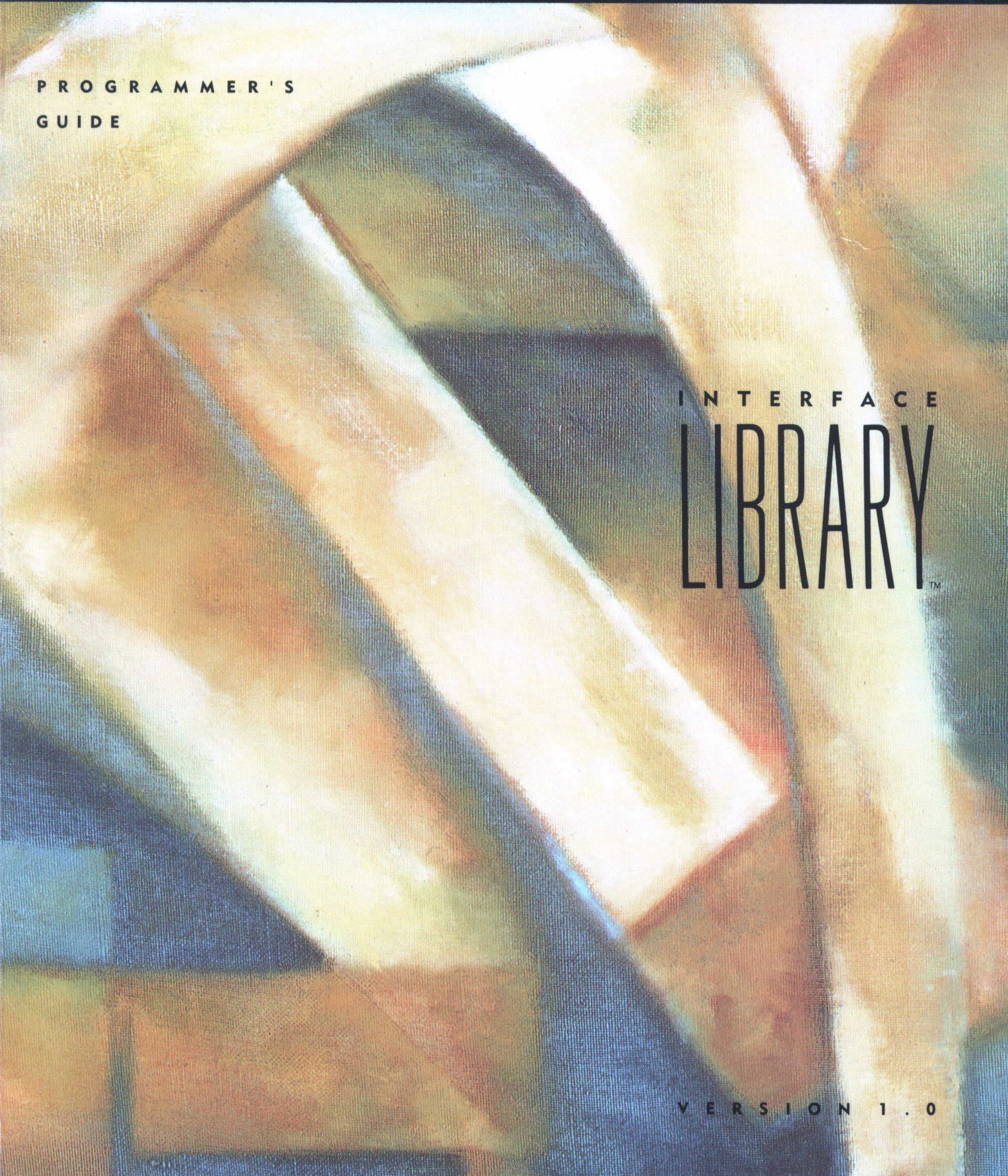
Z i n c™

PROGRAMMER'S GUIDE



INTERFACE LIBRARY™

VERSION 1.0

Zinc™ Interface Library™

Programmer's Guide

Version 1.0

Zinc Software Incorporated
Pleasant Grove, Utah

TABLE OF CONTENTS

INTRODUCTION

The Zinc Interface Library is a powerful user interface library that uses unique features of C++, including virtual functions, class inheritance, operator overloading, multiple inheritance, etc. This library is developed specifically for C++ and is compatible with Borland International's Turbo C®++ (which supports AT&T's C++ V2.0 and ANSI C).

System requirements

To develop applications, you need Turbo C++, DOS 2.1 or later (DOS 3.1 or later is recommended), 640K RAM and a hard disk drive. For mouse support, you need a Microsoft® mouse compatible driver.

Shipping applications

To ship applications, you must include the following run-time files:

- Turbo C++ BGI files (if your application runs in graphics mode),
- Any files generated by the GENHELP.EXE program. (This program generates help screens used by the help window system.)

Suggested reading

The use of this product assumes a working knowledge of C++. Some books that introduce the C++ programming language are:

- Dewhurst, Stephen C. and Stark, Kathy T. *Programming in C++*, Englewood Cliffs, New Jersey: Prentice Hall, 1989, 233 pages.
- Eckel, Bruce. *Using C++*. Berkeley, CA: Osborne/McGraw-Hill, 1989, 617 pages.
- Hansen, Tony L. *The C++ Answer Book*, Reading, MA: Addison-Westley, 1990, 578 pages.
- Lippman, Stanley B. *C++ Primer*. Reading, MA: Addison-Westley, 1989, 464 pages.
- Pohl, Ira. *C++ for C Programmer's*. Redwood City, CA: Benjamin/-Cummings Publishing, 1989, 244 pages.

- Stevens, Al. *Teach Yourself C++*. Portland, OR: MIS Press, 1990, 272 pages.

- Stroustrup, Bjarne. *The C++ Programming Language*. Reading, MA: Addison-Westley, 1986, 328 pages.

- *Turbo C++, Getting Started*. Scotts Valley, CA: Borland International, 1990, 268 pages.

- Wiener, Richard S. and Pinson, Lewis J. *An Introduction to Object Oriented Programming and C++*. Reading, MA: Addison-Westley, 1989, 273 pages.

Programmer's Guide

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

Installation—This section (Chapter 1) tells how to install the library software on your machine.

Conceptual Design—This section (Chapter 2) gives a high-level description of the Zinc Interface Library, including the conceptual operation of the library and its major components.

Window Objects—This section (Chapter 3) describes the types of window objects supported by the library. It also discusses the proper use of window objects in an application program.

Default Input Mapping—This section (Chapter 4) describes the default mapping of keyboard and mouse information.

Default Color Mapping—This section (Chapter 5) describes the default color combinations of windows and window objects.

Tutorials—This section (Chapter 6) provides 5 tutorials that help you get started writing application programs that use the Zinc Interface Library.

Programmer's Reference

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

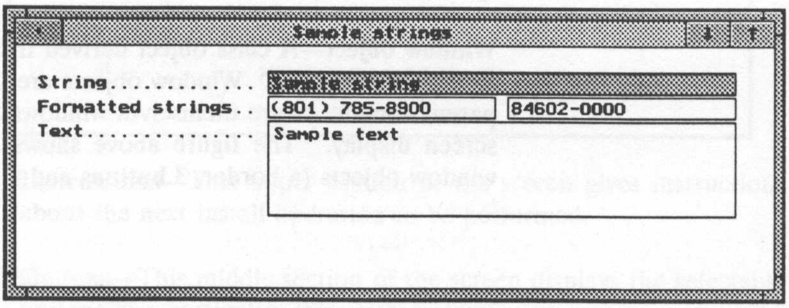
Class object information—This section (Introduction) contains the class hierarchy, header file information and global variables associated with class objects and structures available within the Zinc Interface Library.

Class object references—This section (Chapters 2 through 48) contains short descriptions about the class objects (or structures), the available member variables and functions and the calling conventions used with the class object.

Terminology

The following terms are used extensively throughout the documentation:

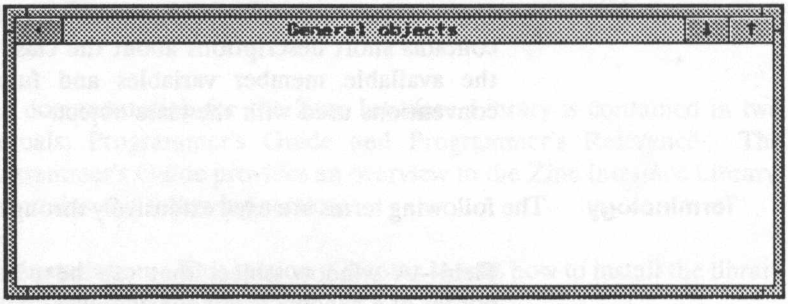
Field—A window object that can be edited. For example, the border of a window is not considered a "field" whereas a number is considered a field. The figure below shows a window with several fields (the fields are shown with outlining borders):



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

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

Window—A region of the screen that contains one or more window objects. A window is used by the end-user to view or edit information associated with the application program. A window is represented by the UIW_WINDOW class object. In the figure below, the window is shown as the main rectangle and all blank portions within the rectangle. All non-blank portions of the window are window objects (the border, buttons and title bar).



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

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

CHAPTER 1 – INSTALLATION

System requirements

Installation of the Zinc Interface Library requires DOS 2.1 or later (DOS 3.1 or later is recommended), 640K RAM and a hard disk drive.

Introduction

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

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

INSTRUCTIONS

- -

OPTIONS

- -

KEYBOARD INTERFACE

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

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

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

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

Installation procedure

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

a:<Enter>
install<Enter>

The install process is accomplished in five steps:

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

Selecting a drive—You will be asked to select a drive to which you want to install. (Be sure to select a hard disk drive.)

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

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

Demo—A program that demonstrates the features and capabilities of the library.

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

Include Files—Program header files used by the library class objects.

Utility Programs—Application programs that are used with the Zinc Interface Library. For example, one utility program is used to generate help files. This program is called GENHELP.EXE.

Library Files—Files that contain the compiled library class objects. These files include small, medium, compact and large models.

Tutorials—Sample programs that give hands-on experience in using the Zinc Interface Library. These programs are explained in "Chapter 6—Tutorials" of this manual.

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

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

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

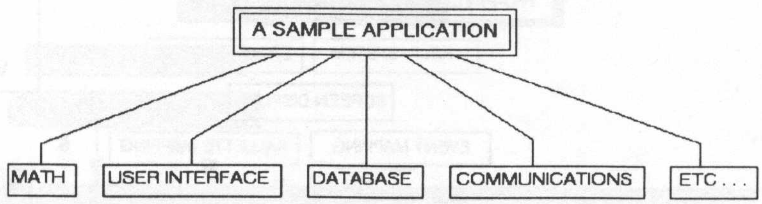
When the installation is complete, a message appears on your screen indicating that you are now free to use the Zinc Interface Library.

CHAPTER 2 – CONCEPTUAL DESIGN

Every computer application has special needs. For instance, a particular application may require:

- A user interface to present information to the user, report run-time errors and give meaningful help information.
- Access to a database for information storage and retrieval.
- A communications package for modem and serial line support.
- Special math capabilities for statistical modeling or advanced mathematical operations.

PIECES OF AN APPLICATION

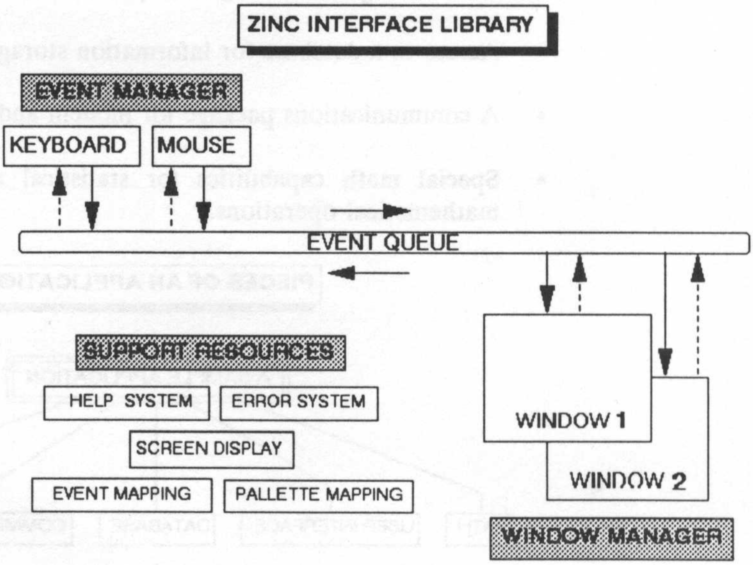


The Zinc Interface Library is a user interface tool that supports programmers with their user interface needs. It is an object-oriented class library, implemented in C++. The main goals of this product are to provide:

Consistency—This means consistency between graphics and text modes of operation, consistency between class objects (e.g., their parameter passing and modes of operation) and consistency in the documentation.

Ease-of-use—This includes a run-time presentation that is easily understood by end-users as well as library code that is quickly learned by programmers. One ease-of-use aspect is achieved through consistency, another through a conceptual design that is easily understood.

Flexibility—The C++ object orientation, combined with the hierarchal method discussed below, provides reusability of code. This design also provides clear points of entry where programmers can derive new objects to customize the run-time operation of their application programs. The Zinc Interface Library product goals are accomplished through a simple, yet powerful, design and implementation scheme (shown below).



The main sections of the library are:

Event manager—This portion of the library controls the flow of end-user input and system messages throughout the library.

Window manager—This portion of the library controls the presentation of windows and window objects to the screen display.

Screen display—This library resource provides low-level graphics or text screen display support.

Help system—This library resource controls the presentation of help information during the run-time operation of an application program.

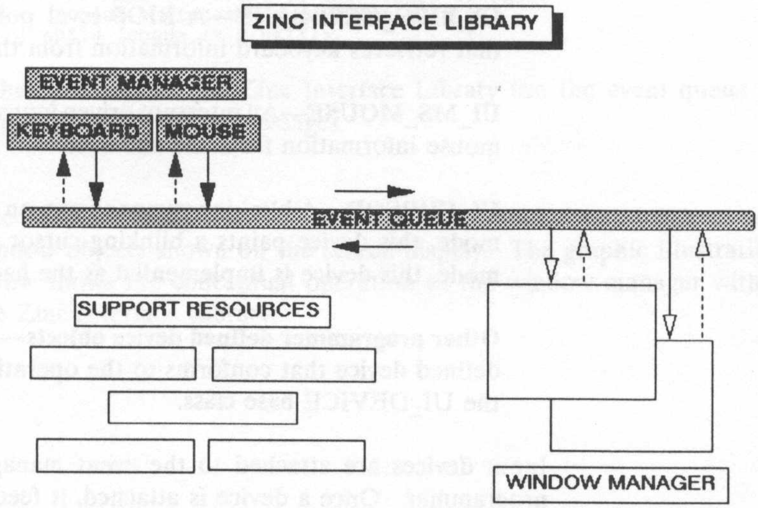
Error system—This library resource controls the presentation of error information during the run-time operation of an application program.

Event mapping—This library resource controls the mapping of raw device events (e.g., keyboard and mouse) to logical system events (e.g., sizing, moving, redrawing).

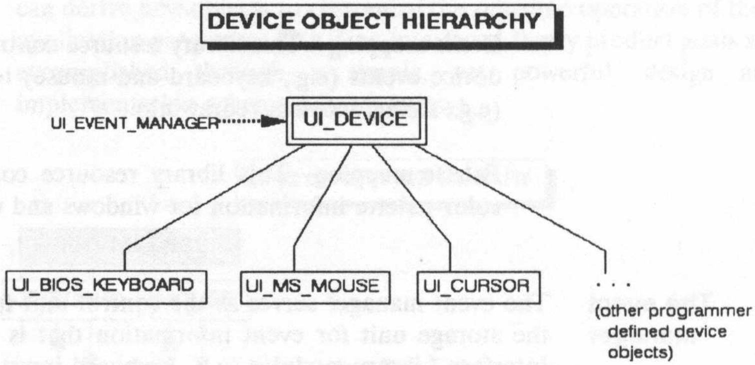
Palette mapping—This library resource controls the mapping of color palette information for windows and window objects.

The event manager

The event manager serves as the control unit for input devices and as the storage unit for event information that is processed by the Zinc Interface Library modules (e.g., keyboard input information as well as system messages). The graphic illustration below shows the conceptual operation of the event manager within the library:



Most compiler libraries have a set of functions to get input information from the keyboard (e.g., **getch()**, **getchar()**) but seldom have functions to get information from other devices, such as a mouse. In addition, the integration of multiple input devices is left to the programmer. With the Zinc Interface Library, all input devices (e.g., keyboard, mouse) are integrated to provide smooth control of the user's input. This interface is handled by the control portion of the event manager. The following device object hierarchy is understood by the UI_EVENT_MANAGER class object:



Classes derived from the UI_DEVICE base class include:

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

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

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

Other programmer defined device objects—Any other programmer defined device that conforms to the operating protocol defined by the UI_DEVICE base class.

