
Z
i
n
c
PROGRAMMER'S
REFERENCE
APPLICATION
FRAMEWORK™
VERSION 3.5

Bringing order to chaos. . .

Zinc® Application Framework™

Programmer's Reference

Version 3.5

Zinc Software Incorporated
Pleasant Grove, Utah

TABLE OF CONTENTS

INTRODUCTION

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

Class object information—This section (Introduction) contains documentation format for a sample class member function, the class hierarchy and include file (**.HPP**) information associated with class objects and structures available within Zinc Application Framework.

Class object references—This section (Chapters 1 through 70) contains short descriptions about the class objects (or structures), the available member variables and functions and the calling conventions used with the class object. All public and protected members have been documented. Private members are not documented and are subject to change.

Miscellaneous information—This section (Appendices A through F) contains support definitions, system event definitions, logical event definitions, class identifications, storage information and raw DOS scan code information.

UI_SAMPLE_CLASS::SampleFunction

Syntax

returnValue SampleFunction(type1 *parameter1*, type2 **parameter2*);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

NOTE: A blackened box indicates a supported environment.

Remarks

A brief description of what **SampleFunction( )** does.

- *returnValue*$_{out}$ gives a complete description of the return value. The subscript "out" indicates that the variable (the return value in this case) does not require an initial value and that it receives a value from the function.

- *parameter1*$_{in}$ gives a complete description of function parameter 1. The subscript "in" indicates that the variable requires an initial value and that it is not changed by the function.

- *parameter2*$_{in/out}$ gives a complete description of function parameter 2. The subscript "in/out" indicates that the variable requires an initial value, but that it may also receive a different value from the function.

Example

This section provides a coding example of how **SampleFunction( )** was used in the development of other library routines or development utilities. The function itself often appears in bold type within the example code.

CLASSES AND STRUCTURES

General purpose

```
attrib
FlagSet
FlagsSet
Max
Min
NULL
NULLF
NULLP
TRUE and FALSE
UCHAR
ULONG
USHORT
VOIDF
VOIDP

class UI_APPLICATION
class UI_BIGNUM
class UI_DATE
class UI_ELEMENT
class UI_INTERNATIONAL
class UI_LIST
class UI_LIST_BLOCK
class UI_PATH
class UI_PATH_ELEMENT
class UI_STORAGE
class UI_STORAGE_OBJECT
class UI_TIME
```

Screen display

```
struct UI_PALETTE
struct UI_PALETTE_MAP
struct UI_POSITION
struct UI_REGION

class UI_BGI_DISPLAY
class UI_DISPLAY
class UI_FG_DISPLAY
class UI_GRAPHICS_DISPLAY
class UI_MOTIF_DISPLAY
class UI_MSC_DISPLAY
class UI_MSWINDOWS_DISPLAY
class UI_OS2_DISPLAY
class UI_REGION_ELEMENT
class UI_REGION_LIST
class UI_TEXT_DISPLAY
```

Event management

```
struct UI_EVENT
struct UI_EVENT_MAP
struct UI_KEY
struct UI_POSITION
struct UI_REGION
struct UI_SCROLL_INFORMATION
```

```
class UI_DEVICE
class UI_EVENT_MANAGER
class UI_QUEUE_BLOCK
class UI_QUEUE_ELEMENT
class UID_CURSOR
class UID_KEYBOARD
class UID_MOUSE
class UID_PENDOS
```

Window management

```
struct UI_ITEM

class UI_WINDOW_MANAGER
class UI_WINDOW_OBJECT
class UIW_BIGNUM
class UIW_BORDER
class UIW_BUTTON
class UIW_COMBO_BOX
class UIW_DATE
class UIW_FORMATTED_STRING
class UIW_GROUP
class UIW_HZ_LIST
class UIW_ICON
class UIW_INTEGER
class UIW_MAXIMIZE_BUTTON
class UIW_MINIMIZE_BUTTON
class UIW_POP_UP_ITEM
class UIW_POP_UP_MENU
class UIW_PROMPT
class UIW_PULL_DOWN_ITEM
class UIW_PULL_DOWN_MENU
class UIW_REAL
class UIW_SCROLL_BAR
class UIW_STRING
class UIW_SYSTEM_BUTTON
class UIW_TEXT
class UIW_TIME
class UIW_TITLE
class UIW_TOOL_BAR
class UIW_VT_LIST
class UIW_WINDOW
```

Error system

```
class UI_ERROR_SYSTEM
```

Help system

```
class UI_HELP_SYSTEM
```

INCLUDE FILE HIERARCHY

UI_ENV.HPP

```
// Compiler/Environment Dependencies
// Borland/Turbo C++
// Zortech C++
// Microsoft C++ (7.0)
// HP-UX, CC (cfront from HP) and Motif
// MS-DOS
// ----- General Definitions
```

UI_GEN.HPP

```
#if !defined(UI_GEN_HPP)
#   define UI_GEN_HPP
#   if !defined(UI_ENV_HPP)
#       include <ui_env.hpp>
#   endif

// basic macro declarations
// basic typedef declarations
// NULL
// TRUE/FALSE
// UIF_FLAGS
// UIS_STATUS
// OBJECTID
// INFO_REQUEST
// UI_ELEMENT
// UI_LIST
// UI_LIST_BLOCK
// UI_INTERNATIONAL
// UI_BIGNUM
// NMF_FLAGS
// NMI_RESULT
// UI_DATE
// DTF_FLAGS
// DTI_RESULT
// UI_TIME
// TMF_FLAGS
// TMI_RESULT
// UI_PATH_ELEMENT & UI_PATH
// UI_STORAGE_OBJECT & UI_STORAGE
// UIS_FLAGS
// Other miscellaneous definitions
// macros, miscellaneous fixups and generic functions
#endif
```

UI_DSP.HPP

```
#if !defined(UI_DSP_HPP)
#   define UI_DSP_HPP
#   if !defined(UI_GEN_HPP)
#       include <ui_gen.hpp>
#   endif

// UI_POSITION
// UI_REGION, UI_REGION_ELEMENT, UI_REGION_LIST
// UI_PALETTE
// palette macro
```

```
// rgb colors
// monochrome
// black & white
// gray scale
// colors
// LOGICAL_PATTERN
// UI_DISPLAY
// LOGICAL_FONT
// IMAGE_TYPE
// UI_TEXT_DISPLAY
// TDM_MODE
// UI_GRAPHICS_DISPLAY
// UI_BGI_DISPLAY
// UI_MSC_DISPLAY
// UI_FG_DISPLAY
// UI_MSWINDOWS_DISPLAY
// UI_OS2_DISPLAY
// UI_MOTIF_DISPLAY
#endif
```

