for statements

Statements following an if or for should be indented one tab, and simple conditionals should use the inline ? 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.
```

Appendix D

Keyboard and Mouse Mappings

This appendix lists all the default keyboard mappings Zinc supports, organized by operating environment.

DOS and Windows

TABLE 1. DOS and Windows keyboard mappings

Action	Key	*Description*
Begin field	<Ctrl+Home> <Ctrl+Gray Home>	Moves to the beginning of field or the beginning of a list.
Close temporary window	<Esc>	If the current window is identified as a temporary window (*WOAF_-TEMPORARY*), pressing <Esc> removes the current window from the screen display. For example, when an end user selects the system button, a pop-up menu appears. If the user presses <Esc> at this time, the pop-up menu is erased from the screen display.
Close window	<Shift+F4>	Closes a window that is not temporary. **NOTE:** All temporary windows will be closed first.
Copy	<Ctrl+Ins>	Copies the marked portion of the current window field and stores it in a global paste buffer. This key only affects fields that can be edited.
Cut	<Shift+Del>	Cuts the marked portion of the current window field. The cut section is removed and stored in a global paste buffer. This key only affects fields that can be edited.
Delete	<Del>	Deletes the marked text from the current window field. The cut section is *not* stored in the global paste buffer. This key only affects fields that can be edited.
Delete next character	<Del> <Gray+Delete>	Deletes the character beneath the cursor, leaving the position of the cursor unchanged. This key only affects fields that can be edited and only where the cursor is not in the field's *last* position.
Delete previous character	<Backspace>	Moves the cursor *left* one position, deleting the character to the character immediately to the left of the cursor. This key only affects fields that can be edited and only where the cursor is not in the field's first character position.
Delete word	<Ctrl+Del> <Ctrl+Gray Delete>	Positions the cursor at the beginning of the word to be deleted, then deletes the word and any trailing spaces. The cursor remains in its original position after the deletion.
Down	<↓> <Gray ↓>	If the field is a multi-line field and the cursor is not positioned on the bottom line, pressing <Down-arrow> moves the cursor *down* one line on the display.
Down page	<PgDn> <Gray+PgDn>	If the field is a multi-line field and the cursor is not positioned on the bottom line, pressing <PgDn> moves the cursor *down* one page in the current field.
End field	<Ctrl+End> <Ctrl+Gray End>	Moves to the end of the field.

TABLE 1. DOS and Windows keyboard mappings

Action	Key	*Description*
End line	<End> <Gray End>	Moves the cursor to the end of the current line.
Exit (DOS only)	<Alt+F4> <Shift+F3> <Ctrl+Break> <Ctrl+C>	Exits the application program. (The <Ctrl+Break> and <Ctrl+C> keystrokes can be modified by changing **UID_KEYBOARD::breakHandlerSet**)
Help—context sensitive	<F1>	Displays context-sensitive help information for the current window object.
Help— general	<Alt+F1>	Displays general help information for the application.
Home	<Home> <Gray Home>	Moves the cursor to the beginning of the current line.
Left	<←> <Gray ←>	If the cursor is not positioned in the *first* character position of a field, pressing <Left-Arrow> moves the cursor one character to the left.
Left word	<Ctrl ←> <Ctrl+Gray ←> <Alt+Gray ←>	Moves the cursor to the beginning of the previous word or to the beginning of the same word if the cursor was originally positioned in the middle of that word.
Mark (DOS only)	<Shift+←> <Shift+→>	Begins a marked region on the position of the cursor.
Menu control	<Alt> <F10> (DOS only)	Toggles between selecting the pull-down menu and the current window field. This changes the highlight field, or cursor position, from the current field to the pull-down menu. This key only affects the window when the current window has a pull-down menu.
Minimize (DOS only)	<Alt –> <Alt+F9>	Minimizes the size of the current window (i.e., reduces the size of the window to the minimum allowed by the object type). This key only affects the window when the current window can be sized and if it is not already in a minimized state. If the window is in a minimized state, selecting this key causes the window to be restored to its original size.