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

```
// Construct the screen display.
UI_DOS_TEXT_DISPLAY display;

// Construct the event manager.
UI_EVENT_MANAGER eventManager(100, &display);
```

```
// Add in the input devices.
eventManager
    + new UI_BIOS_KEYBOARD
    + new UI_MS_MOUSE
    + new UI_CURSOR;
```

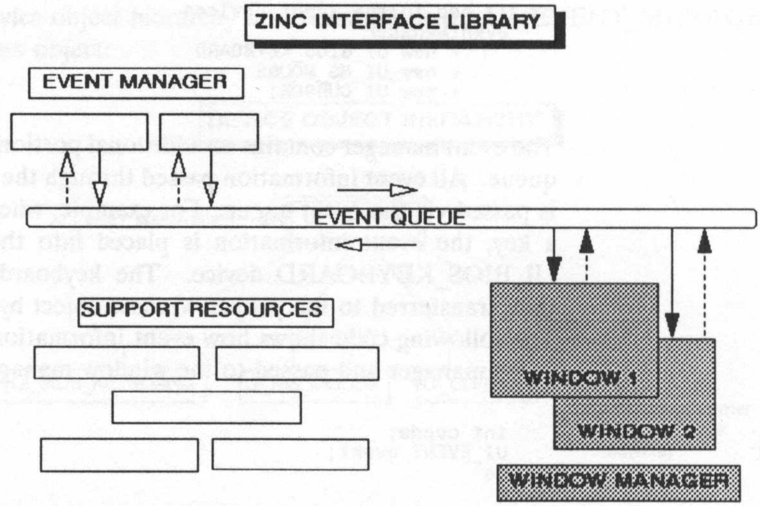
The event manager contains an additional portion identified as the event queue. All event information passed through the Zinc Interface Library is passed via the event queue. For example, when the end-user presses a key, the event information is placed into the event queue by the UI_BIOS_KEYBOARD device. The keyboard event information is then transferred to the proper window object by the window manager. The following code shows how event information is retrieved from the event manager and passed to the window manager:

```
int ccode;
UI_EVENT event;
do
{
    // Get an event from the event manager.
    eventManager.Get(event, Q_NORMAL);

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

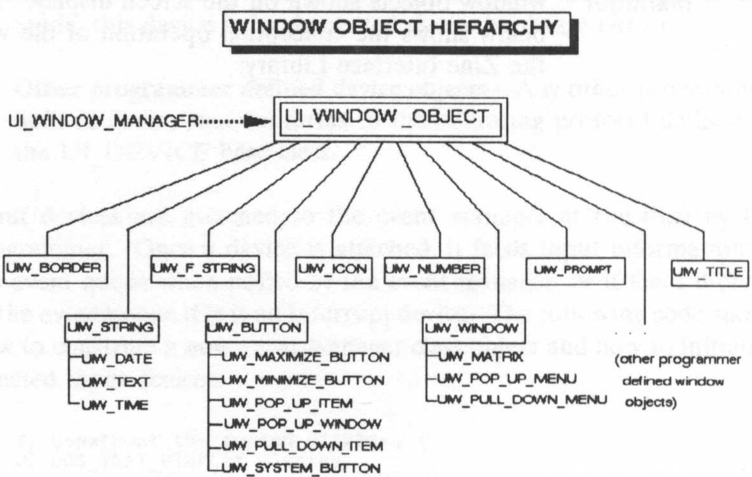
Other portions of the Zinc Interface Library use the event queue to send system or private messages.

The window manager

The window manager serves as the control module for all windows and window objects shown on the screen display. The graphic illustration below shows the conceptual operation of the window manager within the Zinc Interface Library:



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



Classes derived from the UI_WINDOW_OBJECT base class include:

UIW_BORDER—An outlining border drawn around a window.

UIW_STRING—A field used to enter, display, or modify an ascii string buffer.

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

UIW_TEXT—A field used to enter, display, or modify a word-wrapped text buffer.

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

UIW_FORMATTED_STRING—A field used to enter, display, or modify an ascii string buffer that contains literal characters, or characters that cannot be edited (e.g., phone numbers, social security numbers).

UIW_BUTTON—A rectangular region of the screen that, when selected, performs run-time operations specified by the programmer.

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

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

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

UIW_POP_UP_WINDOW—An item that, when selected, displays additional window information (in the form of a sub-window) to the screen display.

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

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

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

UIW_NUMBER—A field used to enter, display, or modify numeric information. This object supports both integer values (e.g., short, int, long) and real values (e.g., float, double).

UIW_WINDOW—A rectangular region of the screen that is composed of one or more class objects derived from the UI_WINDOW_OBJECT base class.

UIW_MATRIX—A two-dimensional list of related items. These items are organized in a row/column fashion and may be any of the objects described in the window object hierarchy.

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

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

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

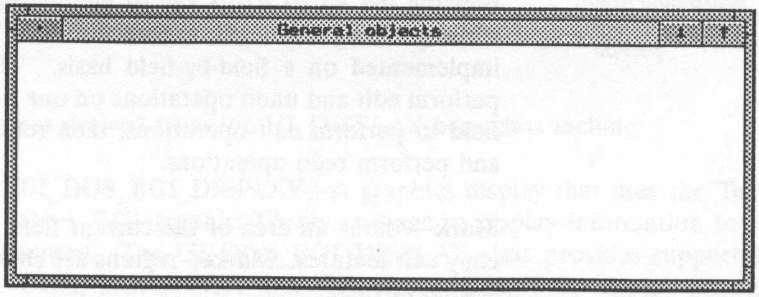
UIW_TITLE—An object that occupies the top region of a window and contains a window's title information.

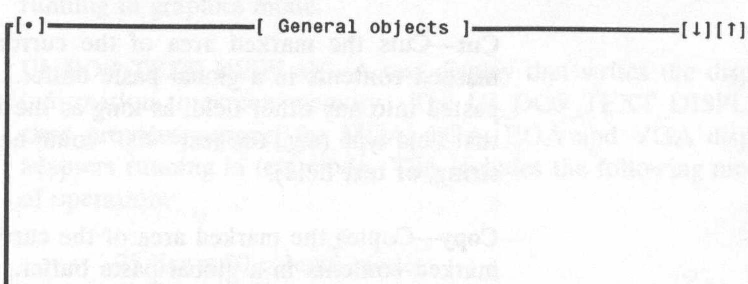
Other programmer defined window objects—Any other programmer defined window object that conforms to the operating protocol defined by the UI_WINDOW_OBJECT base class.

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

```
// Construct the screen display.
UI_DOS_TEXT_DISPLAY display;

// Construct the event manager.
UI_EVENT_MANAGER eventManager(100, &display);
eventManager + new UI_BIOS_KEYBOARD + new UI_MS_MOUSE;

// Construct the window manager.
UI_WINDOW_MANAGER windowManager(&display, &eventManager);

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

Windows and window objects have distinct representations in graphics and text modes of operation. The code shown above would produce the following graphic and textual representations of a simple window:





Window objects that can be edited (UIW_DATE, UIW_-FORMATTED_STRING, UIW_NUMBER, UIW_STRING, UIW_-TIME and UIW_TEXT) support the following features:

Undo—Allows the end-user to roll an editing operation backward. For example, if an end-user accidently deleted a complete line in a text field, the information could be retrieved by pressing the <Ctrl F9> key or by holding the <Alt> key down while pressing the left mouse button. The undo operation is implemented on a field-by-field basis. Thus, an end-user could perform edit and undo operations on one field, move to a different field to perform edit operations, then return to the original field to perform additional edit or undo operations.

Redo—Allows the end-user to roll an editing operation forward (restore information removed with the undo operation). For example, the undo operation (described above) explained how an end-user may accidently delete a complete line in a text field. If the user continued to perform undo operations, then decided some of the old text was worthwhile, the information could be retrieved by pressing the <Ctrl F10> key or by holding the <Alt> key down while pressing the right mouse button. The redo operation is implemented on a field-by-field basis. Thus, an end-user could perform edit and undo operations on one field, move to a different field to perform edit operations, then return to the original field and perform redo operations.

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

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

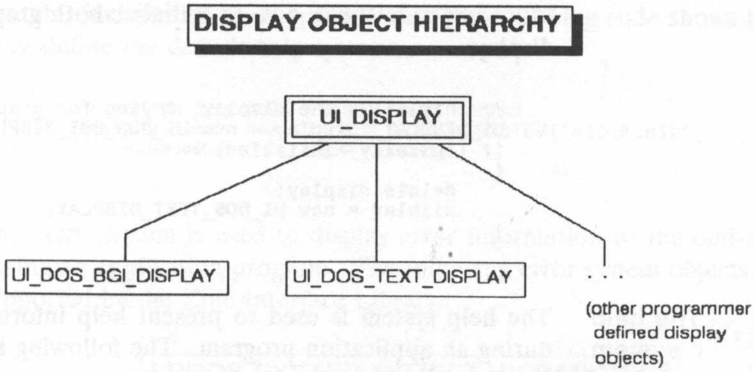
Copy—Copies the marked area of the current field and stores the marked contents in a global paste buffer. This data can later be pasted into any other field, as long as the information is valid for that field type.

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

For more information about window objects see "Chapter 3—Window Objects" of this manual.

The screen display

The screen display is used to control all low-level screen output. The following display objects are supported by the Zinc Interface Library:



Classes derived from the UI_DISPLAY base class include:

UI_DOS_BGI_DISPLAY—A graphics display that uses the Turbo C++ BGI graphics library package to display information to the screen. The UI_DOS_BGI_DISPLAY class provides support for CGA, EGA, VGA and Hercules monochrome display adapters running in graphics mode.

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

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

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

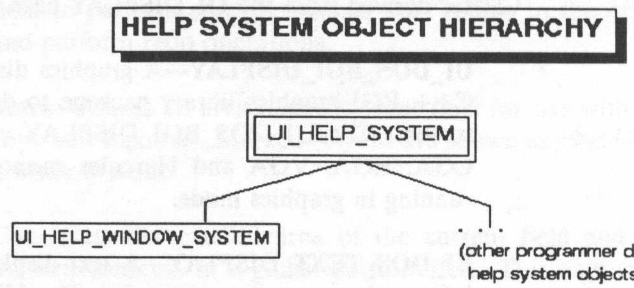
Other programmer defined screen display objects—Any other programmer defined display object that conforms to the operating protocol defined by the UI_DISPLAY base class.

The use of multiple display classes allows the application program to be abstract in its screen display. Thus, one set of source code can be used to produce output for both graphics- and text-based environments. The following code shows how to initialize both graphic and textual screen displays:

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

The help system

The help system is used to present help information to the end-user during an application program. The following help system objects are supported by the Zinc Interface Library:



Help system classes supported by the Zinc Interface Library include:

UI_HELP_SYSTEM—A help system stub that does not present help information to the end-user. This class object is used as the base class for other help system objects.

UI_HELP_WINDOW_SYSTEM—A help system that uses the Zinc Interface Library windowing system to present help information to the end-user.

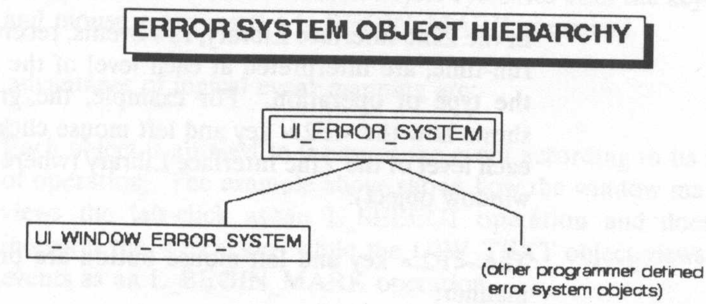
Other programmer defined help system objects—Any other programmer defined help system object that conforms to the operating protocol defined by the UI_HELP_SYSTEM base class.

The Zinc Interface Library initially installs a UI_HELP_SYSTEM object as the default help system. The window help system is not initialized so that programmers are not forced to have the window object code modules included in their application. The following code shows how to re-define the default help system:

```
// Add in the full window help system.
_helpSystem = new UI_HELP_WINDOW_SYSTEM("clock.hlp",
    windowManager, HELP_CLOCK);
```

The error system

The error system is used to display error information to the end-user during an application program. The following error system objects are supported by the Zinc Interface Library:



Error system classes supported by the Zinc Interface Library include:

UI_ERROR_SYSTEM—An error system stub that does not present error information to the end-user. This class object is used as the base class for other error system objects.

UI_ERROR_WINDOW_SYSTEM—An error system that uses the Zinc Interface Library windowing system to present error information to the end-user.

Other programmer defined error system objects—Any other programmer defined error system object that conforms to the operating protocol defined by the UI_ERROR_SYSTEM base class.

The Zinc Interface Library initially installs a UI_ERROR_SYSTEM object as the default error system. The window error system is not initialized so that programmers are not forced to have the window object code modules included in their application. The following code shows how to re-define the default error system:

```
// Add in the full window error system.
_errorSystem = new UI_ERROR_WINDOW_SYSTEM;
```

Event mapping

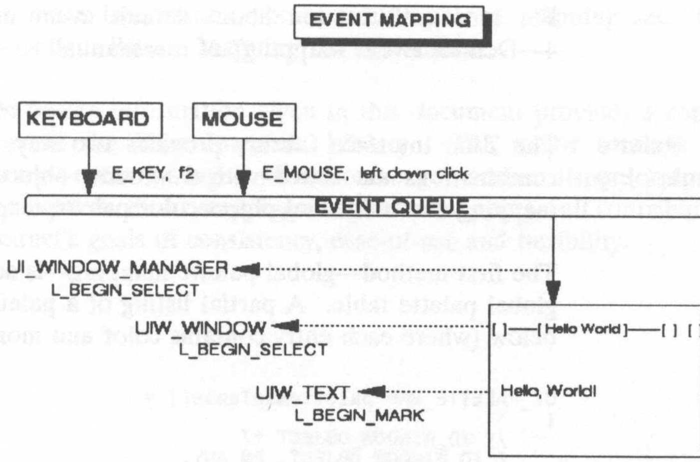
Many user interface libraries convert raw input information to logical information when they are received from the input device. For example, a mouse device may define the left mouse button click to be the select operation (M_SELECT). These implementations allow only one logical mapping of a given raw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

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

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

- first, the key or mouse information is received by the input device (i.e., UI_BIOS_KEYBOARD and UI_MS_MOUSE) and placed in the event queue.

- second, the window manager evaluates the event and passes it to the proper window. The mouse event is interpreted as an L_BEGIN_-SELECT logical event, while the keyboard event is passed directly to the window.



- third, the window evaluates the event and passes it to the proper window object. The mouse event is interpreted as an L_BEGIN_-SELECT logical event, while the keyboard event is passed directly to the UIW_TEXT window object.

- finally, the UIW_TEXT window object evaluates both the keyboard and mouse events as the L_BEGIN_MARK command.

The advantages of logical event mapping are:

- Each object is allowed to interpret the event according to its mode of operation. The example above shows how the window manager views the left-click as an L_SELECT operation and does not interpret the <F2> key, while the UIW_TEXT object views both events as an L_BEGIN_MARK operation.

- The programmer can define additional input devices that generate their own raw event information. This method allows the programmer to define new input devices that generate specialized raw codes. With this implementation, programmers can define logical event mapping for the Zinc Interface Library but still receive all the raw event information for their specific application program.

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

For more information about default event mapping see "Chapter 4—Default Event Mapping" of this manual.

Palette mapping

The Zinc Interface Library provides two ways of defining the color combinations associated with a window object: global color palette mapping and individual object color palette mapping.

The first method—global palette mapping—is accomplished through a global palette table. A partial listing of a palette map table is shown below (where each entry contains color and monochrome attributes):

```
UI_PALETTE_MAP paletteMapTable[] =
{
    /* ID_WINDOW_OBJECT */
    { ID_WINDOW_OBJECT, PM_ANY,
        { ' ', attrib(WHITE, LIGHTGRAY),
        attrib(MONO_NORMAL, MONO_BLACK),
        SOLID_FILL, attrib(BLACK, WHITE),
        attrib(BW_BLACK, BW_WHITE),
        attrib(GS_BLACK, GS_WHITE) } },

    /* ID_BORDER */
    { ID_BORDER, PM_ANY,
        { ' ', attrib(LIGHTRED, LIGHTGRAY),
        attrib(MONO_HIGH, MONO_BLACK),
        SOLID_FILL, attrib(WHITE, LIGHTGRAY),
        attrib(BW_BLACK, BW_WHITE),
        attrib(GS_WHITE, GS_GRAY) } },
        .
        .
        .
};
```

The second method—individual object color palette mapping—is accomplished by setting the palette table pointer associated with a particular window object. This allows the programmer to define a specific instance of a window object but does not affect the overall presentation of the window object. The example below shows how to re-define the palette table associated with a particular window object:

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

// Redefine the colors for this window.
extern UI_PALETTE_MAP *_myPaletteTable;
window->paletteTable = _myPaletteTable;
*windowManager + window;
```

For more information about default palette mapping see "Chapter 5—Default Palette Mapping" of this manual.

The design information given in this document provides a conceptual view of the Zinc Interface Library. The major sections of this library—event manager, window manager, screen display, help system, error system, event mapping and palette mapping—all contribute to the product's goals of consistency, ease-of-use and flexibility.

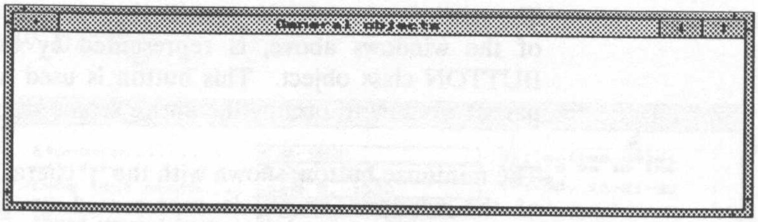
CHAPTER 3 – WINDOW OBJECTS

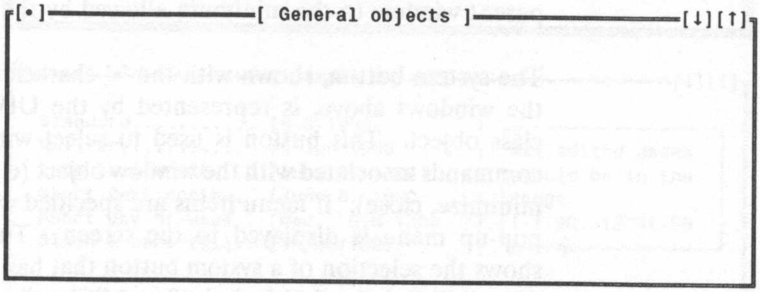
Introduction

"Chapter 2—Conceptual Design" of this manual briefly describes the types of window objects that are available with the Zinc Interface Library. This chapter shows the graphic, textual and code implementations of all the supported window class objects. It also gives a more complete description of each window object along with its normal modes of operation.

Basic window objects

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






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

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

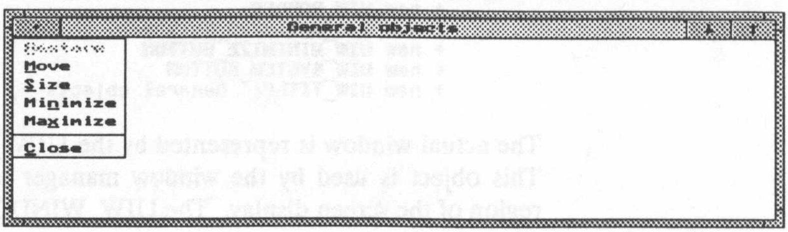
The window's border, shown as the exterior part of the windows above, is represented by the UIW_BORDER class object. If the application is running in graphics mode, the border is shown as a 3-dimensional shaded region drawn around the window. If the application is running in text mode and the window is at the forefront of the screen (the current window) then the border is shown as a double line. Otherwise, the border is shown as a single line.

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

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

The minimize button, shown with the '↓' character on the top-right side of the windows above, is represented by the UIW_MINIMIZE_-BUTTON class object. This button is used to reduce the size of its parent window to the minimum allowed by the window.

The system button, shown with the '•' character on the top-left side of the windows above, is represented by the UIW_SYSTEM_BUTTON class object. This button is used to select window or system specific commands associated with the window object (e.g., size, move, maximize, minimize, close). If menu items are specified with the system button, a pop-up menu is displayed to the screen. The graphic image below shows the selection of a system button that has the "Restore," "Move," "Size," "Minimize," "Maximize" and "Close" system button options:

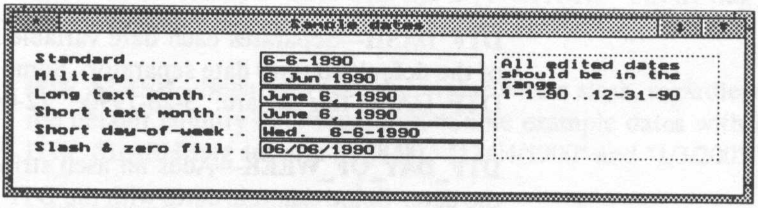


For more information about the basic window objects discussed above see:

> "Chapter 48—UIW_WINDOW,"
> "Chapter 26—UIW_BORDER,"
> "Chapter 47—UIW_TITLE,"
> "Chapter 32—UIW_MAXIMIZE_BUTTON,"
> "Chapter 33—UIW_MINIMIZE_BUTTON," or
> "Chapter 44—UIW_SYSTEM_BUTTON"

of the Programmer's Reference.

Date window objects Date fields should be used anytime date information is presented to the end-user or when date information is to be entered at an application's run-time. The figures below show graphic, textual and code implementations of a window with several variations of the date class object (UIW_DATE):