UI_EVT.HPP

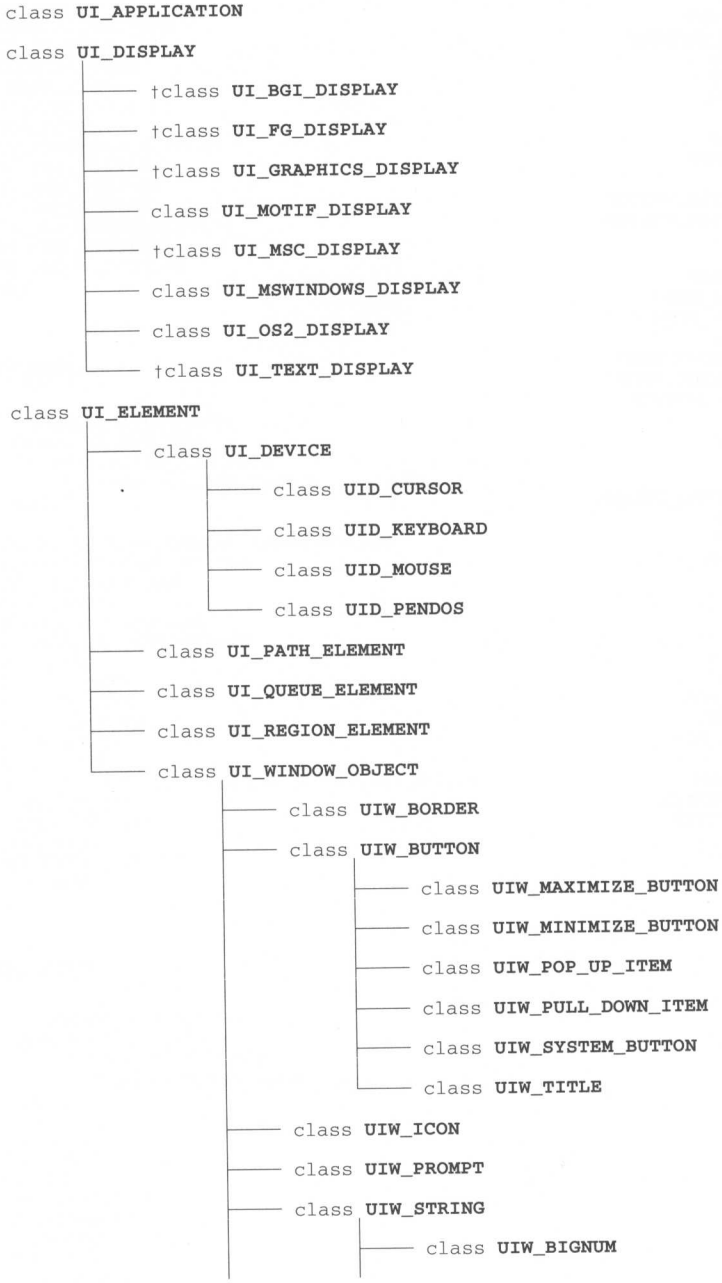
```
#if !defined(UI_EVT_HPP)
#   define UI_EVT_HPP
#   if !defined(UI_DSP_HPP)
#       include <ui_dsp.hpp>
#   endif

// Compiler/Environment Dependencies
// Special hotkey values
// Key Information
// shiftState
// Mouse Information
// UI_SCROLL_INFORMATION
// UI_EVENT
// DEVICE_TYPE
// SYSTEM_EVENT
// LOGICAL_EVENT
// USER_EVENT
// UI_DEVICE
// Private ALT key state enum for UI_DEVICE.
// UID_CURSOR
// UID_KEYBOARD
// UID_MOUSE
// UI_EVENT_MANAGER
// Q_FLAGS
```

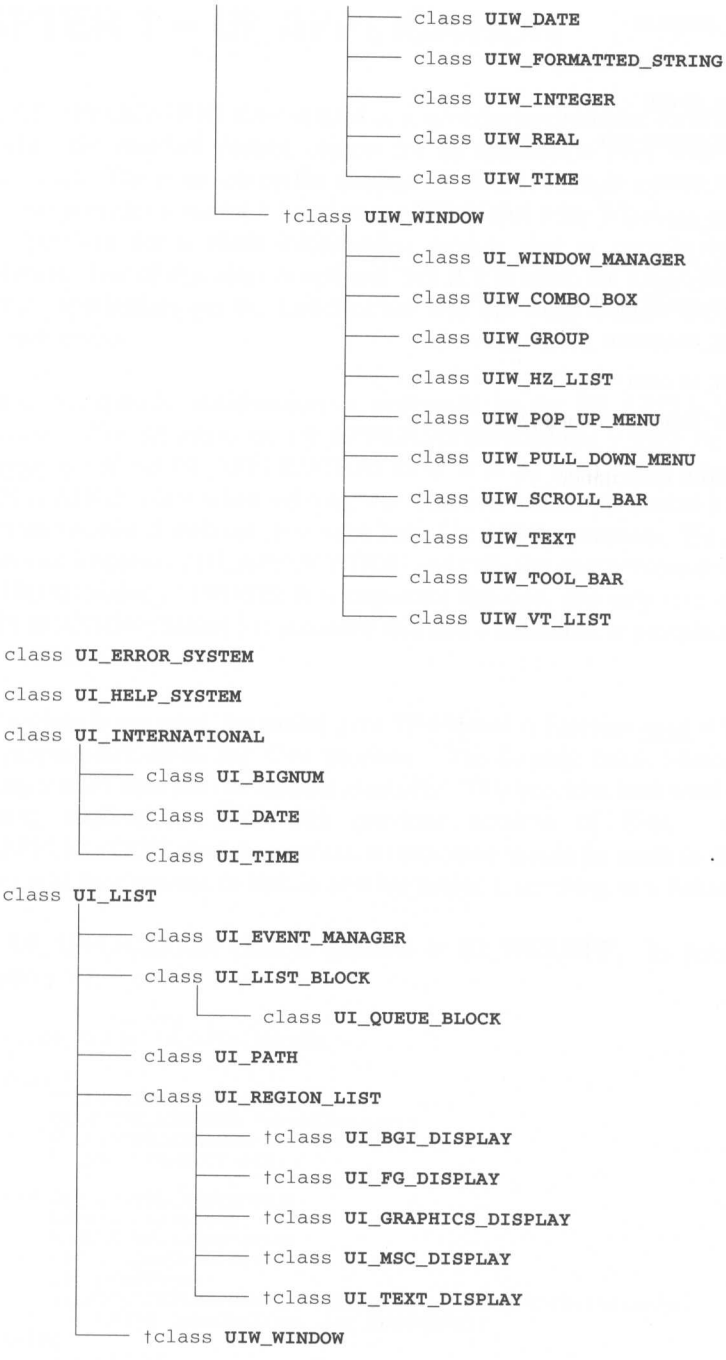
UI_WIN.HPP

```
#if !defined(UI_WIN_HPP)
#   define UI_WIN_HPP
#   if !defined(UI_EVT_HPP)
#       include <ui_evt.hpp>
#   endif

// numberID
// UI_APPLICATION
// UI_ITEM
// UI_PALETTE_MAP
// UI_EVENT_MAP
// UI_WINDOW_OBJECT
// WOF_FLAGS
// WOAF_FLAGS
```

```
// WOS_STATUS
// INFO_REQUEST
// UI_HELP_CONTEXT
// UIW_WINDOW
// WNF_FLAGS
// INFO_REQUEST
// UI_WINDOW_MANAGER
// UIW_BORDER
// UIW_PROMPT
// UIW_BUTTON
// BTF_FLAGS
// BTS_STATUS
// INFO_REQUEST
// UIW_TITLE
// UIW_MAXIMIZE_BUTTON
// UIW_MINIMIZE_BUTTON
// UIW_ICON
// ICF_FLAGS
// INFO_REQUEST
// UIW_POP_UP_MENU
// UIW_POP_UP_ITEM
// MNIF_FLAGS
// UIW_PULL_DOWN_MENU
// UIW_PULL_DOWN_ITEM
// UIW_SYSTEM_BUTTON
// SYF_FLAGS
// UIW_STRING
// STF_FLAGS
// UIW_DATE
// UIW_FORMATTED_STRING
// FMI_RESULT
// UIW_BIGNUM
// UIW_INTEGER
// UIW_REAL
// UIW_TIME
// UIW_TEXT
// UIW_GROUP
// UIW_VT_LIST
// UIW_HZ_LIST
// UIW_COMBO_BOX
// INFO_REQUEST
// UIW_SCROLL_BAR
// sbFlags
// UIW_TOOL_BAR
// UI_ERROR_SYSTEM
// UI_HELP_SYSTEM
```