Move window (DOS only)	<Alt+F7>	Moves the current window when followed by any movement key and then <Enter>. When followed by any movement key and then <Esc>, the selected window is returned to its original position.

TABLE 1. DOS and Windows keyboard mappings

Action	Key	*Description*
Next field	<Tab> <F6> (DOS only)	Moves from the current (or selected) window field to the *next* selectable window field. If the last window field is currently selected, pressing <Tab> cycles to the *first* selectable window field.
Next window	<Alt+F6>	Moves from the current (or selected) window to the *next* selectable window in the Window Manager's list of windows.
Next MDI window	<Ctrl+F6>	Moves from the current (or selected) MDI child window to the *next* selectable MDI child window within the parent window's list of windows.
Paste (DOS only)	<Shift+Ins>	Retrieves the cut section from the global paste buffer and pastes it in the current field. This key only affects fields that can be edited.
Previous field	<Shift+F6> <Shift+Tab>	Moves from the current (or selected) window field to the *previous* selectable window field. If the first window field is currently selected, pressing <BackTab> cycles to the *last* selectable window field.
Refresh	<F5>	Refreshes the screen.
Restore (DOS only)	<Alt+F5>	Restores the original size of the window. Used with <Alt +> and <Alt ->.
Right	<→> <Gray →>	If the cursor is not positioned in the *last* character position of a left-hand field, pressing <Right-Arrow> moves the cursor one character to the right.
Right word	<Ctrl+→> <Ctrl+Gray →>	Moves the cursor to the beginning of the next word.
Size window (DOS only)	<Alt+F8>	Sizes, relative to the top left corner, the current window when followed by any movement key. Pressing <Enter> accepts the alteration in size, while pressing <Esc> returns the window to its original size.
System	<Alt+Spacebar> <Alt+.> (DOS only)	Selects the system button (if any) associated with the current window. This causes the pop-up menu associated with the current window's system button to be displayed on the screen.
System (MDI)	<Ctrl+Spacebar>	Selects the system button (if any) associated with the current MDI child window. This causes the pop-up menu associated with the current MDI child window's system button to be displayed on the screen.
Toggle	<Ins> <Gray Insert>	Toggles the edit mode from *insert* to *overstrike* mode or vice-versa. This key only affects fields that can be edited.
Up	<↑> <Gray ↑>	If the field is a multi-line field and the cursor is not positioned on the top line, pressing <Up-arrow> moves the cursor *up* one line on the display.
Up page	<PgUp> <Gray+PageUp>	If the field is a multi-line field and the cursor is not positioned on the top line, pressing <PgUp> moves the cursor *up* one page in the current field.

TABLE 2. **DOS and Windows mouse mappings**

Action	*Mouse*	*Description*
Choose	<Left-down-click>	If the end user is on the window's title bar, pressing this button moves the window. If the end user is on the window's border, pressing this button sizes the window. Otherwise, pressing the left mouse button selects the field positioned under the mouse cursor (if the field is selectable).
Mark	<Left-drag>	If the current field is a field that can be edited, holding the left button down and dragging the mouse specifies the mark location.
Select	<Left-release>	If the current field is a field that can be edited, releasing this button completes the mark specification. Otherwise, releasing this button completes the select operation.

OSF/Motif and Curses

OSF/Motif uses user-definable "soft" mappings to map actions to the keyboard and mouse.

Curses keyboard mappings differ significantly from other Zinc-supported operating environment. For example, many terminals supported by Curses define few or no regular function keys. Curses does not support an <Alt> or <meta> key, and you may not be able to use <Ctrl> and <Shift> keys with any function keys or keys like <pgup>, <end>, or <tab>.