```
[•]─────────────────[ Sample dates ]─────────────────[↓][↑]
  Standard.......... [6-6-1990          ]  All edited dates
  Military.......... [6 Jun 1990        ]  should be in the
  Long text month.... [June 6, 1990     ]  range
  Short text month... [June 6, 1990     ]  1-1-90..12-31-99
  Short day-of-week.. [Wed.  6-6-1990   ]
  Slash & zero fill. [06/06/1990        ]
```

```
*window
    + new UIW_TITLE(" Sample dates ", WOF_JUSTIFY_CENTER)
    + new UIW_TEXT(43, 1, 20, 6, "All edited dates should be in
        the range 1-1-90..12-31-99", 128, TXF_NO_FLAGS,
        WOF_VIEW_ONLY | WOF_NON_SELECTABLE | WOF_BORDER)

    + new UIW_PROMPT(2, 1, "Standard..........", WOF_NO_FLAGS)
    + new UIW_DATE(22, 1, 20, &date, "1-1-90..12-31-99",
        DTF_SYSTEM, WOF_BORDER)

    + new UIW_PROMPT(2, 2, "Military..........", WOF_NO_FLAGS)
    + new UIW_DATE(22, 2, 20, &date, "1-1-90..12-31-99",
```

```
            DTF_MILITARY_FORMAT | DTF_SYSTEM, WOF_BORDER)
        + new UIW_PROMPT(2, 3, "Long text month....", WOF_NO_FLAGS)
        + new UIW_DATE(22, 3, 20, &date, "1-1-90..12-31-99",
            DTF_ALPHA_MONTH | DTF_SYSTEM, WOF_BORDER)
        + new UIW_PROMPT(2, 4, "Short text month...", WOF_NO_FLAGS)
        + new UIW_DATE(22, 4, 20, &date, "1-1-90..12-31-99",
            DTF_SHORT_MONTH | DTF_SYSTEM, WOF_BORDER)
        + new UIW_PROMPT(2, 5, "Short day-of-week..", WOF_NO_FLAGS)
        + new UIW_DATE(22, 5, 20, &date, "1-1-90..12-31-99",
            DTF_SHORT_DAY | DTF_SYSTEM, WOF_BORDER)
        + new UIW_PROMPT(2, 6, "Slash & zero fill..", WOF_NO_FLAGS)
        + new UIW_DATE(22, 6, 20, &date, "1-1-90..12-31-99",
            DTF_SLASH | DTF_ZERO_FILL | DTF_SYSTEM, WOF_BORDER);
```

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

DTF_ALPHA_MONTH—Shows the month as an ascii string value. Some example dates with the DTF_ALPHA_MONTH flag set are: "March 28, 1990," "December 4, 1980" and "January 3, 2003."

DTF_DASH—Separates each date variable with a dash, regardless of the default country date separator. Some example dates with the DTF_DASH flag set are: "3-28-1990," "12-04-1980" and "1-3-2003."

DTF_DAY_OF_WEEK—Adds an ascii string day-of-week value to the date. Some example dates with the DTF_DAY_OF_WEEK flag set are: "Wednesday March 28, 1990," "Thursday December 4, 1980" and "Saturday January 3, 2003."

DTF_EUROPEAN_FORMAT—Forces the date to be shown in the European format (i.e., *day/month/year*), regardless of the default country information. Some example dates with the DTF_-EUROPEAN_FORMAT flag set are: "28/3/1990," "4 December, 1980" and "3 Jan., 2003."

DTF_JAPANESE_FORMAT—Forces the date to be shown in the Japanese format (i.e., *year/month/day*), regardless of the default country information. Some example dates with the DTF_-JAPANESE_FORMAT flag set are: "1990/3/28," "1980 December 4" and "2003 Jan. 3."

DTF_MILITARY_FORMAT—Forces the date to be shown in the U.S. Military format (i.e., *day/month/year* where *month* is a 3 letter abbreviated word), regardless of the default country information. Some example dates with the DTF_MILITARY_FORMAT flag set (army style) are: "28 Mar 1900," "04 Dec 1980," and "03 Jan 2003." Some example dates with the DTF_MILITARY and DTF_-UPPER_CASE flags set (navy style) are: "28 DEC 1900," "04 DEC 1980," and "03 JAN 2003."

DTF_SHORT_DAY—Adds a shortened day-of-week value to the date. Some example dates with the DTF_SHORT_DAY flag set are: "Wed. March 28, 1990," "Thurs. December 4, 1980" and "Sat. January 3, 2003."

DTF_SHORT_MONTH—Adds a shortened alphanumeric month value to the date. Some example dates with the DTF_SHORT_-MONTH flag set are: "Mar. 28, 1990," "Dec. 4, 1980" and "Jan. 3, 2003."

DTF_SHORT_YEAR—Forces the year to be shown as a 2 digit value. Some example dates with the DTF_SHORT_YEAR flag set are: "3/28/90," "December 4, '80" and "Jan. 3, '89."

DTF_SLASH—Separates each date value with a slash, regardless of the default country date separator. Some example dates with the DTF_SLASH flag set are: "3/28/90," "12/04/1900" and "1/3/2003."

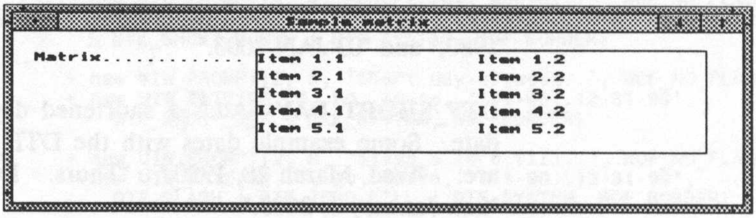
DTF_UPPER_CASE—Shows the date in an upper-case format. Some example dates with the DTF_UPPER_CASE flag set are: "MARCH 28, 1990," "DEC. 4, 1980" and "SATURDAY JAN 3, 2003."

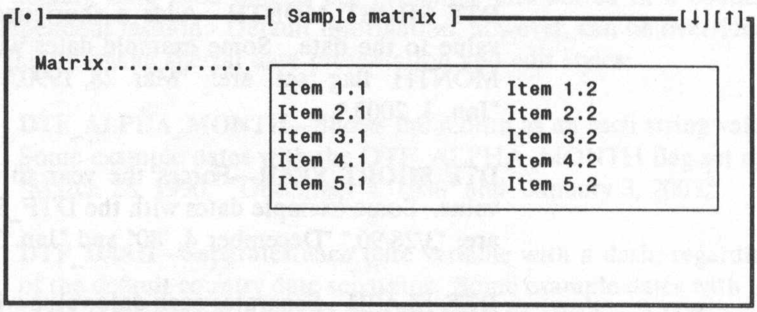
DTF_US_FORMAT—Forces the date to be formatted in the U.S. format (i.e., *month/day/year*), regardless of the default country information. Some example dates with the DTF_US_FORMAT flag set are: "March 28, 1990," "12/4/1980" and "Jan 3, 2003."

DTF_ZERO_FILL—Forces the year, month and day values to be zero filled when their values are less than 10. Some example dates with the DTF_ZERO_FILL flag set are: "March 08, 1990," "12/04/1980" and "01/03/2003."

For more information about the UIW_DATE window object see "Chapter 28—UIW_DATE" of the Programmer's Reference.

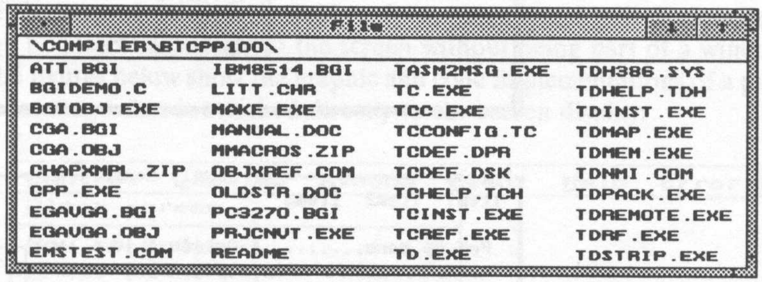
Matrix window objects Matrix fields should be used to present related information in a row/column fashion. The figures below show graphic, textual and code implementations of a window with a matrix field (UIW_MATRIX):






```
*window
    + new UIW_TITLE(" Sample matrix ", WOF_JUSTIFY_CENTER)

    + new UIW_PROMPT(2, 1, "Matrix..........", WOF_NO_FLAGS)
    + &(*new UIW_MATRIX(22, 1, 41, 6, 5, 14, 1, 0,
        MXF_NO_FLAGS, WOF_BORDER, WOAF_NO_FLAGS)
        + new UIW_STRING(0, 0, 19, "Item 1.1", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(20, 0, 19, "Item 1.2", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(0, 1, 19, "Item 2.1", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(20, 1, 19, "Item 2.2", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(0, 2, 19, "Item 3.1", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(20, 2, 19, "Item 3.2", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(0, 3, 19, "Item 4.1", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(20, 3, 19, "Item 4.2", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(0, 4, 19, "Item 5.1", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(20, 4, 19, "Item 5.2", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS));
```

In addition to the standard matrix field, the UIW_MATRIX class permits the creation of a matrix object that takes the complete window region (inside the border). For example, the graphic image below shows a directory window with a pull-down menu and a single matrix field:

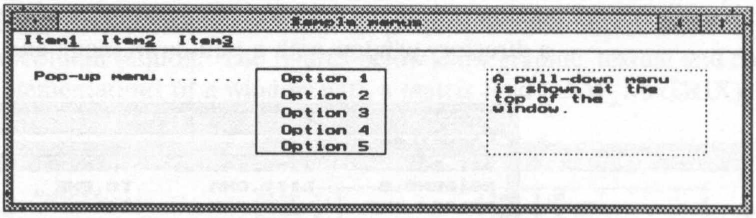


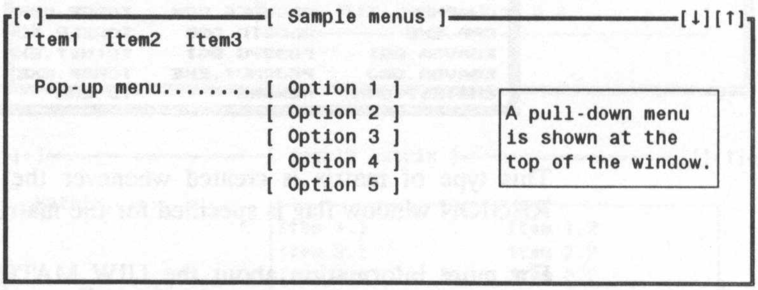
This type of matrix is created whenever the WOF_NON_FIELD_-REGION window flag is specified for the matrix object.

For more information about the UIW_MATRIX window object see "Chapter 31—UIW_MATRIX" of the Programmer's Reference.

Menu window objects

Pop-up menus should be used anytime you want to present selection information to the end-user. Pull-down items should be used when a hierarchal grouping of selection items is to be used. The pull-down menu serves as the first level in the selection process. The figures below show graphic, textual and code implementations of a window with a pull-down and pop-up menu. (The pull-down menu is shown as the horizontal line with the Item1-3 pull-down items. The pop-up menu is shown as the vertical group of Option1-5 pop-up items.)

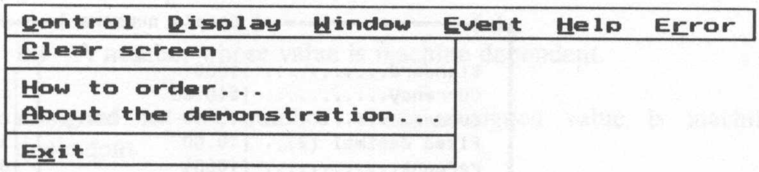





```
*window
        + new UIW_TITLE(" Sample menus ", WOF_JUSTIFY_CENTER)

        + &(*new UIW_PULL_DOWN_MENU(0, WOF_NO_FLAGS, WOAF_NO_FLAGS)
            + &(*new UIW_PULL_DOWN_ITEM(" Item~1 ", MNF_NO_FLAGS,0)
                + new UIW_POP_UP_ITEM("Option 1.1", 0,
                    MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS)
                + new UIW_POP_UP_ITEM("Option 1.2", 0,
                    MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS)
                + new UIW_POP_UP_ITEM("Option 1.3", 0,
                    MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS))
            + &(*new UIW_PULL_DOWN_ITEM(" Item~2 ", MNF_NO_FLAGS,0)
                + new UIW_POP_UP_ITEM("Option 2.1", 0,
                    MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS)
                + new UIW_POP_UP_ITEM("Option 2.2", 0,
                    MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS)
                + new UIW_POP_UP_ITEM("Option 2.3", 0,
                    MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS))
            + &(*new UIW_PULL_DOWN_ITEM(" Item~3 ", MNF_NO_FLAGS,0)
                + new UIW_POP_UP_ITEM("Option 3.1", 0,
                    MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS)
                + new UIW_POP_UP_ITEM("Option 3.2", 0,
                    MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS)
                + new UIW_POP_UP_ITEM("Option 3.3", 0,
                    MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS)))
        + new UIW_TEXT(43, 1, 20, 5,
            "A pull-down menu is shown at the top of the window.",
            128, TXF_NO_FLAGS, WOF_VIEW_ONLY | WOF_NON_SELECTABLE |
            WOF_BORDER)
        + new UIW_PROMPT(2, 1, "Pop-up menu........", WOF_NO_FLAGS)
        + &(*new UIW_POP_UP_MENU(22, 1, MNF_SELECT_ONE, WOF_BORDER,
            WOAF_NO_FLAGS)
            + new UIW_POP_UP_ITEM(" Option 1 ", 0, MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS)
            + new UIW_POP_UP_ITEM(" Option 2 ", 0, MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS)
            + new UIW_POP_UP_ITEM(" Option 3 ", 0, MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS)
            + new UIW_POP_UP_ITEM(" Option 4 ", 0, MNIF_NO_FLAGS,
```

```
            BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM(" Option 5 ", 0, MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS));
```

In addition to the default field options, pop-up and pull-down menus can be attached directly to the screen without being part of a window. The figures below show the graphic and code implementations of a pull-down menu that is attached directly to the screen display:



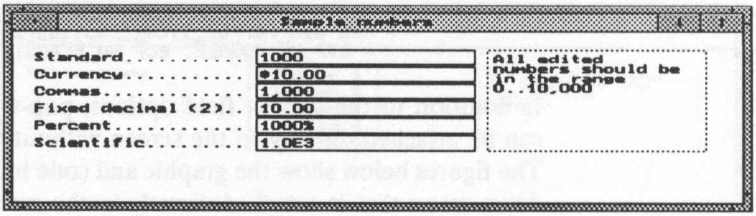
For more information about the menu window objects discussed above see:

"Chapter 38—UIW_POP_UP_MENU,"
"Chapter 35—UIW_POP_UP_ITEM,"
"Chapter 41—UIW_PULL_DOWN_MENU," or
"Chapter 40—UIW_PULL_DOWN_ITEM"

of the Programmer's Reference.

Number window objects

Number fields should be used anytime numeric information is presented to the end-user or when numeric information is to be entered at an application's run-time. The figures below show graphic, textual and code implementations of a window with several variations of a number field (UIW_NUMBER):




```
[•]───────────────[ Sample numbers ]────────────────[↓][↑]

  Standard........... [1000          ]  All edited
  Currency........... [$10.00        ]  numbers should be
  Commas............. [1,000         ]  in the range
  Fixed decimal (2).. [10.00         ]  0..10,000
  Percent............ [1000%         ]
  Scientific......... [1.2345E3      ]
```

```
*window
    + new UIW_TITLE(" Sample numbers ", WOF_JUSTIFY_CENTER)

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

    + new UIW_PROMPT(2, 1, "Standard...........", WOF_NO_FLAGS)
    + new UIW_NUMBER(22, 1, 20, &ivalue, "0..10000",
        NMF_NO_FLAGS, WOF_BORDER)

    + new UIW_PROMPT(2, 2, "Currency...........", WOF_NO_FLAGS)
    + new UIW_NUMBER(22, 2, 20, &ivalue, "0..10000",
        NMF_CURRENCY | NMF_DECIMAL(2), WOF_BORDER)

    + new UIW_PROMPT(2, 3, "Commas.............", WOF_NO_FLAGS)
    + new UIW_NUMBER(22, 3, 20, &ivalue, "0..10000",
        NMF_COMMAS, WOF_BORDER)

    + new UIW_PROMPT(2, 4, "Fixed decimal (2)..", WOF_NO_FLAGS)
    + new UIW_NUMBER(22, 4, 20, &ivalue, "0..10000",
        NMF_DECIMAL(2), WOF_BORDER)

    + new UIW_PROMPT(2, 5, "Percent............", WOF_NO_FLAGS)
    + new UIW_NUMBER(22, 5, 20, &ivalue, "0..10000",
        NMF_PERCENT, WOF_BORDER)

    + new UIW_PROMPT(2, 6, "Scientific.........", WOF_NO_FLAGS)
    + new UIW_NUMBER(22, 6, 20, &dvalue, "0..10000",
        NMF_SCIENTIFIC, WOF_BORDER);
```

The UIW_NUMBER class supports the following numeric types:

char—A number whose value is between -128 and 127 (8 bits, signed).

unsigned char—A number whose value is between 0 and 255 (8 bits, unsigned).

short—A number whose value is between -32,768 and 32,767 (16 bits, signed).

unsigned short—A number whose value is between 0 and 65,535 (16 bits, unsigned).

int—A number whose value is machine dependent.

unsigned int—A number whose unsigned value is machine dependent.

long—A number whose value is between -2,147,483,648 and 2,147,483,647 (32 bits, signed).

unsigned long—A number whose value is between 0 and 4,294,967,295 (32 bits, unsigned).

float—A single precision floating point number.

double—A double precision floating point number.

The number class object also permits the following presentation and edit styles:

NMF_DECIMAL—Shows the number with a decimal point at a fixed location. Some example numbers with the NMF_-DECIMAL(2) flag set are: "10,000.00," "43.45" and "$149.95."

NMF_CURRENCY—Shows the number with the country-specific currency symbol. Some example numbers with the NMF_-CURRENCY flag set are: "$10,000.00," "DM100" and "£195."

NMF_CREDIT—Shows the number with the '(' and ')' credit symbols whenever the number is negative. For example, if the value -10000 were entered and the NMF_CREDIT flag were set, the value would be shown as "(10000)."

NMF_COMMAS—Shows the number with commas. Some example numbers with the NMF_COMMAS flag set are: "$10,000.00," "45,000" and "1,195."

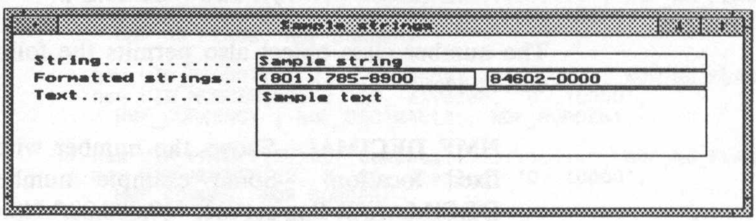
NMF_PERCENT—Shows the number with a percentage symbol. Some example numbers with the NMF_PERCENT flag set are: "100%," "4.5%" and "10%."

NMF_SCIENTIFIC—Shows the number in scientific format. This flag only has effect on real numeric types. Some example real numbers with the NMF_SCIENTIFIC flag set are: "1.0E+3," "4.5E-40" and "1.195E+0."

For more information about the UIW_NUMBER window object see "Chapter 34—UIW_NUMBER" of the Programmer's Reference.

String window objects

Several types of strings are supported by the Zinc Interface library. They include single line string fields (UIW_STRING), multi-line text fields (UIW_TEXT) and formatted or masked strings (UIW_FORMATTED_STRING). The figures below show graphic, textual and code implementations of these string window objects:




```
[•]──────────────[ Sample strings ]──────────────────[↓][↑]

  String............ [Sample string                    ]
  Formatted strings.. [(801) 785-8900    ] [84602-0000       ]
  Text...............
                       ┌──────────────────────────────┐
                       │Sample text                   │
                       └──────────────────────────────┘
```

```
*window
    + new UIW_TITLE(" Sample strings ", WOF_JUSTIFY_CENTER)
```

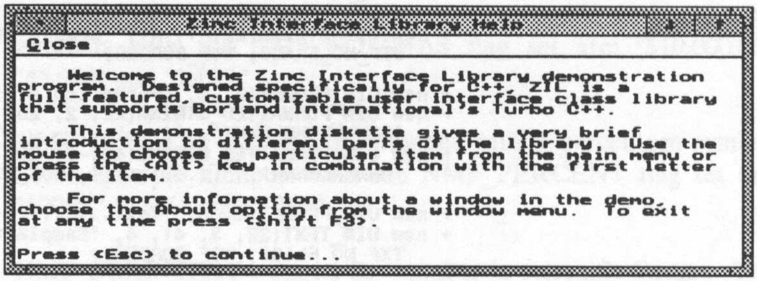
```
+ new UIW_PROMPT(2, 1, "String............", WOF_NO_FLAGS)
+ new UIW_STRING(22, 1, 41, "Sample string", 256,
    STF_NO_FLAGS, WOF_BORDER)
+ new UIW_PROMPT(2, 2, "Formatted strings..", WOF_NO_FLAGS)
+ new UIW_FORMATTED_STRING(22, 2, 20, "8017858900",
    "LNNNLLNNNLXXXX", "(...) ...-....", WOF_BORDER)
+ new UIW_FORMATTED_STRING(43, 2, 20, "846020000",
    "NNNNNLNNNN", ".....-....", WOF_BORDER)
+ new UIW_PROMPT(2, 3, "Text..............", WOF_NO_FLAGS)
+ new UIW_TEXT(22, 3, 41, 4, "Sample text", 1028,
    TXF_NO_FLAGS, WOF_BORDER);
```

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

The formatted string objects, shown with the "(801) 785-8900" and "84602-0000" default information in the windows above, are represented by the UIW_FORMATTED_STRING class object. This class object should be used anytime pre-defined string format information is presented to the end-user or when string information is to be entered at an application's run-time. Formatted strings restrict the type of information that an end-user can enter.

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

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



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

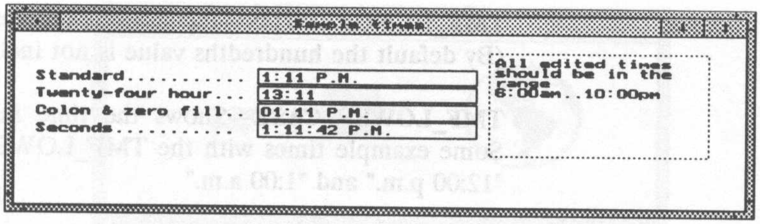
For more information about the string objects discussed above see:

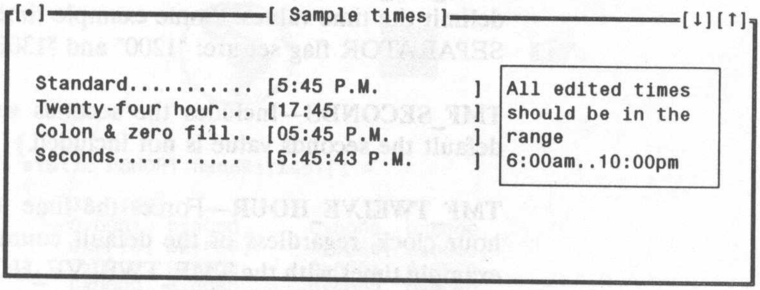
"Chapter 43—UIW_STRING,"
"Chapter 29—UIW_FORMATTED_STRING," or
"Chapter 45—UIW_TEXT"

of the Programmer's Reference.

Time window objects

Time fields should be used whenever time information is presented to the end-user or when time information is to be entered at an application's run-time. The figures below show graphic, textual and code implementations of a window with several variations of a time field (UIW_TIME):






```
*window
    + new UIW_TITLE(" Sample times ", WOF_JUSTIFY_CENTER)

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

    + new UIW_PROMPT(2, 2, "Standard..........", WOF_NO_FLAGS)
    + new UIW_TIME(22, 2, 20, &time, "6:00am..10:00pm",
      TMF_NO_FLAGS, WOF_BORDER)

    + new UIW_PROMPT(2, 3, "Twenty-four hour...", WOF_NO_FLAGS)
    + new UIW_TIME(22, 3, 20, &time, "6:00am..10:00pm",
      TMF_TWENTY_FOUR_HOUR, WOF_BORDER)

    + new UIW_PROMPT(2, 4, "Colon & zero fill..", WOF_NO_FLAGS)
    + new UIW_TIME(22, 4, 20, &time, "6:00am..10:00pm",
      TMF_COLON_SEPARATOR | TMF_ZERO_FILL, WOF_BORDER)

    + new UIW_PROMPT(2, 5, "Seconds..........", WOF_NO_FLAGS)
    + new UIW_TIME(22, 5, 20, &time, "6:00am..10:00pm",
      TMF_SECONDS, WOF_BORDER);
```

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

TMF_COLON_SEPARATOR—Separates each time variable with a colon. Some example times with the TMF_COLON_SEPARATOR flag set are: "12:00," "13:00:00" and "12:00 a.m."

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

TMF_LOWER_CASE—Shows the time in a lower-case format. Some example times with the TMF_LOWER_CASE flag set are: "12:00 p.m." and "1:00 a.m."

TMF_NO_SEPARATOR—Does not use any separator characters to delimit the time values. Some example times with the TMF_NO_-SEPARATOR flag set are: "1200" and "130000."

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

TMF_TWELVE_HOUR—Forces the time to be shown using a 12 hour clock, regardless of the default country information. Some example times with the TMF_TWELVE_HOUR flag set are: "12:00 a.m.," "1:00 p.m." and "5:00 p.m."

TMF_TWENTY_FOUR_HOUR—Forces the time to be shown using a 24 hour clock, regardless of the default country information. Some example times with the TMF_TWENTY_FOUR_HOUR flag set are: "12:00," "13:00" and "17:00."

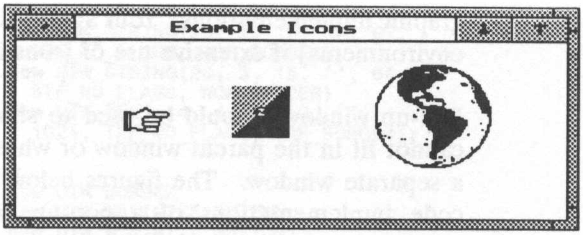
TMF_UPPER_CASE—Shows the time in an upper-case format. Some example times with the TMF_UPPER_CASE flag set are: "12:00 P.M." and "1:00 A.M."

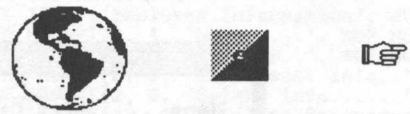
TMF_ZERO_FILL—Forces the hour, minute and second values to be zero filled when their values are less than 10. Some example times with the TMF_ZERO_FILL flag set are: "01:10 a.m," "13:05:03" and "01:01 p.m."

Other window objects

There are two additional object types supported by the Zinc Interface Library: icons and pop-up windows.

Icons are selectable graphic images that can be attached to a window or directly to the screen display. The figures below show graphic and partial code implementations (only 1 bit map is shown) of a window with 3 icons (UIW_ICON). In addition the same 3 icons are shown attached directly to the screen display.

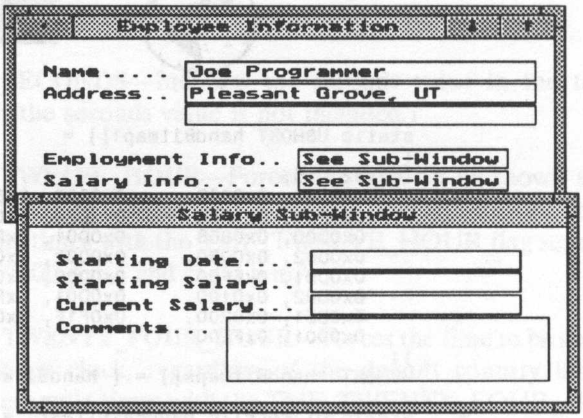




```
static USHORT handBitmap1[] =
{
    32, 15,
    0x0001, 0x8000,     0x0006, 0x6000,
    0x0F18, 0x1FF0,     0x0DE0, 0x0808,
    0x0D00, 0x0808,     0x0D01, 0xFFF0,
    0x0D02, 0x0080,     0x0D02, 0x0080,
    0x0D01, 0xFF00,     0x0D02, 0x0100,
    0x0D02, 0x0100,     0x0D01, 0xFE00,
    0x0DE1, 0x0200,     0x0F1F, 0x0200,
    0x0001, 0xFE00
};
USHORT *handBitmaps[] = { handBitmap1, 0 };

static UI_PALETTE handPalettes[] = { {
    '\260', attrib(BLACK, WHITE), attrib(MONO_DIM, MONO_BLACK),
    INTERLEAVE_FILL, attrib(BLACK, WHITE),
    attrib(BW_WHITE, BW_WHITE), attrib(GS_GRAY, GS_GRAY) } };
.
.
.
*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE("Example Icons", WOF_JUSTIFY_CENTER)
    + new UIW_ICON(7, 3, handBitmaps, handPalettes,
        ICF_NO_FLAGS, WOF_NO_FLAGS)
    + new UIW_ICON(15, 2, zincBitmaps, zincPalettes,
        ICF_NO_FLAGS, WOF_NO_FLAGS)
    + new UIW_ICON(25, 1, worldBitmaps, worldPalettes,
        ICF_NO_FLAGS, WOF_NO_FLAGS);

*windowManager
    + new UIW_ICON(7, 3, handBitmaps, handPalettes,
        ICF_NO_FLAGS, WOF_NO_FLAGS)
    + new UIW_ICON(15, 2, zincBitmaps, zincPalettes,
        ICF_NO_FLAGS, WOF_NO_FLAGS)
    + new UIW_ICON(25, 1, worldBitmaps, worldPalettes,
        ICF_NO_FLAGS, WOF_NO_FLAGS);
```

Icons can be used anytime you want to present a selectable item in graphical form. The main drawback of icons is that they only have graphic implementations. Your application will be less portable (to text environments) if extensive use of icons is used.

Pop-up windows should be used to show additional information that cannot fit in the parent window or whose presentation is enhanced by a separate window. The figures below show the graphic, textual and code implementations of a pop-up window (shown as the Salary Information window):




```
┌[•]─[Employee Information]────[↓][↑]┐
│                                      │
│  Name..... [Joe Programmer       ]   │
│  Address.. [Pleasant Grove, UT   ]   │
│            [                     ]   │
│                                      │
│  Employment Info.. [See Sub-Window]  │
│  Salary Info...... [See Sub-Window]  │
   ┌──────────[Salary Sub-Window]──────────┐
   │                                        │
   │  Starting Date.... [              ]    │
   │  Starting Salary.. [              ]    │
   │  Current Salary... [              ]    │
   │  Comments...                           │
   │              Hourly Wages              │
   │                                        │
   └────────────────────────────────────────┘
```

```
*popup2
    + new UIW_BORDER
    + new UIW_TITLE("Salary Sub-Window", WOF_JUSTIFY_CENTER)
    + new UIW_PROMPT(2, 1, "Starting Date....", WOF_NO_FLAGS)
    + new UIW_PROMPT(2, 2, "Starting Salary..", WOF_NO_FLAGS)
    + new UIW_PROMPT(2, 3, "Current Salary...", WOF_NO_FLAGS)
    + new UIW_PROMPT(2, 4, "Comments...", WOF_NO_FLAGS)
    + new UIW_DATE(20, 1, 15, &UI_DATE(), "",
```

```
         DTF_NO_FLAGS, WOF_BORDER)
    + new UIW_STRING(20, 2, 15, "", 64,
         STF_NO_FLAGS, WOF_BORDER)
    + new UIW_STRING(20, 3, 15, "", 64,
         STF_NO_FLAGS, WOF_BORDER)
    + new UIW_TEXT(14, 4, 23, 3, "",
         1024, TXF_NO_FLAGS, WOF_BORDER);