CLASS HIERARCHY




class UIW_DATE
class UIW_FORMATTED_STRING
class UIW_INTEGER
class UIW_REAL
class UIW_TIME
†class UIW_WINDOW
class UI_WINDOW_MANAGER
class UIW_COMBO_BOX
class UIW_GROUP
class UIW_HZ_LIST
class UIW_POP_UP_MENU
class UIW_PULL_DOWN_MENU
class UIW_SCROLL_BAR
class UIW_TEXT
class UIW_TOOL_BAR
class UIW_VT_LIST
class UI_ERROR_SYSTEM
class UI_HELP_SYSTEM
class UI_INTERNATIONAL
class UI_BIGNUM
class UI_DATE
class UI_TIME
class UI_LIST
class UI_EVENT_MANAGER
class UI_LIST_BLOCK
class UI_QUEUE_BLOCK
class UI_PATH
class UI_REGION_LIST
†class UI_BGI_DISPLAY
†class UI_FG_DISPLAY
†class UI_GRAPHICS_DISPLAY
†class UI_MSC_DISPLAY
†class UI_TEXT_DISPLAY
†class UIW_WINDOW

```
class UI_STORAGE
class UI_STORAGE_OBJECT
struct UI_EVENT
struct UI_EVENT_MAP
struct UI_ITEM
struct UI_KEY
struct UI_PALETTE
struct UI_PALETTE_MAP
struct UI_POSITION
struct UI_REGION
struct UI_SCROLL_INFORMATION
```

† - indicates multiple inheritance

CHAPTER 1 – UI_APPLICATION

The UI_APPLICATION class is used as a generic, environment-independent template to initialize the standard control objects for an application built with Zinc Application Framework. The class sets up the display, the Event Manager and the Window Manager, and also provides a **main( )** function (or **WinMain( )** for Windows and Windows NT). This provides for a clean initialization module that is completely portable across platforms. Use of this class is optional, but if it is used, the programmer merely has to provide application-specific initialization and the main control loop to retrieve and dispatch events.

Application-specific initialization is performed in the **UI_APPLICATION::Main( )** function. The definition of **UI_APPLICATION::Main( )** must be provided by the programmer if the UI_APPLICATION class is to be used. The reference to the UI_APPLICATION class when defining this function causes the **main( )** (or **WinMain( )**) function associated with the class to be linked in with the program. This **main( )** function creates an instance of UI_APPLICATION and calls the programmer-defined **UI_APPLICATION::Main( )**. (**NOTE:** It is important that one, and only one, definition of **UI_APPLICATION::Main( )** is provided and that a definition is provided only if using the UI_APPLICATION class.)

If this class is not used, the **main( )** (or **WinMain( )**) function must still be provided by the programmer, as in any C++ program. The display, Event Manager and Window Manager must then also be created manually. This provides backward-compatibility for existing applications built with previous versions of Zinc. (**NOTE:** If the UI_APPLICATION class is not used, no reference should be made to the class, since the linker will then attempt to link in another **main( )**, resulting in a linker error.)

The UI_APPLICATION class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UI_APPLICATION
{
public:
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PATH *searchPath;

#if defined(ZIL_MSWINDOWS)
    HANDLE hInstance;
    HANDLE hPrevInstance;
    LPSTR lpszCmdLine;
    int nCmdShow;
    UI_APPLICATION(HANDLE hInstance, HANDLE hPrevInstance,
        LPSTR lpszCmdLine, int nCmdShow);
#else
    int argc;
```

```
    char **argv;
    UI_APPLICATION(int argc, char **argv);
#endif

    ~UI_APPLICATION(void);
    EVENT_TYPE Control(void);
    int Main(void);
};
```

- *display* is a pointer to the UI_DISPLAY that is created in the UI_APPLICATION constructor. The type of display created depends on which environment is being used. For DOS, the type of display created also depends on which graphics library is being used for the application. For example, if the DOS BGI library is used, *display* will be of type UI_BGI_DISPLAY (see "Appendix A—Compiler Considerations" in the *Programming Techniques* manual for more information on using graphics libraries in Zinc programs). *display* will be of type UI_MS-WINDOWS_DISPLAY if the application is a Windows application, UI_OS2_-DISPLAY for an OS/2 application or UI_MOTIF_DISPLAY for a Motif application.

- *eventManager* is a pointer to the Event Manager created in the UI_APPLICATION constructor.

- *windowManager* is a pointer to the Window Manager created in the UI_-APPLICATION constructor.

- *searchPath* is a pointer to a UI_PATH object containing the program startup directory as supplied by argv[0], if the application is for DOS, OS/2 or Motif. *searchPath* will also ensure that the current working directory is searched if access to files is required from within the program. If the application is a Windows application, *searchPath* does not maintain a pointer to the program startup directory but ensures that the current working directory is searched.

- *hInstance* is the instance handle of the application. This member is available only in Windows applications.

- *hPrevInstance* is the handle of another instance of the application, if one is running. This member is available only in Windows applications.

- *lpszCmdLine* is a pointer to the string entered at the command line. This member is available only in Windows applications.

- *nCmdShow* indicates how the application is to displayed upon execution. This member is available only in Windows applications.

- *argc* is a count of the number of command-line arguments that were entered when running the application. This member is available only in non-Windows applications.

- *argv* is a pointer to an array of the command-line arguments that were entered when running the application. This member is available only in non-Windows applications.

UI_APPLICATION::UI_APPLICATION

Syntax

#include <ui_win.hpp>

UI_APPLICATION(HANDLE *hInstance*, HANDLE *hPrevInstance*, LPSTR *lpszCmdLine*, int *nCmdShow*);
or
UI_APPLICATION(int *argc*, char ***argv*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded advanced constructors return a pointer to a new UI_APPLICATION class object. The constructor initializes the *display*, *eventManager*, *windowManager* and *searchPath* member variables. By default, *eventManager* will have a UID_KEYBOARD, a UID_MOUSE and a UID_CURSOR device attached to it.

These constructors should not be called by the programmer but are called from within the UI_APPLICATION class' **main( )** (or **WinMain( )**) function, which is linked in automatically when a reference to the UI_APPLICATION class is made.

The first constructor is specific to a Windows application. It takes the following arguments:

- $hInstance_{in}$ is the instance handle of the application. This parameter is automatically passed in to the application as a **WinMain( )** parameter.

- *hPrevInstance*$_{in}$ is the handle of another instance of the application, if one is running. This parameter is automatically passed in to the application as a **WinMain( )** parameter.

- *lpszCmdLine*$_{in}$ is a pointer to the string entered at the command line. This parameter is automatically passed in to the application as a **WinMain( )** parameter.

- *nCmdShow*$_{in}$ indicates if the application's initial window should display in a maximized state, a normal state or a minimized state upon execution of the application. This parameter is automatically passed in to the application as a **WinMain( )** parameter.

The second constructor is specific to non-Windows applications. It takes the following arguments:

- *argc*$_{in}$ is a count of the number of command-line arguments that were entered when running the application. This parameter is passed in to the application as a **main( )** parameter.

- *argv*$_{in}$ is a pointer to an array of the command-line arguments that were entered when running the application. This parameter is passed in to the application as a **main( )** parameter.

Example