On the other hand, some terminals used supported by Curses have specialized keys for actions such as cut, paste, restore, or cancel. If a terminal does not have these, modify the **terminfo** database entry to create mappings from existing keys to the required functions. By default, Zinc maps special Curses function keys to Zinc events. These special Curses function keys include KEY_ENTER, KEY_NPAGE, KEY_BTAB, KEY_CANCEL, KEY_MARK. Additionally, <Alt>, <Ctrl>, and <Shift> keys are not used, and hot key sequences are preceded by <Esc> rather than <Alt>. Zinc's default keyboard mapping allows <Ctrl> key combinations for many events, shown in the following table.

If desired, Zinc provides a mode somewhat compatible with the PC keyboard that uses surrogate <Alt> and <Ctrl> keys with <F1> through <F10>. In this mode, the '`' key is the alt key, and '~' is used as the <Ctrl> key. Pressing a surrogate key followed by another key causes Zinc to recognize an <Alt> or <Ctrl> sequence. Pressing a surrogate key twice causes the key to be recognized as normal. So instead of pressing Curses's *KEY_SCANCEL* to close a window, a user could press '`,' followed by <F4>. Select this mode at compile time by using the *ZIL_PC_KEYBRD* preprocessor flag.

TABLE 3. Curses's <Ctrl> key combinations

Action	*Mouse*	*Description*
Backspace	<CTRL-H>	Move the cursor one character to the left.
Close window	<CTRL-D>	Closes the top or current window.
Cut	<CTRL-T>	Cuts the marked portion of the current window field. The cut section is removed and stored on the paste buffer. This key only affects fields that can be edited.
Help, context-specific	<CTRL-E>	Displays context-sensitive help information for the current window object.
Help, general	<CTRL-P>	Displays general help information for the application.
Make next window current	<CTRL-N>	Makes the next window current.
Mark	<CTRL-K>	If the current field is a field that can be edited, holding the left button down and dragging the mouse specifies the mark location.
Move window	<CTRL-V>	Moves the current window when followed by any movement key and then <Enter>. When followed by any movement key and then <Esc>, the selected window is returned to its original position.
Paste	<CTRL-P>	Retrieves the cut section from the paste buffer and pastes it in the current field. This key only affects fields that can be edited.
Refresh display	<CTRL-L>	Redraws all windows and window objects.
Restore window	<CTRL-O>	Restores the original size of the window.
Size window	<CTRL-B>	Sizes, relative to the top left corner, the current window when followed by any movement key.

Macintosh

TABLE 4. Macintosh keyboard mappings

Action	*Key*	*Description*
Copy	<Cmd-c>	Copies the marked portion of the current window field and stores it on the Clipboard. This key only affects fields that can be edited.
Cut	<Cmd-x>	Cuts the marked portion of the current window field. The cut section is removed and stored on the Clipboard. This key only affects fields that can be edited.
Delete previous character	<Backspace>	Moves the cursor *left* one position, deleting the character immediately to the left of the cursor. This key only affects fields that can be edited and only where the cursor is not in the field's first character position.
Close window	<Cmd-w>	Closes the top or current window.
Down	<↓> <Gray+↓>	If the field is a multi-line field and the cursor is not positioned on the bottom line, pressing <↓> moves the cursor *down* one line on the display.
Quit	<Cmd-q>	Exits the application, closing all open windows.
Left	<←> <Gray+←>	If the cursor is not positioned in the *first* character position of a field, pressing <←> moves the cursor one character to the left.
Next field	<Tab>	Moves from the current field to the *next* selectable field. If the last window field is currently selected, pressing <Tab> cycles to the *first* selectable window field.
Paste	<Cmd-v>	Retrieves the cut or copied section from the Clipboard and pastes it in the current field. This key only affects fields that can be edited.
Previous field	<Shift+Tab>	Moves from the current (or selected) window field to the *previous* selectable window field. If the first window field is currently selected, pressing <BackTab> cycles to the *last* selectable window field.
Right	<→>	If the cursor is not positioned in the *last* character of a field, pressing <→> moves the cursor one character to the right.