*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE("Employee Information", WOF_JUSTIFY_CENTER)
    + new UIW_PROMPT(2, 1, "Name.....", WOF_NO_FLAGS)
    + new UIW_PROMPT(2, 2, "Address..", WOF_NO_FLAGS)
    + new UIW_PROMPT(2, 5, "Employment Info..", WOF_NO_FLAGS)
    + new UIW_PROMPT(2, 6, "Salary Info......", WOF_NO_FLAGS)
    + new UIW_STRING(12, 1, 25, "Joe Programmer", 64,
         STF_NO_FLAGS, WOF_BORDER)
    + new UIW_STRING(12, 2, 25, "Pleasant Grove,  UT", 64,
         STF_NO_FLAGS, WOF_BORDER)
    + new UIW_STRING(12, 3, 25, "", 64,
         STF_NO_FLAGS, WOF_BORDER)
    + new UIW_POP_UP_WINDOW(20, 5, 15, "See Sub-Window",
         popup1, BTF_NO_FLAGS, WOF_NO_FLAGS);
    + new UIW_POP_UP_WINDOW(20, 6, 15, "See Sub-Window",
         popup2, BTF_NO_FLAGS, WOF_NO_FLAGS);
```

For more information about the additional window objects discussed above see:

> "Chapter 30—UIW_ICON,"
> "Chapter 38—UIW_POP_UP_WINDOW"

of the Programmer's Reference.

CHAPTER 4 – DEFAULT EVENT MAPPING

Overview

"Chapter 2—Conceptual Design" of this manual briefly discussed the implementation of event mapping in the Zinc Interface Library. This chapter describes the default mapping of events for the UI_BIOS_-KEYBOARD and UI_MS_MOUSE devices. This default event mapping conforms to the key assignments specified by IBM's Systems Application Architecture document—the Common User Access Panel Design and User Interaction edition.

The default event map information, provided below, can be changed by the programmer by redefining the system event map table *_eventMap-Table*. A complete discussion of how to change the default mapping is given in "Chapter 14—UI_EVENT_MAP" of the Programmer's Reference.

Default keyboard mapping

Action	Key	Description
Begin field	**<Ctrl Home>** **<Ctrl Grey Home>**	Moves to the beginning of the field.
Cancel	**<F12>**	Generates the logical S_CANCEL command.
Copy	**<Ctrl F7>**	Copies the entire contents of the current window field. The copied section is stored in a global paste buffer. This key only has effect in fields that can be edited.
Cut	**<Ctrl F6>**	Cuts the entire contents of the current window field. The cut section is removed and stored in a global paste buffer. This key only has effect in fields that can be edited.
Delete line	**<Alt Del>** **<Alt Grey Delete>**	Deletes all characters from the current cursor position to the end of the line.

Delete next character	**<Del>** **<Gray Delete>**	Deletes the character underneath the cursor, leaving the position of the cursor unchanged. This key only has effect in fields that can be edited and only where the cursor is not in the field's *last* position.
Delete previous character	**<Backspace>**	Moves the cursor *left* one position, deleting the character underneath the cursor (i.e. the character immediately to the left of the cursor before it is moved). This key only has effect in fields that can be edited and only where the cursor is not in the field's *first* character position.
Delete temporary window	**<Esc>**	If the current window is identified as a temporary window (WOAF_-TEMPORARY), pressing <Esc> removes the current window from the screen display. For example, when an end-user selects the system button, a pop-up menu appears. If the user presses <Esc> at this time, the pop-up menu is erased from the screen display.
Delete window	**<F3>** **<Alt F4>**	Closes a window that is not temporary.
Delete word	**<Ctrl Del>** **<Ctrl Grey Delete>**	Positions the cursor at the beginning of the word to be deleted, then deletes the word and any trailing spaces. The cursor remains in its original position after the deletion.
Down	**<↓>** **<Gray ↓>**	If the field occupies a single line on the screen or the cursor is positioned on the bottom line of a multi-line field, pressing <Down-arrow> moves from the current (or selected) window field to the window

		field immediately *below* the current field. The left or right edge of the field above must be on the same boundary as the current field (i.e., their left edges or right edges must have the same pixel or cell coordinate). If the field is a multi-line field and the cursor is not positioned on the bottom line, pressing <Down-arrow> moves the cursor *down* one line on the display.
Down page	**<PgDn>** **<Gray PgDn>**	If the field occupies a single line on the screen or the cursor is positioned on the bottom line of a multi-line field, pressing <PgDn> moves from the current (or selected) window field to the *last* window field. If the field is a multi-line field and the cursor is not positioned on the bottom line, pressing <PgDn> moves the cursor *down* one page in the current field.
End field	**<Ctrl End>** **<Ctrl Grey End>**	Moves to the end of the field.
End line	**<End>** **<Grey End>**	Moves the cursor to the end of the current line.
Exit	**<Shift F3>** **<Ctrl Break>** **<Ctrl C>**	Exits the application program.
Help— context sensitive	**<F1>**	Displays context sensitive help information regarding the current window.
Help— general	**<Alt F1>**	Displays general help information for the application program.

Home	**<Home>** **<Grey Home>**	Moves the cursor to the beginning of the current line.
Left	**<←>** **<Gray ←>**	If the cursor is positioned in the *first* character position of a right-hand field, pressing <Left-Arrow> moves the cursor to the *last* character position of a left-hand field. Otherwise, pressing <Left-Arrow> moves the cursor one character to the left.
Left word	**<Ctrl ←>** **<Ctrl Grey ←>** **<Alt ←>** **<Alt Grey ←>**	Moves the cursor to the beginning of the previous word or to the beginning of the same word if the cursor was originally positioned in the middle of that word. (word left)
Mark	**<Ctrl F5>**	Begins a marked region on the position of the cursor (only in fields that can be edited). When followed by any movement keys and then <Enter>, the marked text is copied. When followed by any movement keys and then <Del>, the marked text is cut. The cut section is removed and stored in a global paste buffer.
Maximize	**<Alt +>** **<Alt F10>**	Maximizes the size of the current window (i.e., increases the size of the window to occupy the entire screen). This key only has effect when the current window can be sized and if it is not already in a minimized state. If the window is in a maximized state, selecting this key causes the window to be restored to its original size.
Menu control	**<Alt>** **<F10>**	Selects the pull-down menu (if any) associated with the current window.

		This changes the highlight field, or cursor position, from the current field to the pull-down menu. This key only has effect when the current window has a pull-down menu.
Minimize	**<Alt –>** **<Alt F9>**	Minimizes the size of the current window (i.e., reduces the size of the window to the minimum allowed by the object type). This key only has effect when the current window can be sized and if it is not already in a maximized state. If the window is in a minimized state, selecting this key causes the window to be restored to its original size.
Move window	**<Alt F7>**	Moves the current window when followed by any movement key and then <Enter>. When followed by any movement key and then <Esc>, the selected window is returned to its original position.
Next field	**<Enter>** **<Gray Enter>** **<Tab>** **<F6>**	Moves from the current (or selected) window field to the *next* selectable window field. If the last window field is currently selected, pressing <Tab> cycles to the *first* selectable window field.
Next window	**<Alt F6>**	Moves from the current (or selected) window to the *next* selectable window in the window manager's list of windows.
Paste	**<Ctrl F8>**	Retrieves the cut section from the global paste buffer and pastes it in the current field. This key only has effect in fields that can be edited.

Previous field	<BackTab> <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.
Redo	<Ctrl F10>	Restores, in the current field, the most recent changes executed using the undo function (<F9>) in that field. For example, the undo operation below explains how an end-user may accidently delete a complete line in a text field. If the user continued to perform undo operations, then decided some of the old text was worthwhile, the information could still be retrieved by pressing the <Ctrl F10> key or the <Alt Right> mouse button (hold the <Alt> key while pressing the right mouse button). The redo operation is implemented on a field-by-field basis. Thus, an end-user could perform edit and undo operations on one field, move to a different field to perform edit operations, then return to the original field and perform redo operations.
Refresh	<F5>	Refreshes the screen. (Re-displays all of the window objects on the screen.)
Restore	<Alt F5>	Restores the original size of the window. Used with <Alt +> and <Alt ->.
Right	<→> <Gray →>	If the cursor is positioned in the *last* character position of a left-hand field, pressing <Right-Arrow>

		moves the cursor to the *first* character position of a right-hand field. Otherwise, pressing <Right-Arrow> moves the cursor one character to the right.
Right word	**<Ctrl →>** **<Ctrl Gray →>** **<Alt →>** **<Alt Gray →>**	Moves the cursor to the beginning of the next word. (word right)
Size window	**<Alt F8>**	Sizes, from the bottom right corner, the current window when followed by any movement key. Pressing <Enter> accepts the alteration in size, while pressing <Esc> returns the window to its original size.
System	**<Alt Spacebar>** **<Alt .>**	Selects the system button (if any) associated with the current window. This causes the pop-up menu associated with the current window's system button to be displayed on the screen.
Toggle	**<Ins>** **<Gray Insert>**	Toggles the edit mode from *insert* to *overstrike* mode or vise-versa. This key only has effect in fields that can be edited.
Undo	**<Ctrl F9>**	Undoes the most recent changes in the current field. For example, if an end-user accidently deleted a complete line in a text field, the information could be retrieved by pressing the <Ctrl F9> key or the <Alt Left> mouse button (i.e., holding the <Alt> key while pressing the left mouse button). The undo operation is implemented on a field-by-field basis. Thus, an

			end-user could perform edit and undo operations on one field, move to a different field to perform edit operations, then return to the original field and continue edit or undo operations.
	Up	<↑> <Gray ↑>	If the field occupies a single line on the screen or the cursor is positioned on the top line of a multi-line field, pressing <Up-arrow> moves from the current (or selected) window field to the window field immediately *above* the current field. The left or right edge of the field above must be on the same boundary as the current field (i.e., their left edges or right edges must be on the same pixel or cell coordinate). If the field is a multi-line field and the cursor is not positioned on the top line, pressing <Up-arrow> moves the cursor *up* one line on the display.
	Up page	<PgUp> <Gray PageUp>	If the field occupies a single line on the screen or the cursor is positioned on the top line of a multi-line field, pressing <PgUp> moves from the current (or selected) window field to the *first* window field. If the field is a multi-line field and the cursor is not positioned on the top line, pressing <PgUp> moves the cursor *up* one page in the current field.
Default mouse mapping	**Action**	**Mouse**	**Description**
	Choose	<Left-down-click>	If the end-user is on the window's title bar, pressing this button moves the window. If the end-user is on

		the window's border, pressing this button sizes the window. Otherwise, pressing the left mouse button selects the field positioned under the mouse cursor (if the field is selectable).
Copy	**<Left-drag, Right-down-click>**	**If the user is in an edit field and a region has been marked, pressing these buttons copies the marked portion of the current window field.**
Cut	**<Right-down-click>**	**If the user is in an edit field and a region has been marked, pressing this button cuts the marked portion of the current window field. If the user is in an edit field and a region has <u>not</u> been marked, pressing this button pastes the contents of the global paste buffer (if any) to the current field.**
Mark	**<Left-drag>**	**If the current field is an field that can be edited, holding the left button down and dragging the mouse specifies the mark location.**
Paste	**<Right-down-click>**	**If the user is in an edit field and a region has been previously cut or copied, pressing these buttons together copies the marked portion of the current window field into the global paste buffer.**
Redo	**<Alt Right-down-click>**	**Restores, in the current field, the most recent changes executed using the undo function (<F9>) in that field. For example, the undo operation below explains how an end-user may accidently delete a complete line in a text field. If the user continued to perform undo operations, then decided some of the**

		old text was worthwhile, the information could still be retrieved by pressing the <Ctrl F10> key or the <Alt Right> mouse button (holding the <Alt> key while pressing the right mouse button). The redo operation is implemented on a field-by-field basis. Thus, an end-user could perform edit and undo operations on one field, move to a different field to perform edit operations, then return to the original field and perform redo operations.
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.
Undo	**<Alt Left-down-click>**	Undoes the most recent changes in the current field. For example, if an end-user accidently deleted a complete line in a text field, the information could be retrieved by pressing the <Ctrl F9> key or the <Alt Left> mouse button (holding the <Alt> key while pressing the left mouse button). The undo operation is implemented on a field-by-field basis. Thus, an end-user could perform edit and undo operations on one field, move to a different field to perform edit operations, then return to the original field and continue edit or undo operations.

CHAPTER 5 – DEFAULT PALETTE MAPPING

Introduction

"Chapter 2—Conceptual Design" of this manual briefly discusses the implementation of palette mapping in the Zinc Interface Library. This chapter describes the default mapping of color palettes for all the window objects.

The default palette map information, provided below, can be changed by the programmer by redefining the following global palette map tables:

_normalPaletteMapTable,
_helpPaletteMapTable and
_errorPaletteMapTable.

A complete discussion of how to change the default mapping is given in "Chapter 20—UI_PALETTE_MAP" of the Programmer's Reference.

The following naming convention is used by the palette tables shown below:

BORDER—The UIW_BORDER class object.

BUTTON—The UIW_BUTTON class object, which includes the following derived classes: UIW_MAXIMIZE_BUTTON, UIW_MINIMIZE_BUTTON and UIW_SYSTEM_BUTTON.

Color Attributes—The colors used by the Zinc Interface Library. The following colors are used: black, blue, green, cyan, red, magenta, brown, lightgray, darkgray, lightblue, lightgreen, lightcyan, lightred, lightmagenta, yellow, white, black, dim, normal, high.

CURRENT—Refers to the current or selected window or window object.

COLOR GRAPHICS—Refers to a color graphics display.

COLOR TEXT—Refers to a color text display.

MENU—Refers to the objects displayed as menus and includes the following class objects: UIW_POP_UP_ITEM, UIW_POP_UP_MENU, UIW_PULL_DOWN_ITEM and UIW_PULL_DOWN_MENU.

MONOCHROME GRAPHICS—Refers to a monochrome graphics display.

MONOCHROME TEXT—Refers to a monochrome text display.

NON-CURRENT—Refers to a non-current window or window object.

PROMPT—The UIW_PROMPT class object.

TITLE—The UIW_TITLE class object.

WINDOW—The UIW_WINDOW class object.

WINDOW OBJECT—The UI_WINDOW_OBJECT class object.

Standard window colors

Standard window colors are obtained from the *_normalPaletteMapTable* global palette table. The default color combinations for this palette table are shown in the table below:

OBJECT	DISPLAY		FOREGROUND	BACKGROUND
BORDER	COLOR GRAPHICS	CURRENT	darkgray	darkgray
		NON-CURRENT	darkgray	darkgray
	COLOR TEXT	CURRENT	white	lightgray
		NON-CURRENT	white	lightgray
	MONOCHROME GRAPHICS	CURRENT	black	white
		NON-CURRENT	black	white
	MONOCHROME TEXT	CURRENT	high	black
		NON-CURRENT	normal	black
BUTTON	COLOR GRAPHICS	CURRENT	darkgray	lightgray
		NON-CURRENT	darkgray	lightgray
	COLOR TEXT	CURRENT	white	lightgray
		NON-CURRENT	white	lightgray
	MONOCHROME GRAPHICS	CURRENT	black	white
		NON-CURRENT	black	white
	MONOCHROME TEXT	CURRENT	high	black
		NON-CURRENT	normal	black
MENU	COLOR GRAPHICS	CURRENT	black	lightgray
		NON-CURRENT	black	white
	COLOR TEXT	CURRENT	lightgray	darkgray
		NON-CURRENT	darkgray	lightgray
	MONOCHROME GRAPHICS	CURRENT	white	black
		NON-CURRENT	black	white
	MONOCHROME TEXT	CURRENT	black	normal
		NON-CURRENT	normal	black
PROMPT	COLOR GRAPHICS	CURRENT	black	white
		NON-CURRENT	black	white

(continued on the next page)

(continued from the previous page)

PROMPT	COLOR TEXT	CURRENT	yellow	lightgray
		NON-CURRENT	yellow	lightgray
	MONOCHROME GRAPHICS	CURRENT	black	white
		NON-CURRENT	black	white
	MONOCHROME TEXT	CURRENT	high	black
		NON-CURRENT	high	black
TITLE	COLOR GRAPHICS	CURRENT	yellow	blue
		NON-CURRENT	blue	white
	COLOR TEXT	CURRENT	yellow	lightgray
		NON-CURRENT	white	lightgray
	MONOCHROME GRAPHICS	CURRENT	white	black
		NON-CURRENT	black	white
	MONOCHROME TEXT	CURRENT	high	black
		NON-CURRENT	normal	black
WINDOW	COLOR GRAPHICS	CURRENT	black	white
		NON-CURRENT	black	white
	COLOR TEXT	CURRENT	white	lightgray
		NON-CURRENT	white	lightgray
	MONOCHROME GRAPHICS	CURRENT	white	black
		NON-CURRENT	white	black
	MONOCHROME TEXT	CURRENT	normal	black
		NON-CURRENT	normal	black
WINDOW OBJECT	COLOR GRAPHICS	CURRENT	black	white
		NON-CURRENT	black	white
	COLOR TEXT	CURRENT	black	white
		NON-CURRENT	black	lightgray
	MONOCHROME GRAPHICS	CURRENT	black	white
		NON-CURRENT	black	white
	MONOCHROME TEXT	CURRENT	black	normal
		NON-CURRENT	normal	black

Help window colors The help window system uses the default window colors specified in the *_helpPaletteMapTable* global palette table. The default color combinations for this palette table are shown in the table below:

OBJECT	DISPLAY		FOREGROUND	BACKGROUND
BORDER	COLOR GRAPHICS	CURRENT	lightgray	lightgray
		NON-CURRENT	lightgray	lightgray
	COLOR TEXT	CURRENT	lightgreen	lightgray
		NON-CURRENT	lightgreen	lightgray
	MONOCHROME GRAPHICS	CURRENT	white	black
		NON-CURRENT	white	black
	MONOCHROME TEXT	CURRENT	high	black
		NON-CURRENT	normal	black
BUTTON	COLOR GRAPHICS	CURRENT	lightgreen	lightgray
		NON-CURRENT	darkgray	lightgray
	COLOR TEXT	CURRENT	lightgreen	lightgray
		NON-CURRENT	lightgreen	lightgray
	MONOCHROME GRAPHICS	CURRENT	white	black
		NON-CURRENT	white	black
	MONOCHROME TEXT	CURRENT	normal	black
		NON-CURRENT	normal	black
TITLE	GRAPHICS	CURRENT	yellow	green
		NON-CURRENT	green	white
	TEXT	CURRENT	yellow	lightgray
		NON-CURRENT	lightgreen	lightgray
	MONOCHROME GRAPHICS	CURRENT	black	white
		NON-CURRENT	white	black
	MONOCHROME TEXT	CURRENT	high	black
		NON-CURRENT	normal	black

Error window colors The error window system uses the default window colors specified in the *_errorPaletteMapTable* global palette table. The default color combinations for this palette table are shown in the table below:

OBJECT	DISPLAY		FOREGROUND	BACKGROUND
BORDER	COLOR GRAPHICS	CURRENT	lightgray	lightgray
	COLOR TEXT	CURRENT	lightred	lightgray
	MONOCHROME GRAPHICS	CURRENT	white	black
	MONOCHROME TEXT	CURRENT	high	black
BUTTON	COLOR GRAPHICS	CURRENT	darkgray	lightgray
	COLOR TEXT	CURRENT	lightred	lightgray
	MONOCHROME GRAPHICS	CURRENT	white	black
	MONOCHROME TEXT	CURRENT	normal	black
TITLE	COLOR GRAPHICS	CURRENT	yellow	red
	COLOR TEXT	CURRENT	yellow	lightgray
	MONOCHROME GRAPHICS	CURRENT	black	white
	MONOCHROME TEXT	CURRENT	high	black

CHAPTER 6 – TUTORIALS

Overview

The following tutorials will help you get started using the Zinc Interface Library ("ZIL"). It is assumed that you are familiar with the basic concepts of the C++ language and know how to compile, link and execute the programs provided. A sample MAKEFILE is included with the tutorials. It is used with Borland's MAKE utility to compile and link each of the programs and can be modified for the specific environment of your system. The tutorials move through a large number of examples with comments on the particular features and ZIL objects used. Particular attention should be paid to the code itself as this is the easiest way to quickly understand difficult concepts.

The following tutorials demonstrate the basic Zinc Interface Library elements:

- **"Hello World!"** is a simple window with text. It shows how the display, event manager and window manager are constructed.

- **"Notepad"** is program with multiple windows containing data that can be moved between them. It shows how the help and error systems are used. In addition, the cut, copy, paste, undo and redo capabilities are demonstrated.

- **"Calendar"** is a short program which shows how to derive objects from ZIL class objects.

- **"The Custom Application"** is a continuation of the "Calendar" tutorial and shows how colors and key mapping can be customized.

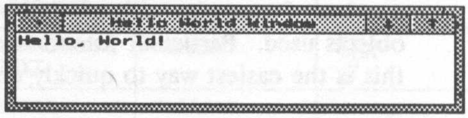
- **"Phone Book"** is a simple data base. It uses data entry fields inside a window to collect and save names and phone numbers to a file.

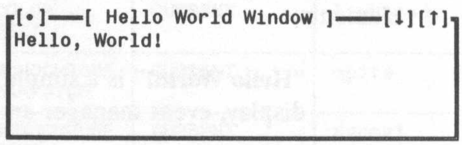
The five tutorials range from the simplest program "Hello World" to a more complex and useful phone book program. Each tutorial program is found in the TUTORIAL directory. For more help or for specific examples of the ZIL class objects, see the example programs in the EXAMPLES directory and refer to the Programmer's Reference.

Hello World!

This first tutorial program demonstrates how to set up the basic Zinc Interface Library elements. Using a slight modification of the classic sample program "Hello World," this tutorial sets up the display, creates a simple window and displays the text inside the window.

The final program will produce a screen similar to one of the following, depending on the graphics or text display mode:





The code for the "Hello World!" program is located in **\ZINC\TUTORIAL\HELLO.CPP**. Be sure that the C++ compiler directory is in the path. The executable program is made by typing "make hello.exe" at the command line while in the TUTORIAL directory.

Creating the display

This window is created by following some simple steps. At the top of the program the header file **UI_WIN.HPP** is included by using the following code:

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

This header file contains all of the class prototypes for windows and window objects. Because of ZIL's header file hierarchy all of the header files required by ZIL objects are also included. (See the "Introduction" in the Programmer's Guide for more information about the header files.) To gain access to the ZIL header files, list them with the INCLUDE environment variable (in **TURBOC.CFG**) or in the IDE configuration file.

In the first few lines of the main program you must initialize the screen display, event manager and window manager. These modules are used by all of the higher level ZIL objects. The following code segment initializes the screen display:

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

The screen display is the first object that must be set up. The code above uses the new operator to construct a display object. The new operator calls the graphics display class constructor and allocates memory for each of its members. If this fails (i.e., a graphics compatible card does not exist) then the graphics display object is deleted and a text display object is created. If a graphics display is used, the Turbo C++ BGI files must be in the environment PATH in order to run the final program. The BGI files include important graphics display information that is needed at run-time.

If the text display is to be the default display, then the following code segment should be used in place of the code segment above:

```
UI_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
```

Both of the examples above auto-detect the particular default display mode.

The event manager

The following code segment creates the event manager and input devices:

```
// Initialize the event manager.
UI_EVENT_MANAGER eventManager(100, display);
eventManager
    + new UI_BIOS_KEYBOARD
    + new UI_MS_MOUSE
    + new UI_CURSOR;
```

The event manager is constructed in the first line. It requires two parameters:

- *100* is the maximum number of elements in the event queue.

- *display* is a pointer to the screen display.

The event manager polls each device in its device list for events. When an event occurs (i.e., a key is pressed) or a message is sent, it is added to the event queue. The event is then channeled to the correct receiving object (e.g., the window manager).

Adding devices to the event manager is very easy with the **UI_EVENT_MANAGER::operator+** operator overload. The keyboard, mouse and cursor devices are added to the event manager in the example above using the + operator. These devices and others can be added or subtracted (using the **UI_EVENT_MANAGER::operator-** operator overload) from the event manager at other places in the program also.

The window manager

The window manager is created in a similar way to the event manager.

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

The window manager is constructed with two parameters:

- *display* is a pointer to the screen display.
- *&eventManager* is a pointer to the event manager.

The window manager controls the presentation and operation of windows and window objects on the screen. It routes all events from the event manager devices to windows attached to the window manager. Once a window is attached, it receives event information from the window manager.

Creating the window

The example below calculates the screen center by using information from the screen display and then creates a new window with basic window objects (border and button) and a text field:

```
// Create a window in the screen center with text inside.
int centerX = display->columns / display->cellWidth / 2;
int centerY = display->lines / display->cellHeight / 2;
UIW_WINDOW *window = new UIW_WINDOW(centerX - 20, centerY - 3,
    40, 6, WOF_NO_FLAGS, WOAF_NO_FLAGS);
*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE(" Hello World Window ",
        WOF_JUSTIFY_CENTER)
```

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

Each window object is created and added to the window using the **UIW_WINDOW::operator+** operator overload.

In addition, the text object is created using the WOF_NON_-FIELD_REGION flag. This flag tells the parent window that the text region will cover all of the window. (See "Chapter—25 UI_WINDOW_-OBJECT" of the Programmer's Reference for more information about WOF flags.)

Adding the window

The window must be added to the window manager in order to be displayed to the screen and receive event information. The UI_-**WINDOW_MANAGER::operator+** operator overload is used to add the window to the window manager.