```
#include <ui_win.hpp>

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

// This main() is part of the UI_APPLICATION class.
int main(int argc, char **argv)
{
    UI_APPLICATION *application = new UI_APPLICATION(argc, argv);

    // Call the application program.
    int ccode = application->Main();

    // Restore the system.
    delete application;

    return(ccode);
}
```

UI_APPLICATION::~UI_APPLICATION

Syntax

#include <ui_win.hpp>

~UI_APPLICATION(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This destructor destroys the *display*, *eventManager*, *windowManager* and *searchPath* members, if they exist. It is not recommended that the programmer modify these members—but if they are changed, it is important to set these members to NULL if they are deleted prior to the destructor being called.

UI_APPLICATION::Main

Syntax

#include <ui_win.hpp>

int Main(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine is used to set up program-specific initialization and the main control loop. It is defined by the programmer. The **main( )** (or **WinMain( )**) function provided with

the UI_APPLICATION class calls this function after first creating an instance of UI_APPLICATION. It is possible to modify the *display*, *eventManager*, *windowManager* and *searchPath* member variables from within the **Main( )** function, but it is not recommended. If these members are modified, it is important to consider the order in which the members are modified, as some of these members maintain pointers to the other members.

NOTE: The programmer must provide one, and only one, definition for this function if **main( )** or **WinMain( )** is not used. If no definition is provided, or if more than one definition is provided, a linker error will occur.

- *returnValue*$_{out}$ is the program exit code.

Example

```
#include <ui_win.hpp>

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

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

    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + &(*new UIW_SYSTEM_BUTTON
            + new UIW_POP_UP_ITEM("~Restore", MNIF_RESTORE)
            + new UIW_POP_UP_ITEM("~Move", MNIF_MOVE)
            + new UIW_POP_UP_ITEM("~Size", MNIF_SIZE))
        + new UIW_TITLE("Window 1");
    *windowManager + window;

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

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

    // DO NOT delete the display, eventManager, or windowManager.
    // They are deleted in the UI_APPLICATION destructor.

    // Return the exit code.
    return (0);
}
```

UI_APPLICATION::Control

Syntax

#include <ui_win.hpp>

EVENT_TYPE Control(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine acts as the main control loop used to get events from the Event Manager and to dispatch them to the Window Manager. Use of this function is optional. If this function is not used, the programmer must implement a loop that collects events from the Event Manager and passes them to the Window Manager.

- *returnValue*$_{out}$ is the event type that caused the event loop to exit (i.e., L_EXIT or S_NO_OBJECT).

Example

```
#include <ui_win.hpp>

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

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

    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + &(*new UIW_SYSTEM_BUTTON
            + new UIW_POP_UP_ITEM("~Restore", MNIF_RESTORE)
            + new UIW_POP_UP_ITEM("~Move", MNIF_MOVE)
            + new UIW_POP_UP_ITEM("~Size", MNIF_SIZE))
        + new UIW_TITLE("Window 1");
```

```
    *windowManager + window;

    // Use Control() to get and dispatch events.
    EVENT_TYPE ccode = Control();

    // DO NOT delete the display, eventManager, or windowManager.
    // They are deleted in the UI_APPLICATION destructor.

    // Return the exit code.
    return (0);
}
```

CHAPTER 2 – UI_BGI_DISPLAY

The UI_BGI_DISPLAY class implements a graphics display that uses the Borland BGI graphics library package to display information to the screen. Since the UI_BGI_DISPLAY class is derived from UI_DISPLAY, only details specific to the UI_BGI_DISPLAY class are given in this chapter. For descriptions and examples regarding virtual or inherited display members, see "Chapter 6—UI_DISPLAY."

The UI_BGI_DISPLAY class is declared in **UI_DSP.HPP**. Its public and protected members are:

```
class EXPORT UI_BGI_DISPLAY : public UI_DISPLAY, public UI_REGION_LIST
{
public:
    struct BGIFONT
    {
        int font;
        int charSize;
        int multX, divX;
        int multY, divY;
        int maxWidth, maxHeight;
    };
    typedef char BGIPATTERN[8];

    static UI_PATH *searchPath;
    static BGIFONT fontTable[MAX_LOGICAL_FONTS];
    static BGIPATTERN patternTable[MAX_LOGICAL_PATTERNS];

    UI_BGI_DISPLAY(int driver = 0, int mode = 0);
    virtual ~UI_BGI_DISPLAY(void);
    virtual void Bitmap(SCREENID screenID, int column, int line,
        int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette = NULL,
        const UI_REGION *clipRegion = NULL, HBITMAP *colorBitmap = NULL,
        HBITMAP *monoBitmap = NULL);
    virtual void BitmapArrayToHandle(SCREENID screenID, int bitmapWidth,
        int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette, HBITMAP *colorBitmap,
        HBITMAP *monoBitmap);
    virtual void BitmapHandleToArray(SCREENID screenID, HBITMAP colorBitmap,
        HBITMAP monoBitmap, int *bitmapWidth, int *bitmapHeight,
        UCHAR **bitmapArray);
    virtual void Ellipse(SCREENID screenID, int column, int line,
        int startAngle, int endAngle, int xRadius, int yRadius,
        const UI_PALETTE *palette, int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void IconArrayToHandle(SCREENID screenID, int iconWidth,
        int iconHeight, const UCHAR *iconArray, const UI_PALETTE *palette,
        HICON *icon);
    virtual void IconHandleToArray(SCREENID screenID, HICON icon,
        int *iconWidth, int *iconHeight, UCHAR **iconArray);
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground);
    virtual void Polygon(SCREENID screenID, int numPoints,
        const int *polygonPoints, const UI_PALETTE *palette,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
```