Up	<↑> <Gray ↑>	If the field is a multi-line field and the cursor is not positioned on the top line, pressing <↑> moves the cursor *up* one line on the display.
Help	<Cmd ?>	Brings up the help context assigned to the current field.
Mark	<Shift+↑> <Shift+↓> <Shift+←> <Shift+→>	Marks text in an editable field or selection of child items in lists.

TABLE 5. Macintosh mouse mappings

Action	Mouse	*Description*
Choose	<Click>	If the mouse cursor is on the window's title bar, pressing the button moves the window. If the mouse cursor is on the window's size box, pressing this button sizes the window. Otherwise, pressing the left mouse button selects the field positioned under the mouse cursor, if the field is selectable.
Choose mark	<Drag>	If the current field is a field that can be edited, holding the left button down and dragging the mouse specifies the mark location.
Select	<Release>	If the current field is a field that can be edited, releasing the button completes the mark specification. Otherwise, releasing the button completes the select operation.

NEXTSTEP

TABLE 6. NEXTSTEP keyboard mappings

Action	Key	*Description*
Copy	<Cmd-c>	Copies the marked portion of the current window field and stores it in a global paste buffer. This key only affects fields that can be edited.
Cut	<Cmd-x>	Cuts the marked portion of the current window field. The cut section is removed and stored on the Pasteboard. This key only affects fields that can be edited.
Delete previous character	<Backspace>	Moves the cursor *left* one position, deleting the character to the character immediately to the left of the cursor. This key only affects fields that can be edited and only where the cursor is not in the field's first character position.
Close window	<Cmd-w>	Closes a window.
Down	<↓>	If the text field is a text object, and the cursor is not positioned on the bottom line, pressing <↓> moves the cursor *down* one line on the display.
Quit	<Cmd-q>	Exits the application.
Left	<←>	If the cursor is not positioned in the *first* character position of a field, pressing <Left-Arrow> moves the cursor one character to the left.
Minimize	<Cmd-m>	Minimizes the current window. This key only affects the window when the current window can be sized and if it is not already in a minimized state.

TABLE 6. **NEXTSTEP keyboard mappings**

Action	Key	*Description*
Next field	<Tab>	Moves from the current field to the *next* selectable field. If the last window field is currently selected, pressing <Tab> cycles to the *first* selectable window field.
Paste	<Cmd-v>	Retrieves the cut section from the Pasteboard and pastes it in the current field. This key only affects fields that can be edited.
Previous field	<Shift+Tab>	Moves from the current window field to the *previous* window field. If the first window field is currently selected, pressing <Shift+Tab> cycles to the *last* selectable window field.
Right	<→>	If the cursor is not positioned in the *last* character position of a left-hand field, pressing <Right-Arrow> moves the cursor one character to the right.
Up	<↑> <Gray ↑>	If the text field is a text object, and the cursor is not positioned on the top line, pressing <Up-arrow> moves the cursor *up* one line on the display.

NOTE: In order to use **Cut**, **Copy**, and **Paste**, your **Edit** menu must include **Cut**, **Copy**, and **Paste**. Further, to use **<Tab>** and **<Shift-Tab>**, your Zinc application must use **select:previousText** and **select:nextText**.

TABLE 7. **NEXTSTEP mouse mappings**

Action	Mouse	*Description*
Choose	<Left-click>	If the mouse cursor is on the window's title bar, pressing this button moves the window. If the mouse cursor is on the window's border, pressing this button sizes the window. Otherwise, pressing the left mouse button selects the field positioned under the mouse cursor (if the field is selectable).
Choose mark	<Left-drag>	If the current field is an editable field, holding the left button down and dragging the mouse specifies the mark location.
Select	<Left-release>	If a *choose mark* operation is in progress, releasing this button completes the mark specification. Otherwise, releasing this button completes the select operation.
Display menu bar	<Right-click>	If the right mouse button is enabled in the **Preferences.app** application, pressing and holding this button will display the current application's menu bar at the position of the mouse cursor.