```
windowManager + window;
```

The event loop

After the display, event manager and window manager have been initialized, the following loop is used to retrieve input from the user:

```
// Wait for user response.
int ccode;
UI_EVENT event;
do
{
    // Get input from the user.
    eventManager.Get(event, Q_NORMAL);

    // Interpret an <Esc> as an L_EXIT message.
    if (event.type == E_KEY && event.rawCode == ESCAPE)
        event.type = L_EXIT;

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

The first step calls the event manager to get an event. At run-time the event manager polls all of the attached input devices until an event occurs. The returned event is then checked by the program before it is passed on to the window manager. In this example, the <Esc> key is retrieved by the main program loop and changed to an exit message (L_EXIT) before the window manager receives it. If the L_EXIT key is interpreted or passed back from the window manager, the program ends.

Clean up At the conclusion of the program you must manually destroy any objects you have created using the new operator. Any objects that have been added to the event manager or window manager, however, are automatically destroyed by the manager's destructor routine. In this tutorial the event manager and window manager destructors are called automatically when the scope of main ends.

The delete operator calls the class destructor and de-allocates memory used by the class object. The following code deletes the screen display:

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

Run-time features Some of the best features of the Zinc Interface Library are inherently available to windows and the objects attached to them. Running the program you see the window with the text "Hello World!" inside. This window can be moved to another place on the screen (or off the screen) using either the keyboard or mouse. You can perform the following actions on the "Hello World!" window at run-time:

Move—Pressing the left mouse button with the mouse pointer on the window's title bar and "dragging" the mouse or pressing <Alt F7> and using the arrow keys allows you to move the window to any part of the screen.

Size—Moving the mouse pointer to one of the corner or border regions on the window and "dragging" the corner or pressing <Alt F8> and using the arrow keys allows you to size the window.

Minimize—Clicking the mouse on the minimize button (shown as a button at the right top of the window with a '↓' character) or pressing <Alt F9> reduces the window to the minimum size allowed by the window.

Maximize—Clicking the mouse on the maximize button (shown as a button at the right top of the window with a '↑' character) or pressing <Alt F10> changes the window size to occupy the entire screen display.

Restore—Clicking the mouse on the maximize button after the window is maximized (or on the minimize button if minimized) or pressing <Alt F5> changes the window size to the former size.

Exit—Clicking on the system button (shown as a button on the left top side of the window with a '•' character), pressing <Esc>, or pressing <Alt F10>, closes the window and exits the program.

Notepad

This tutorial demonstrates the following:

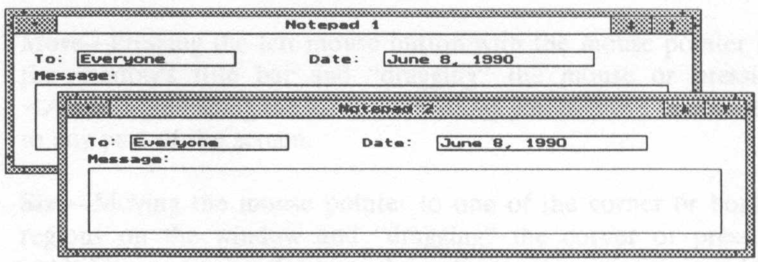
- How to create two windows with fields that allow interactive editing at run-time.

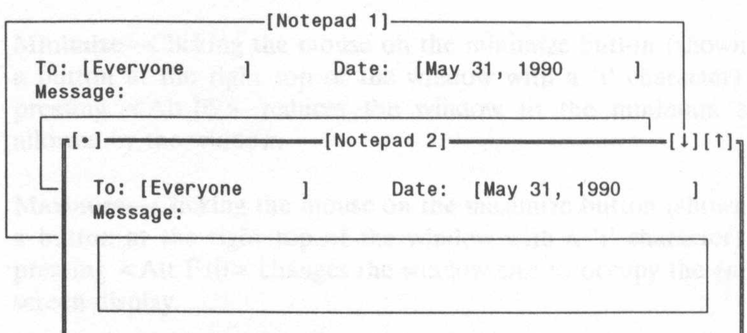
- How to create a help file.

- How to initialize the help and error systems.

Creating fields allows the user to cut, copy and paste between various windows on the screen at run-time. Together with the undo/redo capabilities of each data field in each window, this allows you to have greater flexibility in customizing the final product to any need.

The final program will produce a screen similar to one of the following, depending on the graphics or text display mode:





The code for the "Notepad" program is located in **\ZINC\TUTORIAL\NOTEPAD.CPP**. Be sure that the C++ compiler directory is in the path. The executable program is made by typing "make notepad.exe" at the command line while in the TUTORIAL directory.

The help system

For a complete application to be user friendly, a context sensitive help system must be installed. Using the help system supplied with the Zinc Interface Library, you can easily create and modify help information that can be accessed throughout your application.

Each help context—a page or more of specific information—can be attached to any number of windows or be assigned to the general help context. At run-time, the help key (defaulted to <F1>) displays the help information assigned to the current window. If you do not assign a help context to a particular window then the general help information is presented. You may also call the help system at any time to display a particular help context.

The help context information is read from a binary help file on the disk when needed. This file is created from a text file using the GENHELP.EXE utility which is supplied with the Zinc Interface Library. For example, the text file NOTEPAD.TXT below was converted into a binary help file using GENHELP.EXE:

```
--- HELP_GENERAL 1 ---
General Help
This application demonstrates how to mark, cut, copy and
paste between windows.

Press <Esc> to continue...

--- HELP_NOTEPAD 2 ---
Notepad Help
Use the following keys to move information between the windows.

Mark   - <Ctrl F5> or <Left-drag> on the mouse                    \
Cut    - <Ctrl F6> or <Right-down-click> on the mouse             \
Copy   - <Ctrl F7> or <Left-down><Right-down-click> the mouse     \
Paste  - <Ctrl F8> or <Right-down-click> on the mouse             \
Undo   - <Ctrl F9>                                                \
Redo   - <Ctrl F10>

Press <Esc> to continue...
```

There are two help contexts in the example above. Each one is preceded by the help context name and unique identification number, enclosed by three dashes on both sides. The first line after the help context name is the title that is displayed in the help window at run-time. All lines between the title and the next help context or file end are displayed inside the scrollable help window. Each of these lines is displayed in the window *without* the carriage return at the end of the line, *unless* it is followed by either a blank line or a backslash. For example, two consecutive lines without a backslash would be equivalent to one long line.

Generating the help file

Typing "genhelp notepad.txt" at the DOS command line generates two files. Be sure that the file **GENHELP.EXE**, located in the UTIL directory, is included in the environment PATH variable.

The first file generated is the binary help file **NOTEPAD.HLP** and the second file is a header file named **NOTEPAD.HLH**. The header file should be included in each module of your program, since it contains declarations for the constants used to reference the help context information. The generated header file appears as follows:

```
// This file was created by the genhelp utility.
// PLEASE DO NOT MODIFY WITH AN EDITOR!.

const int HELP_GENERAL = 1; // General Help
const int HELP_NOTEPAD = 2; // Notepad Help
```

The help context information in the text file can be modified and regenerated without recompiling the program if the help context names do not change. This is very useful if you have international versions of your application that require different help files.

Initialization

Creating two windows with editable fields is just as easy as creating the window for the "Hello World!" tutorial. Two header files are included at the top of the "Notepad" program:

```
#include <ui_win.hpp>
#include "notepad.hlh"
```

The first is the ZIL header file containing class prototypes of ZIL objects. The second is the help header file that contains the help context name declarations.

A display class object is constructed in the following manner. (This is exactly the same as the code example in the "Hello World!" tutorial.)

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

```
    display = new UI_DOS_TEXT_DISPLAY;
}
```

The event manager and window manager are constructed next.

```
// Initialize the event manager.
UI_EVENT_MANAGER *eventManager =
    new UI_EVENT_MANAGER(100, display);
*eventManager
    + new UI_BIOS_KEYBOARD
    + new UI_MS_MOUSE
    + new UI_CURSOR;

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

The creation of the event and window managers is different in this tutorial than the way that they were created in the "Hello World!" tutorial. In the "Hello World!" example the two managers were created in scope (without the new operator). There are two advantages in using the new operator to construct the event manager and window manager:

- You can create them in separate initialization procedures (i.e., outside main). If the new operator is not used, the managers are destroyed automatically when the scope of the initialization procedure ends.
- The managers can be accessed by other routines. For instance, in most large applications, it is helpful to have global variables that point to these managers. The "Phone Book" tutorial program (given later in this chapter) uses global variables so that all routines have direct access to the event manager and window manager.

(See the C++ compiler user's guide for more information about scope.)

Help window system

The next step in the initialization constructs the help window system. The new help system is assigned to the global variable *_helpSystem* using the following code:

```
_helpSystem = new UI_HELP_WINDOW_SYSTEM("notepad.hlp",
    &windowManager, HELP_GENERAL);
```

The help window system constructor has three parameters:

- *"notepad.hlp"* is name of the binary help file (generated from an ascii text file using **GENHELP.EXE**).

- *&windowManager* is a pointer to the window manager.
- *HELP_GENERAL* is the name of the general help context listed in the help file.

If you do not have any general help context information, then the third parameter is not needed.

The help system uses the help file generated earlier in this tutorial to display context sensitive help in a window. (See "Chapter 16—UI_HELP_WINDOW_SYSTEM" for more information about the help window system and the difference between the help system and help window system.)

Error window system

The basic error system that is installed automatically with the Zinc Interface Library warns the user of an error with a simple beep on the computer's speaker. In this tutorial the error window system is used. This allows you to tailor error messages to be announced when errors occur. The following code segment initializes the error system.

```
// Initialize the error window system.
_errorSystem = new UI_ERROR_WINDOW_SYSTEM;
```

At run-time the error system will report an error message for any invalid date that is entered by the user in the date field on either notepad. (In the "Phone book" tutorial, later in this chapter, you will see how the programmer can define custom error messages.)

The Zinc Interface Library does not include the help window system or error window system automatically. This allows you as the programmer to specify and create your own help or error system. The files **G_HELP.CPP** and **G_ERROR.CPP** are included in \ZINC\UTIL as example files of the basic help and error systems.

(See "Chapter 11—UI_ERROR_WINDOW_SYSTEM" of the Programmer's Reference for more information on the error window system.)

Creating two notepads

The two notepads are created as windows with two data entry "fields" which are editable window objects attached to a window. In this tutorial three fields and two prompt objects, along with the border and system buttons, are attached to each notepad window. The construction of the notepad windows is shown in the code segment below:

```
UIW_WINDOW *notepad1 = new UIW_WINDOW(5, 5, 68, 12,
    WOF_NO_FLAGS, WOAF_NO_FLAGS);

UIW_WINDOW *notepad2 = new UIW_WINDOW(10, 10, 68, 12,
    WOF_NO_FLAGS, WOAF_NO_FLAGS);
```

Notepad1 is in the upper left-hand corner of the screen. Notepad2 window is created in the same way and is positioned below and to the right of notepad1.

Adding window objects

Several window objects are added to the notepad windows as shown in the code segment below for notepad1:

```
*notepad1
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE("Notepad 1", WOF_JUSTIFY_CENTER)
```

The border, system buttons and title are constructed and added to the notepad1 window. This section is the similar to how the "Hello, World!" window was created. Each one is added to the notepad window using the **UI_WINDOW::operator+** operator overload.

Prompts

Each data entry field (or several related fields) should have a prompt associated with it. A prompt is used to describe the field following it. The prompt window object is constructed using four parameters as shown below:

```
    + new UIW_PROMPT(2, 1, "To:", WOF_NO_FLAGS)
    + new UIW_STRING(6, 1, 15, "Everyone", 40, STF_NO_FLAGS,
        WOF_BORDER)

    + new UIW_PROMPT(22, 1, "Date:", WOF_NO_FLAGS)
    + new UIW_DATE(22, 1, 20, &UI_DATE(), "",
        DTF_NO_FLAGS, WOF_BORDER)

    + new UIW_PROMPT(2, 2, "Message:", WOF_NO_FLAGS)
    + new UIW_TEXT(2, 3, 60, 4, "", 1028, TXF_NO_FLAGS,
        WOF_BORDER);
```

The first prompt describes the string field in which the name is entered:

```
    + new UIW_PROMPT(2, 1, "To:", WOF_NO_FLAGS)
```

- *2* and *1* are left and top coordinate positions of the prompt string (relative to the upper left corner of the window). These coordinates are relative cell coordinates inside the window and are zero based. For example, this prompt is located one character below the title of the window and two characters to the right of the window's left border. (These two parameters are common to most window objects.)

- "*To:*" is the text that is displayed at the position specified by the first two parameters.

- *WOF_NO_FLAGS* indicates that no special window object flags are specified. This parameter allows you to specify flags that control the display of the window.

(See "Chapter 38—UIW_PROMPT" of the Programmer's Reference for more information about the prompt window object and WOF flags.)

String field The first editable field is a string window object.

```
+ new UIW_STRING(6, 1, 15, "Everyone", 40, STF_NO_FLAGS,
    WOF_BORDER)
```

This field is used at run-time to enter the name of the person receiving the message. The string constructor is passed seven parameters:

- *6* and *1* are the top and left coordinates of the field inside the window.

- *15* is the display width of the string field. If more than 15 characters are entered in the field at run-time then the string buffer is scrolled left until the string's maximum length (40) is reached.

- "*Everyone*" is the initial information string that is to be displayed in the field. You can change this text at run-time since the field can be edited.

- *40* is the maximum length of the string. At run-time, the user can enter up to 40 characters even though only 15 characters (display length) are shown in the field.

- *STF_NO_FLAGS* indicates that no special string flags are specified. This causes the string to be left justified.

- *WOF_BORDER* specifies that a simple border will surround the string edit region at run-time.

(See "Chapter 41—UIW_STRING" of the Programmer's Reference for more information about the string window object.)

Date field The next field appears on the right side of the top line in the window and is the date field:

```
+ new UIW_DATE(22, 1, 20, &UI_DATE(), "",
    DTF_NO_FLAGS, WOF_BORDER)
```

The first three parameters passed to this field specify the field's position and display width. (See the string field above). The other parameters passed to the date field object include:

- *&UI_DATE()* passes a pointer to the system date which is then copied into the date field's data. The UIW_DATE window object uses the UI_DATE object internally to store the date information.

- "" indicates that no date range is set. For example, if the dates were only valid from 1980 to 1999 then this is set to *"1-1-90..12-31-90."*

- *DTF_NO_FLAGS* indicates that no special date field flags are specified.

- *WOF_BORDER* specifies that a simple border will surround the string edit region at run-time.

(See "Chapter 3—UI_DATE" and "Chapter 28—UIW_DATE" in the Programmer's Reference for more information about the date object and the date window object.)

Text field The last field added to notepad1 is the text field (used for the note itself). The text field object is very similar to the string field. The two parameters which are different are shown below:

```
+ new UIW_TEXT(2, 3, 60, 4, "", 1028, TXF_NO_FLAGS,
    WOF_BORDER);
```

- *4* is the height of the text edit region.

- *TXF_NO_FLAGS* indicates that no special text field flags are specified.

(See "Chapter 43—UIW_TEXT" of the Programmer's Reference for more information about the text window object.)

Adding the notepads to the window manager

The two notepad windows are added to the window manager using the UI_WINDOW_MANAGER::operator+ operator overload.