```
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion, SCREENID screenID = ID_SCREEN);
    virtual void RegionDefine(SCREENID screenID, int left, int top,
        int right, int bottom);
    virtual void RegionMove(const UI_REGION &oldRegion, int newColumn,
        int newLine, SCREENID oldScreenID = ID_SCREEN,
        SCREENID newScreenID = ID_SCREEN);
    virtual void Text(SCREENID screenID, int left, int top,
        const char *text, const UI_PALETTE *palette, int length = -1,
        int fill = TRUE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN, LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextWidth(const char *string, SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
        int bottom);
    virtual int VirtualPut(SCREENID screenID);

protected:
    int maxColors;
    char _virtualCount;
    UI_REGION _virtualRegion;
    char _stopDevice;

    void SetFont(LOGICAL_FONT logicalFont);
    void SetPattern(const UI_PALETTE *palette, int xor);
};
```

- *BGIFONT* is a structure that contains the following font information:

  font contains the value of the font. FNT_SMALL_FONT (font is 0), FNT_-DIALOG_FONT (font is 1) and FNT_SYSTEM_FONT (font is 2) are predefined by Zinc.

  charSize can be used to magnify the size of a character. For more information see **settextstyle( )** in the *Borland C++ Library Reference*.

  multX, *divX*, *multY* and *divY* provide additional ways to scale the font. For more information see **setusercharsize( )** in the *Borland C++ Library Reference*.

  maxHeight is the height of the tallest character.

  maxWidth is the width of the widest character.

- *BGIPATTERN* is an array of 8 bytes that make up the 8x8 bitmap pattern. Each byte (8 bits) corresponds to 8 pixels in the pattern. The patterns defined by Zinc are: PTN_SOLID_FILL, PTN_INTERLEAVE_FILL and PTN_BACKGROUND_FILL. For more information see **setfillpattern( )** in the *Borland C++ Library Reference*.

- *searchPath* contains the path to be searched for the .BGI drivers or .CHR files.

- *fontTable* is an array of *BGIFONT*. The default array contains space for 10 *BGIFONT* entries. The following entries are pre-defined by Zinc:

  FNT_SMALL_FONT—a font used to display an icon's text string.

  FNT_DIALOG_FONT—a font used when text is displayed on window objects (e.g., UIW_BUTTON, UIW_STRING, UIW_TEXT, etc.)

  FNT_SYSTEM_FONT—a sans-serif style font used to display a window's title.

  NOTE: When these three fonts are used, no .CHR files are needed since they are linked into Zinc Application Framework. However, if other "stroked" fonts are added to this table, the proper .CHR files must either be in the current path or be linked into the application.

- *patternTable* is an array of *BGIPATTERN*. The default array contains space for 15 *BGIPATTERN* entries. The following entries are pre-defined by Zinc:

  PTN_SOLID_FILL—Solid fill.

  PTN_INTERLEAVE_FILL—Interleaving line fill.

  PTN_BACKGROUND_FILL—Background fill style.

- *maxColors* tells the maximum number of colors supported by the display. For example, an EGA display supports sixteen colors.

- *_virtualCount* is a count of the number of virtual screen operations that have taken place. For example, when the **VirtualGet( )** function is called, *_virtualCount* is decremented. Additionally, when the **VirtualPut( )** function is called, *_virtualCount* is incremented.

- *_virtualRegion* is the region affected by either **VirtualGet( )** or **VirtualPut( )**.

- *_stopDevice* is a variable used to prevent recursive updates of device images on the display. If *_stopDevice* is TRUE, no drawing will be done to the screen. Otherwise, drawing will be made directly to the screen display.

UI_BGI_DISPLAY::UI_BGI_DISPLAY

Syntax

#include <ui_dsp.hpp>

UI_BGI_DISPLAY(int *driver* = 0, int *mode* = 0);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This constructor returns a pointer to a new UI_BGI_DISPLAY object. When a new UI_BGI_DISPLAY class is constructed, the system finds the associated BGI device driver and sets the screen display to the background color and pattern specified by the inherited variable *backgroundPalette*.

- $driver_{in}$ and $mode_{in}$ are arguments passed to the Borland **initgraph( )** function. The argument *driver* specifies the graphics driver to be used. (The value 0 indicates an auto-detection.) The argument *mode* specifies the initial graphics mode (used only if *driver* is not auto-detect). For more information on these arguments see **detectgraph( )** in the *Borland C++ Library Reference* manual.

Example 1

```
#include <ui_win.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_BGI_DISPLAY;

    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;

    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);

    .
    .
    .
```

```
    // Restore the system.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

Example 2

This example shows how a different font can be installed into the *fontTable* so that it may be used by the system.

```
#include <ui_win.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_BGI_DISPLAY;
    if (display->installed)
    {
        // Set up a BGIFONT structure with the Borland default font.
        UI_BGI_DISPLAY::BGIFONT BGIFont = { DEFAULT_FONT, 1, 1, 1, 1, 1, 8, 8 };
        UI_BGI_DISPLAY::fontTable[FNT_DIALOG_FONT] = BGIFont;

        // Update the cellWidth and the cellHeight according to the new font.
        display->preSpace = 4;      // The space between the top (or bottom) of
        display->postSpace = 4;     // the text and the border.
        display->cellWidth = display->TextWidth("M", ID_SCREEN,
            FNT_DIALOG_FONT);
        display->cellHeight = display->TextHeight("M", ID_SCREEN,
            FNT_DIALOG_FONT) + 2*display->preSpace + 2*display->postSpace;
    }

    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;

    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .

    // Restore the system.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

CHAPTER 3 – UI_BIGNUM

The UI_BIGNUM class is a lower-level class used to store and manipulate numerical values. It is not a window object. (See "Chapter 45—UIW_BIGNUM" of this manual for information about the bignum window object.) The values handled by UI_BIGNUM include both integer and real bignums with a default maximum of 30 digits to the left and 8 digits to the right of the decimal point.

The UI_BIGNUM class is declared in **UI_GEN.HPP**. Its public members are:

```
#define NUMBER_WHOLE 30
#define NUMBER_DECIMAL 8
#if defined(ZIL_BITS32)
#define DIGITS 8
#else
#define DIGITS 4
#endif

typedef long ibignum;
typedef double rbignum;
#if DIGITS == 4
typedef unsigned int nm_number;
#elif DIGITS == 8
typedef unsigned long nm_number;
#endif

class EXPORT UI_BIGNUM : public UI_INTERNATIONAL
{
    // Standard math function.
    friend UI_BIGNUM &abs(const UI_BIGNUM &number);
    friend UI_BIGNUM &ceil(const UI_BIGNUM &number);
    friend UI_BIGNUM &floor(const UI_BIGNUM &number);
    friend UI_BIGNUM &round(const UI_BIGNUM &number, int places = 0);
    friend UI_BIGNUM &truncate(const UI_BIGNUM &number, int places = 0);

public:
    // Constructor/Destructor.
    UI_BIGNUM(void);
    UI_BIGNUM(ibignum value);
    UI_BIGNUM(rbignum value);
    UI_BIGNUM(const char *string);
    UI_BIGNUM(const UI_BIGNUM &number);
    virtual ~UI_BIGNUM(void);
    void Export(ibignum *value);
    void Export(rbignum *value);
    void Export(char *string, NMF_FLAGS nmFlags);
    NMI_RESULT Import(ibignum value);
    NMI_RESULT Import(rbignum value);
    NMI_RESULT Import(const UI_BIGNUM &number);
    NMI_RESULT Import(const char *string, const char *decimalString = NULL,
        const char *signString = NULL);