```
*windowManager
    + notepad1
    + notepad2;
```

The last window added to the window manager is always the current window at run-time.

The event loop

After both notepads have been set up, the following loop is used to receive input from the user:

```
// Wait for user response.
int ccode;
UI_EVENT event;
do
{
    eventManager->Get(event, Q_NORMAL);
    if (event.type == E_KEY && event.rawCode == ESCAPE)
        event.type = L_EXIT;
    ccode = windowManager->Event(event);
} while (ccode != L_EXIT);
```

This code segment is almost the same (the '->' is used instead of '.') as the event loop section in the "Hello World!" tutorial.

Clean up

At the end of the program you must destroy the managers, the display and other objects you have created. Any objects that have been added to the event manager or window manager are automatically destroyed by the manager's destructor routine which is called when the manager is destroyed. The delete operator calls the class destructor and de-allocates memory used by the class object. The following code deletes the display screen, event manager, window manager and window help system:

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

The order in which the screen display, event manager, window manager, and other window objects are created and destroyed is very important. The event manager constructor depends on the existence of the display object; likewise, the window manager constructor depends on the existence of the event manager and the help system depends on the window manager. Be sure to destroy all objects (using the delete operator) in the opposite order.

In this tutorial the help system is created last and deleted first. The window manager is the next to be destroyed and it in turn destroys all windows and window objects attached to it. The event manager is created second and deleted second from last along with all objects attached to it. Finally, the display object is deleted last because it was constructed first.

If the screen display, event manager and window manager had been created in scope (as in "Hello World!") then they would be automatically destroyed in the opposite order in which they were created.

Run-time features

Each of the notepads created in the program can be moved and sized like the "Hello World!" window was at run-time. In addition, the following actions can be performed at run-time:

Next Window—Pressing the left mouse button with the mouse pointer on one of the notepads selects that notepad as the current window. The current window is indicated by the color of the title bar at the top of the window. Also, pressing <Alt F6> cycles from the current window to the next window.

Next Field—Pressing the left mouse button with the mouse pointer on one of the fields selects that as the current field. The current field is indicated by the color of the field region and the cursor. Also pressing <Tab> cycles from the current field to the next field.

Mark—Moving the mouse pointer to one of editable fields and "dragging" the mouse across the text or pressing <Ctrl F5> and using the arrow keys will mark that text region. This is usually followed by cutting or copying the information.

Cut—After marking a region of text in an editable field, clicking the right mouse button or pressing <Ctrl F6> will cut (erase) the text from the region and place it in the global paste buffer.

Copy—After marking a region of text in an editable field, clicking the right mouse button with the left button still held down or pressing <Ctrl F7> will copy the text and place it in the global paste buffer.

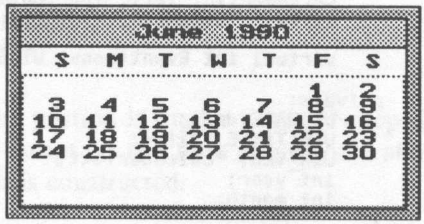
Paste—After cutting or copying, clicking the right mouse button or pressing <Ctrl F6> will paste (copy) the text into the current field from the global paste buffer. To paste the text in another field or another window (in the other notepad) first select the window and field and then paste the information.

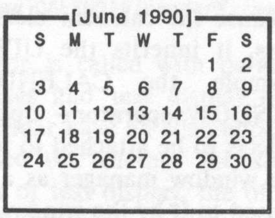
Undo—Pressing <Ctrl F9> or <Alt> in combination with the left mouse button undoes the most recent changes in the current field. Each field has its own undo buffer. (See "Chapter 4—Default Event Mapping" for more information about undo.)

Redo—Pressing <Ctrl F10> or <Alt> and the right mouse button restores the most recent changes in the current field. This function allows you to 'undo' the undo action. (See "Chapter 4—Default Event Mapping" for more information about redo.)

Edit—After selecting a field, any key typed will be entered into the field. When you leave the date field (select another field) the date editor checks the validity of any entered date and formats it into the format specified by the programmer.

This tutorial is the first of two parts of the complete calendar program. In this tutorial you will learn about deriving a programmer defined object from a window object. The final program will produce a calendar with the current month on the screen at run-time similar to one of the following, depending on the graphics or text display mode:





The second part, "The Custom Application" tutorial, contains information about how to modify the color palettes and event mappings. The final application will allow you to use the <PgUp> and <PgDn> keys to change the months displayed on the calendar. Also, the color combinations will be changed.

The code for the first "Calendar" program is located in \ZINC-\TUTORIAL\CALENDR1.CPP. Be sure that the C++ compiler directory is in the path. The executable program is made by typing "make calendar.exe" at the DOS command line while in the TUTORIAL directory.

Creating the calendar

The calendar program is created by deriving a CALENDAR class object from the UIW_WINDOW class. Deriving objects from ZIL class objects takes advantage of the inheritance features of C++. Inheritance allows customization of some member functions and complete inheritance of others. The CALENDAR class used in this tutorial contains member functions to construct the class, interpret events received from the event manager and inherited window functions to manipulate the window.

The following is the CALENDAR class definition:

```
class CALENDAR : public UIW_WINDOW
{
public:
    CALENDAR(int left, int top, int offset);
    virtual ~CALENDAR(void) {}

    virtual int Event(const UI_EVENT &event);

private:
    UI_DATE date;
    UIW_TITLE *title;
    UIW_TEXT *calendarText;
    int year;
    int month;
};
```

Because the calendar class above is derived from a UIW_WINDOW class, it inherits the UIW_WINDOW class object attributes. For example, the CALENDAR class object inherits the **UIW_-WINDOW::operator+** operator overload that enables other window objects to be attached to it. In addition, the calendar can be added to the window manager as a window. The calendar will receive events passed to it by the window manager when it is the current window. It is always the current window in this program since it is the only window on the screen.

The calendar is constructed (by calling the calendars constructor CALENDAR::CALENDAR) in the main program below:

```
main()
{
    // Initialize the display.
    UI_DISPLAY *display = new UI_DOS_BGI_DISPLAY();
    if (!display->installed)
    {
        delete display;
        display = new UI_DOS_TEXT_DISPLAY();
    }

    // Initialize the event and window managers.
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100,display);
    *eventManager
        + new UI_BIOS_KEYBOARD + new UI_MS_MOUSE + new UI_CURSOR;
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);

    // Create the calendar.
    int centerX = display->columns / display->cellWidth / 2;
    int centerY = display->lines / display->cellHeight / 2;
```

```
    int offset = (display->isText) ? 1 : 0;
    *windowManager + new CALENDAR(centerX, centerY, offset);

    // Process the events.
    int ccode;
    UI_EVENT event;
    do
    {
        eventManager->Get(event, Q_NORMAL);
        ccode = windowManager->Event(event);
    } while (ccode != L_EXIT);

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

The main program portion is similar to the "Notepad" program in the previous tutorial except for two lines (highlighted above). In these two lines the calendar is constructed:

```
    int offset = (display->isText) ? 1 : 0;
    *windowManager + new CALENDAR(centerX, centerY, offset);
```

The calendar constructor is called with parameters that specify the screen center coordinates and text display offset. The *offset* (not required) helps the calendar have an even appearance in both graphics and text modes. Based on whether the screen display has been initialized as a graphics or text display, the offset is determined using the screen *display* member variable *display->isText*.

The screen member functions for the calendar (to create, move and size the window) are inherited from the UIW_WINDOW base class. Two additional member functions are required:

CALENDAR::CALENDAR—This constructor initializes the calendar information and constructs the calendar window.

CALENDAR::Event—This member function interprets events sent to the calendar window by the window manager.

Calendar constructor

The first, the **CALENDAR::CALENDAR** constructor, is used to create and add window objects to the calendar window.

```
CALENDAR::CALENDAR(int centerX, int centerY, int offset) :
    UIW_WINDOW(centerX - 11, centerY - 4, 24, 8 + offset,
        WOF_NO_FLAGS, WOAF_NO_SIZE)
{
    // Get the current year and month.
    date.Export(&year, &month, 0, 0);
```

```
// Create the window objects.
*this
    + new UIW_BORDER
    + (title = new UIW_TITLE("", WOF_JUSTIFY_CENTER))
    + new UIW_STRING(0, 0, 24, " S  M  T  W  T  F  S", 23,
        STF_NO_FLAGS, WOF_VIEW_ONLY | WOF_NON_SELECTABLE)
    + (calendarText = new UIW_TEXT(0 + offset, 1,
        24 - offset, 6, "", 256, WOF_VIEW_ONLY |
        WOF_NON_SELECTABLE | (WOF_BORDER * !offset)));

// Initialize the current months calendar.
UI_EVENT event;
event.type = L_CURRENT_MONTH;
Event(event);
}
```

Initializing the base class

The base class UIW_WINDOW constructor is called to create the calendar window:

```
UIW_WINDOW(centerX - 11, centerY - 4, 24, 8 + offset,
    WOF_NO_FLAGS, WOAF_NO_SIZE)
```

It is passed parameters that specify the position and size of the window to be created. In addition, the *WOAF_NO_SIZE* flag specifies that the window size cannot be changed by the user at run-time. This flag is used because the size of the calendar will not change (i.e., there are only seven days in a week). (See "Chapter 48—UIW_WINDOW" of the Programmer's Reference for a description of other advanced window object flags.)

Adding window objects

Adding the window objects is the next step inside the CALENDAR constructor:

```
// Create the window objects.
*this
    + new UIW_BORDER
    + (title = new UIW_TITLE("", WOF_JUSTIFY_CENTER))
    + new UIW_STRING(0, 0,24, " S  M  T  W T  F  S", 23,
        STF_NO_FLAGS, WOF_VIEW_ONLY | WOF_NON_SELECTABLE)
    + (calendarText = new UIW_TEXT(0 + offset, 1,
        24 offset, 6, "", 256,WOF_VIEW_ONLY |
        WOF_NON_SELECTABLE | (WOF_BORDER * !offset)));
```

All four window objects have been used in the two previous tutorials with the following differences:

1—The UIW_TITLE object is assigned to the CALENDAR member variable *title* before being added to the calendar window. The title bar will contain the month name and must be updated periodically.

2—The two window object flags *WOF_VIEW_ONLY* and *WOF_-NON_SELECTABLE* are combined together in both the string and text field constructors. This indicates that the field cannot be edited nor selected.

3—The UIW_TEXT field is assigned to the CALENDAR member variable *calendarText* before being added to the calendar window. It displays the days of the month in the field and is updated when the month is changed.

Event function

The **CALENDAR::Event** member function is used to process events sent from the window manager. In the next tutorial, the event map table is redefined to include mapping for the logical events of L_PREVIOUS_-MONTH and L_NEXT_MONTH. These are programmer defined events and are assigned numbers between 10,000 and 99,000 in the program. Inside the Event function these events are interpreted and some action performed.

```
const int L_PREVIOUS_MONTH              = 10000;
const int L_NEXT_MONTH                  = 10001;
const int L_CURRENT_MONTH               = 10002;
```

These events are used to update the calendar string inside the text field and the month name in the title bar.

The **CALENDAR::Event** function below maps real device events (e.g., pressing a key) to logical events (e.g., L_NEXT_MONTH) and performs some action according to what was pressed:

```
int CALENDAR::Event(const UI_EVENT &event)
{
    static char text[256];
    static char monthString[20];

    // Switch on the event type.
    int ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_CALENDAR);
    switch (ccode)
    {
    case L_CURRENT_MONTH:
        .
        .
        .

        // Change the window data to reflect new month.
        title->DataSet(monthString);
        calendarText->DataSet(text, 256);
        break;
```

```
	default:
		// Call the window event to process other events.
		ccode = UIW_WINDOW::Event(event);
		break;
	}
	// Return the control code.
	return (ccode);
}
```

The CALENDAR::Event function consists of three parts:

1—Calling the *LogicalEvent* function to map the device events to logical events. In the next tutorial you will learn how to change the event map table to map keys and mouse events to the logical events listed above.

2—Interpreting the logical event and performing some action because of it. This includes using the DataSet functions of the various fields to update the information. (See "Chapter 28—UIW_-DATE" and "Chapter 48—UIW_TEXT" of the Programmer's Reference for more information about DataSet.)

3—Calling the **UI_WINDOW::Event** function of the inherited class. All events that are not specific to the calendar class are passed to the base class Event function to be interpreted. For example, the events that relate to moving the window would be passed to **UI_WINDOW::Event**.

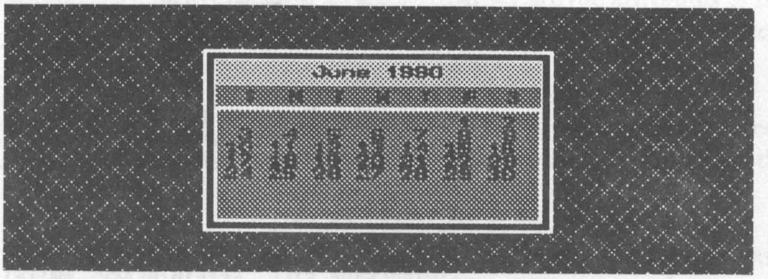
The Custom Application

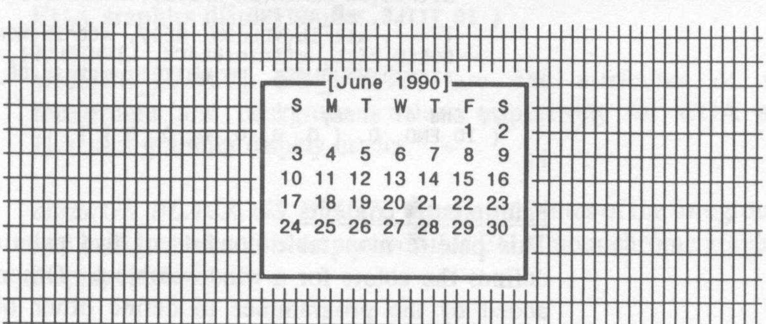
This tutorial, the second part of the calendar program, demonstrates:

- Setting up the color palette map table for customized colors.
- Changing the background color palette.
- Adding entries to and modifying the event map table (mouse and keyboard events) for an application.

This tutorial is a continuation of the "Calendar" tutorial. It is assumed that you have studied the previous tutorial before beginning this one. The same calendar program is used with emphasis on how the map tables are set up.

The final program will produce a screen similar to one of the following, depending on the graphics or text display mode:





Palette mapping

The Zinc Interface Library provides two ways of defining the color combinations associated with a window object. The first, global color palette mapping, determines the default color combinations of window objects on the screen. The second, individual window object mapping, is attached to a window or window object and determines color combinations for that specific window or window object.

In this tutorial, the second method, individual window object mapping, is used. The following palette map table is attached to the calendar. It is used at run-time to define the color mapping of the calendar window and its attached objects:

```
// Palette map table
static UI_PALETTE_MAP calendarPaletteMapTable[] =
{
    // ID_WINDOW_OBJECT
    { ID_WINDOW_OBJECT, PM_ANY,
        { ' ', attrib(BLACK, RED), attrib(MONO_NORMAL,MONO_BLACK),
        SOLID_FILL, attrib(BLACK, WHITE),
        attrib(BW_WHITE, BW_BLACK), attrib(GS_WHITE, GS_BLACK)} },

    // ID_STRING
    { ID_STRING, PM_ANY,
        { ' ', attrib(BLACK, RED), attrib(MONO_NORMAL, MONO_BLACK),
        SOLID_FILL, attrib(BLACK, RED),
        attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)} },

    // ID_BORDER
    { ID_BORDER, PM_ANY,
        { ' ', attrib(BLUE, RED), attrib(MONO_HIGH, MONO_BLACK),
        SOLID_FILL,attrib(LIGHTGRAY,BLUE),
        attrib(BW_WHITE, BW_BLACK), attrib(GS_WHITE, GS_GRAY) } },

    // ID_TITLE
    { ID_TITLE, PM_ANY,
        { ' ', attrib(WHITE, LIGHTGRAY),
        attrib(MONO_NORMAL,MONO_BLACK),
        INTERLEAVE_FILL, attrib(BLACK, WHITE),
        attrib(BW_BLACK,BW_WHITE), attrib(GS_BLACK, GS_WHITE) } },
    { ID_TITLE, PM_ACTIVE,
        { ' ', attrib(YELLOW, RED), attrib(MONO_NORMAL,MONO_BLACK),
        SOLID_FILL, attrib(BLACK, LIGHTBLUE),
        attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE) } },

    // End of array
    { ID_END, 0, { 0, 0, 0, 0, 0, 0 } }
};
```

Palette entry contents

This palette map table consists of five palette entries. Each entry defines the colors for a window object. Other palette entries can be added by the programmer to define other objects if needed. For example, the fourth palette in the palette map table defines the colors to be used for a string window object:

```
// ID_STRING
{ ID_STRING, PM_ANY,
    ' ', attrib(BLACK, RED), attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(BLACK, RED),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)} },
```

This palette structure contains the following items:

- *ID_STRING* is the window object identification. It indicates that this palette mapping is used for a window object with the ID_STRING identification (i.e., UIW_STRING objects).

- *PM_ANY* is a status flag that indicates the status that an object needs in order to use this particular palette.

- *' '* is the text fill character. It is used to fill all blank space on the window object when the screen display is created in text mode.

- *attrib(BLACK, RED)* are the attributes of the foreground and background colors respectively for color text display mode.

- *attrib(MONO_NORMAL, MONO_BLACK)* are the attributes of the foreground and background colors respectively for monochrome text display mode.

- *SOLID_FILL* is the graphics fill pattern. It is used when the screen display is created in graphics mode to fill all blank space on the window object.

- *attrib(BLACK, RED)* are the attributes of the foreground and background colors respectively for VGA, VGA monochrome and EGA graphics display modes.

- *attrib(BW_BLACK, BW_WHITE)* are the attributes of the foreground and background colors respectively for CGA and Hercules graphics display modes.

- *attrib(GS_BLACK, GS_WHITE)* are the attributes of the foreground and background colors respectively for EGA monochrome graphics display mode.

<u>Palette identification</u>

Each window object class has its own unique identification and understands the identifications of the base class window objects from which it has been derived. The UIW_STRING object is derived from the UIW_WINDOW_OBJECT class and thus has a unique identification of ID_STRING, but it also understands that it is derived from an object with an identification of ID_WINDOW_OBJECT.

All window objects that do not have a unique identification listed in the palette map will use the next identification that the window object understands. For example, the text object that is added to the calendar window later in this tutorial has a unique identification of ID_TEXT. ID_TEXT it is not listed in the palette map table above, so the text object defaults to ID_STRING, since it is derived from the UIW_STRING class. If an ID_STRING entry did not exist then the text object would default to ID_WINDOW_OBJECT. (See "Chapter 20—UI_PALETTE_MAP" of the Programmer's Reference for more information about the palette map table.)

<u>Palette status</u>

The status of an object is also important in determining which palette in the palette map table is used. In each palette entry, the second item (e.g., PM_ANY) determines what status an object needs to use the palette:

```
// ID_TITLE
{ ID_TITLE, PM_ANY,
    { ' ', attrib(WHITE, LIGHTGRAY),
    attrib(MONO_NORMAL,MONO_BLACK),
    INTERLEAVE_FILL, attrib(BLACK, WHITE),
    attrib(BW_BLACK,BW_WHITE), attrib(GS_BLACK, GS_WHITE) } },
{ ID_TITLE, PM_ACTIVE,
    { ' ', attrib(YELLOW, RED), attrib(MONO_NORMAL,MONO_BLACK),
    SOLID_FILL, attrib(BLACK, LIGHTBLUE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE) } },
```

When the title status is active (e.g., the window to which it is attached is the current window), the second palette entry (with the PM_ACTIVE status) is used. The title object uses the first palette with PM_ANY to obtain color combinations for any other status. (See "Chapter 20—UI_PALETTE_MAP" for a complete listing of possible status indicators.)

<u>Assigning specific palette information</u>

In the **CALENDAR::CALENDAR** constructor, one line is added to attach the palette map table described above to the calendar window. After the window objects have been added to the window, the calendar is assigned a unique palette map table:

```
paletteMapTable = calendarPaletteMapTable;
```

The calendar will look in this table to get color information for the calendar window and all objects attached to it. The global palette map table is used if no unique palette map table is defined (as in the previous tutorial).

Background palette

The palette associated with the screen display background can also be modified to allow different color combinations. The global variable *_backgroundPalette* points to the current background palette. In this tutorial it is reassigned to the following *calendarBackgroundPalette* defined in the program:

```
// Background palette.
static UI_PALETTE calendarBackgroundPalette = { '\305',
    attrib(CYAN, BLACK), attrib(MONO_DIM, MONO_BLACK),
    XHATCH_FILL, attrib(CYAN, CYAN),
    attrib(BW_WHITE, BW_WHITE), attrib(GS_GRAY, GS_GRAY) };
UI_PALETTE *_backgroundPalette = &calendarBackgroundPalette;
```

The background palette contains the same structure as a palette entry in the palette map table. For this example the background is filled with a cross-hatch pattern created using the '┼' (ascii 305 octal) character for text mode and the XHATCH_FILL pattern for graphics. The global default background palette is used if no other background palette is defined (as in the previous tutorial).

See "Chapter 5—Default Palette Mapping" of the Programmer's Guide for a listing of the default color combinations available in ZIL. The following files, included in the \ZINC\UTIL directory, give complete listings of the palette map tables:

G_PNORM—Contains the normal default palette map table used for general windows and window objects.

G_PHELP.CPP—Contains the help window palette map table used for the help window system.