    UI_BIGNUM &operator=(const UI_BIGNUM &number);
    UI_BIGNUM &operator+(const UI_BIGNUM &number);
    UI_BIGNUM &operator-(const UI_BIGNUM &number);
    UI_BIGNUM &operator*(const UI_BIGNUM &number);
    UI_BIGNUM &operator++(void);
    UI_BIGNUM &operator--(void);
    UI_BIGNUM &operator+=(const UI_BIGNUM &number);
    UI_BIGNUM &operator-=(const UI_BIGNUM &number);
```

```
    int operator==(const UI_BIGNUM &number);
    int operator!=(const UI_BIGNUM &number);
    int operator>(const UI_BIGNUM &number);
    int operator>=(const UI_BIGNUM &number);
    int operator<(const UI_BIGNUM &number);
    int operator<=(const UI_BIGNUM &number);
};
```

- *NUMBER_WHOLE* is the number of digits allowed to the left of the decimal place. The default value is 30. For numbers with greater than 30 digits to the left of the decimal place, simply change the default *#define* value to the desired amount and recompile the bignum module. No other changes are necessary.

- *NUMBER_DECIMAL* is the number of digits allowed to the right of the decimal place. The default value is 8. For numbers with greater precision than 8 decimal places, simply change the default *#define* value to the desired amount and recompile the bignum module. No other changes are necessary.

- *DIGITS* is used for number conversion and manipulation. The default value is 4, which allows the UI_BIGNUM class to work with integer values. If this value is changed to 8, the UI_BIGNUM class will work with long values.

NOTE: The UI_BIGNUM class uses special number types to do numerical operations. With the UI_BIGNUM class, use the following number types:

- *ibignum* is an integer data type associated with UI_BIGNUM. This type will be used when integer operations are done.

- *rbignum* is the real number data type associated with UI_BIGNUM. Using this type will require that the floating point library be used. Unless an individual application requires that floating point numbers be used, it is recommended that the string equivalent of a decimal number be used instead of the floating point numbers (i.e., "1.1" vs. 1.1).

UI_BIGNUM::UI_BIGNUM

Syntax

#include <ui_gen.hpp>

UI_BIGNUM(void);
or
UI_BIGNUM(ibignum *value*);
or

UI_BIGNUM(rbignum *value*);

or

UI_BIGNUM(const char **string*);

or

UI_BIGNUM(const UI_BIGNUM &*number*);

Portability

These functions are available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors return a pointer to a new UI_BIGNUM class object.

The first overloaded constructor creates a UI_BIGNUM object and initializes its value to zero.

The second overloaded constructor creates a UI_BIGNUM object and initializes its value with *value*.

- $value_{in}$ is an *ibignum* value to which the UI_BIGNUM object will be initialized.

The third overloaded constructor creates a UI_BIGNUM object and initializes its value with *value*.

- $value_{in}$ is an *rbignum* value to which the UI_BIGNUM object will be initialized.

The fourth overloaded constructor creates a UI_BIGNUM object and initializes its value with *string*.

- $string_{in}$ is a character string that contains the value to which the UI_BIGNUM object will be initialized.

The fifth overloaded constructor creates a UI_BIGNUM object and initializes its value with *number*.

- $number_{in}$ is another UI_BIGNUM object whose value will be copied into the UI_-BIGNUM object being constructed.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    ibignum i = 4;
    rbignum r = 7.1;

    UI_BIGNUM number1;
    UI_BIGNUM *number2 = new UI_BIGNUM(&number1);
    UI_BIGNUM *number3 = new UI_BIGNUM(i);
    UI_BIGNUM *number4 = new UI_BIGNUM(r);
    UI_BIGNUM *number5 = new UI_BIGNUM("100");
    .
    .
    .

    delete number5;
    delete number4;
    delete number3;
    delete number2;
}
```

UI_BIGNUM::~UI_BIGNUM

Syntax

#include <ui_gen.hpp>

virtual ~UI_BIGNUM(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_BIGNUM object. Care should be taken to only destroy a UI_BIGNUM class that is not attached to another associated object.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_BIGNUM *number = new UI_BIGNUM("100");
    .
    .
    .
    delete number;
}
```

UI_BIGNUM::abs

Syntax

friend UI_BIGNUM &abs(const UI_BIGNUM &*number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a pointer to the UI_BIGNUM object containing the absolute value of the UI_BIGNUM value.

- $returnValue_{out}$ is a UI_BIGNUM object containing the absolute value of *number*.
- $number_{in}$ is a UI_BIGNUM object of which the absolute value is taken.

Example

```
#include <ui_gen.hpp>

ExampleFunction( )
{
    UI_BIGNUM *firstValue;
    UI_BIGNUM secondValue("-100");
    UI_BIGNUM thirdValue("-100");
```

```
    .
    .
    .
    firstValue = abs(secondValue + thirdValue);
}
```

UI_BIGNUM::ceil

Syntax

friend UI_BIGNUM &ceil(const UI_BIGNUM &*number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a pointer to the UI_BIGNUM object containing the ceiling value of the UI_BIGNUM value. The ceiling of a bignum is considered to be the smallest integer that is greater than or equal to *number*.

- $returnValue_{out}$ is a UI_BIGNUM object containing the ceiling value of *number*.
- $number_{in}$ is a UI_BIGNUM object of which the ceiling value is taken.

Example

```
#include <ui_gen.hpp>

ExampleFunction( )
{
    UI_BIGNUM *firstValue;
    UI_BIGNUM secondValue("100.6");
    .
    .
    .
    firstValue = ceil(secondValue);
}
```

UI_BIGNUM::Export

Syntax

```
#include <ui_gen.hpp>

void Export(ibignum *value);
    or
void Export(rbignum *value);
    or
void Export(char *string, NMF_FLAGS nmFlags);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded functions return the numerical information through a function-specific numeric value.

The first two overloaded functions copy the bignum information into the *value* argument.

- $value_{in/out}$ is a numeric value. The following values are supported:

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

  rbignum—A double precision floating point number.

The last overloaded function copies number information into the *string* argument. When this function is used, it is very important that space for *string* be previously allocated by the programmer.

- $string_{in/out}$ is a pointer to a string that represents a bignum in ASCII form.

- $nmFlags_{in}$ gives formatting information about the return ASCII bignum. The following flags (declared in **UI_GEN.HPP**) are used to format the ASCII bignum string:

NMF_COMMAS—Formats the bignum with commas.

NMF_CREDIT—Formats the bignum with the '(' and ')' credit symbols whenever the bignum is negative.

NMF_CURRENCY—Formats the bignum string with the country-specific currency symbol.