G_PERROR.CPP—Contains the error window palette map table used for the error window system.

G_PBACK.CPP—Contains the default background palette.

These files can be modified directly and linked in with your application to change the normal, help and error default palette mapping.

Event mapping

Like the palette mapping, logical event mapping is done through the event map table. Raw events received from the various input devices at run-time are interpreted at each level of the application according to the type of operation. The simplified event map table below contains events and the logical events to which they are mapped:

```
// Event map table -
// { windowID, logicalValue, event.type, event.rawCode }
static UI_EVENT_MAP myEventMapTable[] =
{
	// ID_WINDOW_MANAGER
	{ ID_WINDOW_MANAGER, L_EXIT, E_KEY, SHIFT_F3 },
	{ ID_WINDOW_MANAGER, L_EXIT, E_KEY, ESCAPE },
	{ ID_WINDOW_MANAGER, L_EXIT, E_MOUSE, M_LEFT | M_RIGHT},
	{ ID_WINDOW_MANAGER, L_WINDOW_MOVE, E_KEY, ALT_F7 },

	// ID_WINDOW_OBJECT
	{ ID_WINDOW_OBJECT, L_SELECT, E_KEY, ENTER },
	{ ID_WINDOW_OBJECT, L_SELECT, E_KEY, GRAY_ENTER },
	{ ID_WINDOW_OBJECT, L_VIEW, E_MOUSE, 0 },
	{ ID_WINDOW_OBJECT, L_BEGIN_SELECT, E_MOUSE,
	  M_LEFT | M_LEFT_CHANGE },
	{ ID_WINDOW_OBJECT, L_CONTINUE_SELECT, E_MOUSE, M_LEFT },
	{ ID_WINDOW_OBJECT, L_END_SELECT, E_MOUSE, M_LEFT_CHANGE },

	// ID_CALENDAR
	{ ID_CALENDAR, L_PREV_MONTH, E_KEY, WHITE_PGUP },
	{ ID_CALENDAR, L_PREV_MONTH, E_KEY, GRAY_PGUP },
	{ ID_CALENDAR, L_NEXT_MONTH, E_KEY,WHITE_PGDN },
	{ ID_CALENDAR, L_NEXT_MONTH, E_KEY,GRAY_PGDN },

	// End of array.
	{ ID_END, 0, 0, 0 }
};
UI_EVENT_MAP *_eventMapTable = myEventMapTable;
```

<u>Event map entry contents</u>

Each event map entry contains information about how the event should be interpreted. For example, the key <Shift><F3> is mapped to the L_EXIT message:

```
{ ID_WINDOW_MANAGER, L_EXIT, E_KEY, SHIFT_F3 },
```

The entry above contains the following information:

- *ID_WINDOW_MANAGER* indicates that this entry should be used to interpret events for the window manager.

- *L_EXIT* indicates that the event should be interpreted as an exit message.

- *E_KEY* indicates that the information is for a key type event.
- *SHIFT_F3* indicates that <Shift><F3> is to be interpreted to be the L_EXIT message above.

At run-time, events are interpreted at each level of operation. For example, pressing <Shift><F3> puts an E_KEY event with a raw code of SHIFT_F3 on the event queue. This event is then passed to the window manager. The window manager, since it performs the highest level of operation, tries to interpret the event first and then passes it on to other window objects. This event (<Shift><F3>) is mapped to the L_EXIT logical event which is a message that causes the program to end.

Multiple mappings

Each logical event can be mapped to various input device events. In this tutorial, three different device events are mapped to the L_EXIT logical event:

```
{ ID_WINDOW_MANAGER, L_EXIT, E_KEY, SHIFT_F3 },
{ ID_WINDOW_MANAGER, L_EXIT, E_KEY, ESCAPE },
{ ID_WINDOW_MANAGER, L_EXIT, E_MOUSE, M_LEFT | M_RIGHT },
```

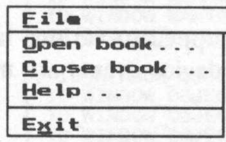
Pressing <Shift><F3>, <Esc>, or holding down both the left and right mouse buttons at run-time cause an L_EXIT message to be generated and sent to objects with the ID_WINDOW_MANAGER identification (i.e., window manager).

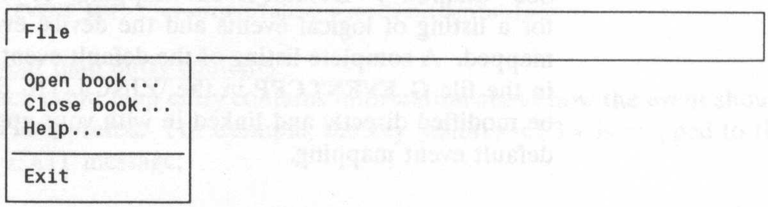
See "Chapter 4—Default Event Mapping" of the Programmer's Guide for a listing of logical events and the device events to which they are mapped. A complete listing of the default event mapping can be found in the file **G_EVENT.CPP** in the \ZINC\UTIL directory. This file can be modified directly and linked in with your application to change the default event mapping.

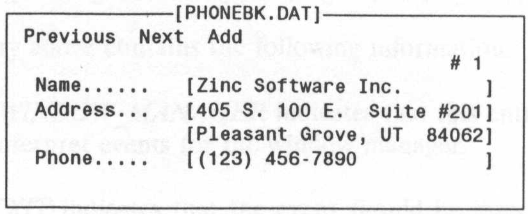
This tutorial program creates a simple data base containing addresses and phone numbers. The program illustrates the following features:

- Creating and using control menus.
- Saving information entered at run-time in window fields.
- Displaying custom error messages.

The final program will produce a screen similar to one of the following, depending on the graphics or text display mode:







It is assumed that you have read about deriving new class objects in the "Calendar" tutorial earlier in this chapter. The code for the "Phone Book" program is located in \ZINC\TUTORIAL\PHONEBK.CPP. Be sure that the C++ compiler directory is in the path. The executable program is made by typing "make phonebk.exe" at the DOS command line while in the TUTORIAL directory.

Creating the phone book

The phone book program, like the calendar program in the previous tutorial, is created using an object derived from the UIW_WINDOW class. The PHONE_BOOK class structure shown below contains all of the information and member functions necessary to read, write and display names and addresses from a disk file:

```
class PHONE_BOOK : public UIW_WINDOW
{
public:
    static int fileHandle;

    PHONE_BOOK(char *filename, int left, int top);
    ~PHONE_BOOK(void);

    static void PHONE_BOOK::AddRecord(void *item, UI_EVENT &event);
    static void PHONE_BOOK::NextRecord(void *item,UI_EVENT &event);
    static void PHONE_BOOK::PrevRecord(void *item,UI_EVENT &event);
private:
    static int newRecord;
    static int recordNumber;
    static PHONE_RECORD record;
    static PHONE_RECORD tmpRecord;

    static void PHONE_BOOK::ReadRecord(void);
    static void PHONE_BOOK::WriteRecord(void);
};
```

This phone book class inherits the UIW_WINDOW class object attributes.

The phone book constructor, listed below, initializes the phone book window and each of the data entry fields attached to it.

```
PHONE_BOOK::PHONE_BOOK(char *filename, int left, int top) :
    UIW_WINDOW(left - 20, top - 5, 40, 9, WOF_NO_FLAGS,
    WOAF_NO_FLAGS, HELP_RECORD)
{
    newRecord = FALSE;
    record.name[0] =
        record.address1[0] =
        record.address2[0] =
        record.phone[0] = '\0';
    tmpRecord = record;

    // Open the file and read the first record.
    fileHandle = open(filename, O_RDWR);
    if (fileHandle < 0)
```

```
    fileHandle = open(filename, O_CREAT, S_IREAD | S_IWRITE);
  if (fileHandle < 0)
    return;
ReadRecord();

// Create the window menu.
UIW_PULLDOWN_MENU menu(0, WOF_NO_FLAGS, WOAF_NO_FLAGS);
menu
    + new UIW_PULL_DOWN_ITEM(" ~Previous ", MNIF_NO_FLAGS,
        PHONE_BOOK::PrevRecord)
    + new UIW_PULL_DOWN_ITEM(" ~Next ", MNIF_NO_FLAGS,
        PHONE_BOOK::NextRecord)
    + new UIW_PULL_DOWN_ITEM(" ~Add ", MNIF_NO_FLAGS,
        PHONE_BOOK::AddRecord);

// Create the phone book record entry window.
*this
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE(filename, WOF_JUSTIFY_CENTER)
    + &menu

    + new UIW_PROMPT(32, 0, "#", WOF_NO_FLAGS)
    + new UIW_NUMBER(34, 0, 6, &recordNumber, "", NMF_NO_FLAGS,
       WOF_NO_ALLOCATE_DATA | WOF_NON_SELECTABLE)

    + new UIW_PROMPT(2, 1, "Name......", WOF_NO_FLAGS)
    + new UIW_STRING(15, 1, 21, tmpRecord.name, 20,
       STF_NO_FLAGS, WOF_BORDER | WOF_NO_ALLOCATE_DATA)

    + new UIW_PROMPT(2, 2, "Address...", WOF_NO_FLAGS)
    + new UIW_STRING(15, 2, 21, tmpRecord.address1, 20,
       STF_NO_FLAGS, WOF_BORDER | WOF_NO_ALLOCATE_DATA)
    + new UIW_STRING(15, 3, 21, tmpRecord.address2, 20,
       STF_NO_FLAGS, WOF_BORDER | WOF_NO_ALLOCATE_DATA)

    + new UIW_PROMPT(2, 4, "Phone.....", WOF_NO_FLAGS);
    + new UIW_FORMATTED_STRING(15, 4, 21, tmpRecord.phone,
       "LNNNLLNNNLXXXX", "(...) ...-....",
       WOF_BORDER | WOF_NO_ALLOCATE_DATA);

// Make the ( File | Close ) option active and re-display menu.
closeOption->woFlags &= ~WOF_NON_SELECTABLE;
}
```

This constructor performs the following steps to set up a phone book:

- Clears the record information structures.
- Opens or creates the phone book file specified by the user. The file name is passed as a parameter to the constructor.
- Initializes the control menu to be added to the window. Each menu item that is created and added to the menu has an associated function that is performed when the item is selected. The menu and menu items are discussed in more detail later in this tutorial.

- Creates and adds the fields to the window for entering names, addresses and phone numbers at run-time.

- Modifies the main control menu item (File | Close) to be a selectable item. This allows the phone book file to only be closed after it is opened.

The following are the other member functions which are used to perform operations on the phone book:

~PHONE_BOOK—The destructor closes the phone book file.

AddRecord, NextRecord and PrevRecord—These three member functions allow the user to move through the phone book file at run-time to enter or modify phone number records.

ReadRecord and WriteRecord—These two functions read and write the records respectively from the phone book file.

Besides the PHONE_BOOK constructor, each of the functions above and the simple algorithms used to read and write to a file will not be described. Refer to the file **\ZINC\TUTORIAL\PHONEBK.CPP** for a complete listing of the program.

User supplied data

In order to display the phone numbers located in a phone book file, the information must be read and then placed in the window fields. There are two ways to move data from window fields to a file or memory structure:

- Copying the new information into a field using the field's **DataSet** function and then retrieving the information from the field (after it has been modified by the user at run-time) using the field's **DataGet** function. This method was used in the **CALENDAR::Event** in the previous tutorial to update the new month name to the window title.

- Passing a pointer to the actual record as a parameter to the field's constructor and using the WOF_NO_ALLOCATE_DATA flag. This method allows the data to be tied directly to the field. When the field is destroyed (when the window manager and window are destroyed) the data pointed to by the actual record is not destroyed.

In the phone book program the second method is used, because it allows much more simplicity in this case. For example, the name field is tied directly to the *tmpRecord.name* by using the following:

```
+ new UIW_PROMPT(2, 1, "Name......", WOF_NO_FLAGS)
+ new UIW_STRING(15, 1, 21, tmpRecord.name, 20,
    STF_NO_FLAGS, WOF_BORDER | WOF_NO_ALLOCATE_DATA)
```

The *tmpRecord.name* contains the initial string that will appear in the field at run-time. When the user edits the name, the information is changed directly in *tmpRecord.name.*

If the flag WOF_NO_ALLOCATE_DATA were not used in the example above then the information in *tmpRecord.name* would be copied to a temporary buffer that is allocated by the string editor. The information in this temporary buffer could then be transferred back to the *tmpRecord.name* (after the user had changed the name information) by using the following code segment:

```
UIW_STRING *field = UIW_STRING(15, 1, 21, tmpRecord.name, 20,
    STF_NO_FLAGS, WOF_BORDER);
window + field;
:
char *string = field->DataGet();
strcpy(tmpRecord.name, string);
```

Updating changed data

If the data in *tmpRecord.name* is changed in the program (e.g., the next name read from the file) then the screen display must be updated to reflect this change. The easiest way to accomplish this for all fields on the window, since all were changed when the new record was read, is to re-add the window to the window manager. If the window already exists, the window manager will re-display the window and all of its objects. The following code segment, taken from the **ReadRecord** function, accomplishes this:

```
if (book)
    *_windowManager + book;
```

Using the data acquisition routines **DataGet** and **DataSet** automatically updates the changed field.

Initialization

The main procedure for the phone book program is very similar to all of the other tutorial main procedures. It consists of the following steps:

1—Initialize the screen display. If it cannot be constructed as a graphics display then it is constructed as a text display.

2—Construct the event manager and window managers. The keyboard, mouse and cursor devices are also added to the window manager in this step.

3—Initialize the help window system and error window system. The help file is set up to be read from **PHONEBK.HLP** with the help context HELP_GENERAL used as the general help.

4—Create the main control menu. This sets up the menu along the screen top that can be accessed at all times.

5—Wait for the user to respond. This loop continually accepts events from the input devices and then passes them to the window manager to be processed until an exit message is received.

6—Delete all objects in the opposite order in which they were constructed. The window manager and event managers automatically delete all objects attached to them when they are deleted. Objects that are created in scope are deleted at the end of the main.

Creating menus

The phone book program consists of two pull down menus. The first is the main menu that controls the program usage. It is constructed and attached directly to the window manager. At run-time, the user can access the menu at all times. The following code segment inside the **CreateMenu** procedure constructs the main menu:

```
static void CreateMenu(void)
{
    // Create the main control menu.
    controlMenu = new UIW_PULL_DOWN_MENU(0, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    controlMenu->woAdvancedFlags |= WOAF_LOCKED | WOAF_NON_CURRENT;

    // ( File ) option pull down menu (Close is inactive).
    UIW_PULL_DOWN_ITEM *fileOption = new UIW_PULL_DOWN_ITEM(
        " ~File ", MNF_NO_FLAGS, 0);
    closeOption = new UIW_POP_UP_ITEM("~Close book...",
        MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS, CloseBook);
    closeOption->woFlags |= WOF_NON_SELECTABLE;
    *fileOption
        + new UIW_POP_UP_ITEM("~Open book...", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS, OpenBook)
        + closeOption
        + new UIW_POP_UP_ITEM("~Help...", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS, Help)
```

```
        + new UIW_POP_UP_ITEM
        + new UIW_POP_UP_ITEM("E~xit", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS, Exit);

    // Add the option menus to the control menu.
    *controlMenu + fileOption;
    *_windowManager + controlMenu;
}
```

One pull down item for 'File' operations is created and added to the window. Four pop up items are added to the 'File' option to create the pull down option list. The **UIW_PULL_DOWN_ITEM::operator+** operator overload is used to add the options to the 'File' option and the **UIW_PULL_DOWN_MENU::operator+** operator overload is used to add the 'File' option to the control menu. The control menu is then added directly to the window manager using the **UI_WINDOW_-MANAGER::operator+** operator overload.

Each of the pop up items added to the 'File' option are constructed with five parameters that indicate how it is displayed and what action takes place when it is selected:

```
+ new UIW_POP_UP_ITEM("~Open book...", MNIF_NO_FLAGS,
    BTF_NO_TOGGLE, WOF_NO_FLAGS, OpenBook)
```

- "~*Open book...*" is the text displayed in that menu option. The 'hot key' is preceded by a ~ . The option can be selected by clicking the mouse on the option region or pressing <Alt> in combination with the 'hot key.'

- *MNIF_NO_FLAGS* indicates that no special menu item flags are specified.

- *BTF_NO_TOGGLE* indicates that the item is not to be toggled on and off.

- *WOF_NO_FLAGS* indicates that no special window object flags are specified.

- *OpenBook* is the function that will be called when the user selects this menu option at run-time. This function is called with two parameters; the event that selected the item and a pointer to the pop up item object.

The 'File | Close' option above is set to be non-selectable in the **CreateMenu** procedure:

```
closeOption->woFlags |= WOF_NON_SELECTABLE;
```

At run-time the user can see this option but is not able to select it. After a phone book file is opened in the program this option is changed to be selectable as shown below:

```
closeOption->woFlags &= ~WOF_NON_SELECTABLE;
```

The second menu is used inside the phone book data entry window to move between records. It is set up in the same way as the control menu above. The second menu is added to the window instead of the window manager and is located below the title bar inside the window.

Menu functions

When one of the menu items described above is selected, a function is called to perform some action. For example, when the 'File | Help' option is selected the following function is called:

```
// Control menu ( File | Help ) option to _display general help.
#pragma argsused
static void Help(void *item, UI_EVENT &event)
{
	// Call the help system to display general help.
	_helpSystem->DisplayHelp(_windowManager, HELP_GENERAL);
}
```

This function then calls the help system to display the general help context information.

Another example, listed below, places an exit message on the event queue when the 'File | Exit' option is selected:

```
// Control menu ( File | Exit ) option to exit the program.
#pragma argsused
static void Exit(void *item, UI_EVENT &event)
{
	// Put an EXIT message on the event queue.
	event.type = L_EXIT;
	_eventManager->Put(event, Q_BEGIN);
}
```

Error messages

In the second tutorial, using the "Notepad" program, you learned how to initialize the error window system. You can define and display your own error messages along with those automatically associated with the field editors. The code segment below displays an error message if the phone book file cannot be created:

```
// Add the new phone book to the window manager if no error.
if (book->fileHandle < 0)
{
    _errorSystem->ReportError(_windowManager, -1,
        "Error opening file.");
    delete book;
}
```

(See "Chapter 11—UI_ERROR_WINDOW_SYSTEM" for more information about the ReportError function.)

Conclusion

This concludes the tutorial section of the Programmer's Guide. However, there are additional resources for your use. These include: the example programs found in the \ZINC\EXAMPLES subdirectory, the many sample applications found on the Zinc BBS, free telephone support, etc.

INDEX

L

M

N

O

P

R

S

T

U

W

Z

GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
    Copyright (c)  YEAR  YOUR NAME.
    Permission is granted to copy, distribute and/or modify this document
    under the terms of the GNU Free Documentation License, Version 1.3
    or any later version published by the Free Software Foundation;
    with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
    A copy of the license is included in the section entitled "GNU
    Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
    with the Invariant Sections being LIST THEIR TITLES, with the
    Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.