NMF_DECIMAL_FLAGS—May be used to clear the decimal flags (e.g., nmFlags &= ~NMF_DECIMAL_FLAGS). This flag may also be used to isolate the number of decimal places in a bignum.

```
ExampleFunction( )
{
    int digits = ((nmFlags & NMF_DECIMAL_FLAGS) >> 12) - 1;
    .
    .
    .

}
```

NOTE: NMF_DECIMAL_FLAGS is an advanced flag and should not be used in conjunction with any other NMF_DECIMAL(decimal) flags.

NMF_DECIMAL(decimal)—Formats the bignum string with *decimal* number of decimal places to the right of the decimal point. Decimal places from 0 to 8 are supported.

NMF_DIGITS(digits)—Formats the bignum string with *digits* number of digits to the left of the decimal point. Digits from 0 to 30 are supported.

NMF_NO_FLAGS—Does not associate any special flags with the UI_BIGNUM class object. This is the default argument if no other argument is provided. This flag should not be used in conjunction with any other NMF flags.

NMF_PERCENT—Formats the bignum with the percent symbol.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    .
    .
    .
```

```
    char string2[40];
    number.Export(string2, NMF_NO_FLAGS);
}
```

UI_BIGNUM::floor

Syntax

friend UI_BIGNUM &floor(const UI_BIGNUM &*number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a pointer to the UI_BIGNUM object containing the floor value of the UI_BIGNUM value. The floor of a bignum is considered to be the largest integer value that is not greater than *number*.

- $returnValue_{out}$ is a UI_BIGNUM object containing the floor value of *number*.
- $number_{in}$ is a UI_BIGNUM object of which the floor value is taken.

Example

```
#include <ui_gen.hpp>

ExampleFunction( )
{
    UI_BIGNUM *firstValue;
    UI_BIGNUM secondValue("100.6");
    .
    .
    .

    firstValue = floor(secondValue);
}
```

UI_BIGNUM::Import

Syntax

```
#include <ui_gen.hpp>

NMI_RESULT Import(ibignum value);
    or
NMI_RESULT Import(rbignum value);
    or
NMI_RESULT Import(const UI_BIGNUM &number);
    or
NMI_RESULT Import(const char *string, const char *decimalString = NULL,
    const char *signString = NULL);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded functions set the numerical information with a function-specific numeric value.

The first two overloaded functions copy the bignum information from the *value* argument.

- $returnValue_{out}$ is 0 (NMI_OK) if the conversion was successful. Otherwise, *returnValue* is NMI_OUT_OF_RANGE, and the bignum object will not be modified.

- $value_{in}$ is a numeric value. The following values are supported:

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

  rbignum—A double precision floating point bignum.

  UI_BIGNUM—The value of another UI_BIGNUM object may be copied into the UI_BIGNUM object.

The third overloaded function copies the bignum information from the *number* argument.

- *returnValue*$_{out}$ is 0 (NMI_OK) if the conversion was successful. Otherwise, *returnValue* is NMI_OUT_OF_RANGE, and the bignum object will not be modified.
- *number*$_{in}$ is a UI_BIGNUM reference variable. The value in *number* is copied into the bignum object.

The last overloaded function sets the UI_BIGNUM information according to the string argument.

- *returnValue*$_{out}$ is 0 (NMI_OK) if the conversion was successful. Otherwise, *returnValue* is NMI_OUT_OF_RANGE, and the number object will not be modified.
- *string*$_{in}$ is a pointer to a string that represents a bignum in ASCII form.
- *decimalString*$_{in}$ is a pointer to the decimal character to be used in formatting the decimal number.
- *signString*$_{in}$ is a pointer to the sign character to be used in formatting the bignum.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    .
    .
    .

    char *string = "100";
    UI_BIGNUM number;
    number.Import(string);
}
```

UI_BIGNUM::round

Syntax

friend UI_BIGNUM &round(const UI_BIGNUM &*number*, int *places* = 0);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a pointer to a UI_BIGNUM object containing the rounded value of *number*.

- $returnValue_{out}$ is a UI_BIGNUM object containing the value of *number* rounded to the *places* decimal place.
- $number_{in}$ is the UI_BIGNUM value to be rounded.
- $places_{in}$ determines how many decimal places to round *number*. For example, if *places* were 1, the value 100.163 would be rounded to 100.2. If *places* were -1, the value 123.789 would be rounded to 120. The default value, 0, causes *number* to be rounded to the decimal point.

Example

```
void UIW_INTL_CURRENCY::SetCountryCode(int _countryTableEntry)
{
    // Do Currency translation.
    UI_BIGNUM *amount = DataGet();
    rbignum value;
    amount->Export(&value);

    value *= _currency[countryTableEntry][_countryTableEntry];
    amount->Import(value);
    *amount = round(*amount, 2);

    countryTableEntry = _countryTableEntry;
    DataSet(amount);
}
```

UI_BIGNUM::truncate

Syntax

friend UI_BIGNUM &truncate(const UI_BIGNUM &*number*, int *places* = 0);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a pointer to the UI_BIGNUM object containing the truncated value of the UI_BIGNUM value.

- *returnValue*$_{out}$ is a UI_BIGNUM object containing the value of *number* after being truncated to *places* decimal places.
- *number*$_{in}$ is the UI_BIGNUM value to be truncated.
- *places*$_{in}$ determines to which digit to truncate *number*. For example, if *places* were 1, the value 100.163 would be truncated to 100.1. If *places* were -1, the value 123.789 would be truncated to 120. The default value, 0, causes *number* to be truncated to the decimal point.

Example

```
#include <ui_gen.hpp>

ExampleFunction( )
{
    UI_BIGNUM *firstValue;
    UI_BIGNUM secondValue("100.6");
    .
    .
    .

    firstValue = truncate(secondValue);
}
```

UI_BIGNUM::operator =

Syntax

#include <ui_gen.hpp>

UI_BIGNUM &operator = (const UI_BIGNUM &*number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The operator overload assigns the value of another UI_BIGNUM object specified by *number* to the UI_BIGNUM object.

- *returnValue*$_{out}$ is the resulting number object used to transfer the value of the bignum to another UI_BIGNUM object.
- *number*$_{in}$ is another UI_BIGNUM object containing the value to be assigned to the UI_BIGNUM object.

Example

```
#include <ui_gen.hpp>

ExampleFunction( )
{
    UI_BIGNUM firstValue;
    UI_BIGNUM secondValue;

    firstValue.Import("100");
    secondValue = firstValue;
}
```

UI_BIGNUM::operator +

Syntax

#include <ui_gen.hpp>

UI_BIGNUM &operator + (const UI_BIGNUM &*number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The operator overload adds the value of another UI_BIGNUM object specified by *number* to the UI_BIGNUM object.

- *returnValue*$_{out}$ is the resulting bignum object used to transfer the value of the bignum to another UI_BIGNUM object.

- *number*$_{in}$ is another UI_BIGNUM object containing the value to be added to the UI_BIGNUM object.

Example

```
#include <ui_gen.hpp>

ExampleFunction( )
{
    UI_BIGNUM firstValue;
    UI_BIGNUM secondValue("200");

    firstValue.Import("100");
    secondValue = secondValue + firstValue;
}
```

UI_BIGNUM::operator -

Syntax

#include <ui_gen.hpp>

UI_BIGNUM &operator - (const UI_BIGNUM &*number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The operator overload subtracts the value of another UI_BIGNUM object specified by *number* from the UI_BIGNUM object.

- *returnValue*$_{out}$ is the resulting bignum object used to transfer the value of the bignum to another UI_BIGNUM object.
- *number*$_{in}$ is another UI_BIGNUM object containing the value to be subtracted from the UI_BIGNUM object.

Example

```
#include <ui_gen.hpp>

ExampleFunction( )
{
    UI_BIGNUM firstValue;
    UI_BIGNUM secondValue("200");

    firstValue.Import("100");
    secondValue = secondValue - firstValue;
}
```

UI_BIGNUM::operator *

Syntax

#include <ui_gen.hpp>

UI_BIGNUM &operator * (const UI_BIGNUM *&number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The operator overload multiplies the value of another UI_BIGNUM object specified by *number* by the UI_BIGNUM object.

- *returnValue*$_{out}$ is the resulting bignum object used to transfer the value of the bignum to another UI_BIGNUM object.
- *number*$_{in}$ is another UI_BIGNUM object containing the value to be multiplied by the UI_BIGNUM object.

Example

```
#include <ui_gen.hpp>

ExampleFunction( )
{
    UI_BIGNUM firstValue;
    UI_BIGNUM secondValue("200");

    firstValue.Import("100");
    secondValue = secondValue * firstValue;
}
```

UI_BIGNUM::operator ++

Syntax

#include <ui_gen.hpp>

UI_BIGNUM &operator ++ (void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator overload increments the UI_BIGNUM object's value by one.

- *returnValue*$_{out}$ is the incremented UI_BIGNUM object used to update the current UI_BIGNUM object.

Example

```
#include <ui_gen.hpp>

ExampleFunction(UI_BIGNUM &number)
{
    .
    .
    .

    number++;
}
```

UI_BIGNUM::operator --

Syntax

#include <ui_gen.hpp>

UI_BIGNUM &operator -- (void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator overload decrements the UI_BIGNUM object's value by one.

- *returnValue*$_{out}$ is the decremented UI_BIGNUM object used to update the current UI_BIGNUM object.

Example

```
#include <ui_gen.hpp>

ExampleFunction(UI_BIGNUM &number)
{
    .
    .
    .
    number--;
}
```

UI_BIGNUM::operator +=

Syntax

#include <ui_gen.hpp>

UI_BIGNUM &operator += (const UI_BIGNUM *&number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The operator overload adds the value of another UI_BIGNUM object specified by *number* to the UI_BIGNUM object and copies the result back into the UI_BIGNUM object.

- *returnValue*$_{out}$ is the resulting bignum object used to transfer the value of the bignum to another UI_BIGNUM object.
- *number*$_{in}$ is another UI_BIGNUM object containing the value to be added to the UI_BIGNUM object.

Example

```
#include <ui_gen.hpp>

ExampleFunction( )
{
    UI_BIGNUM firstValue;
    UI_BIGNUM secondValue;

    firstValue.Import("100");
    secondValue.Import("200");
    secondValue += firstValue;
}
```

UI_BIGNUM::operator -=

Syntax

#include <ui_gen.hpp>

UI_BIGNUM &operator -= (const UI_BIGNUM &*number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The operator overload subtracts the value of another UI_BIGNUM object specified by *number* from the UI_BIGNUM object and copies the result back into the UI_BIGNUM object.

- $returnValue_{out}$ is the resulting bignum object used to transfer the value of the bignum to another UI_BIGNUM object.

- $number_{in}$ is another UI_BIGNUM object containing the value to be subtracted from the UI_BIGNUM object.

Example

```
#include <ui_gen.hpp>

ExampleFunction( )
{
    UI_BIGNUM firstValue;
    UI_BIGNUM secondValue;

    firstValue.Import("100");
    secondValue.Import("200");
    secondValue -= firstValue;
}
```

UI_BIGNUM::operator ==

Syntax

#include <ui_gen.hpp>

int operator == (const UI_BIGNUM &*number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The operator overload determines if the UI_BIGNUM object is equal to the UI_BIGNUM object specified by *number*.

- *returnValue*$_{out}$ is TRUE if the UI_BIGNUM object is equal to *number*. Otherwise, *returnValue* is FALSE.

- *number*$_{in}$ is the other UI_BIGNUM object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_BIGNUM governmentRevenue("10389230299.49");
    UI_BIGNUM governmentSpending("378321783443199.81");

    if (governmentRevenue == governmentSpending)
        printf("Budget is balanced?\n");
    else if (governmentRevenue < governmentSpending)
        printf("Big deal, this is normal.\n");
    else
        printf("Must be a computer error!\n");
}
```

UI_BIGNUM::operator !=

Syntax

#include <ui_gen.hpp>

int operator != (const UI_BIGNUM *&number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The operator overload determines if the UI_BIGNUM object is not equal to the UI_BIGNUM object specified by *number*.

- *returnValue*$_{out}$ is TRUE if the UI_BIGNUM object is not equal to *number*. Otherwise, *returnValue* is FALSE.

- *number*$_{in}$ is the other UI_BIGNUM object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_BIGNUM totalDays("400");
    UI_BIGNUM daysPerYear("365");

    if (totalDays != daysPerYear)
    {
        if (totalDays < daysPerYear)
            printf("Less than one year has passed.\n");
        else
            printf("More than one year has passed.\n");
    }
}
```

UI_BIGNUM::operator >

Syntax

#include <ui_gen.hpp>

int operator > (const UI_BIGNUM &*number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The operator overload determines whether the UI_BIGNUM object is greater than the UI_BIGNUM object specified by *number*.

- $returnValue_{out}$ is TRUE if the UI_BIGNUM object is greater than *number*. Otherwise, *returnValue* is FALSE.

- $number_{in}$ is the other UI_BIGNUM object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_BIGNUM governmentRevenue("10389230299.49");
    UI_BIGNUM governmentSpending("378321783443199.81");

    if (governmentRevenue == governmentSpending)
        printf("Budget is balanced?\n");
    else if (governmentRevenue > governmentSpending)
        // A good optimizing compiler would eliminate this option
        // as it would never occur.
        printf("Must be a computer error!\n");
    else
        printf("Big deal, this is normal.\n");
}
```

UI_BIGNUM::operator >=

Syntax

#include <ui_gen.hpp>

int operator >= (const UI_BIGNUM *&number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator overload determines whether the UI_BIGNUM object is greater than or equal to the UI_BIGNUM object specified by *number*.

- *returnValue*$_{out}$ is TRUE if the UI_BIGNUM object is greater than or equal to *number*. Otherwise, *returnValue* is FALSE.

- *number*$_{in}$ is the other UI_BIGNUM object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_BIGNUM totalDays("400");
    UI_BIGNUM daysPerYear("365");

    if (totalDays >= daysPerYear)
        printf("One year has passed.\n");
    else
        printf("Less than one year has passed.\n");
}
```

UI_BIGNUM::operator <

Syntax

#include <ui_gen.hpp>

int operator < (const UI_BIGNUM &*number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The operator overload determines whether the UI_BIGNUM object is less than the UI_BIGNUM object specified by *number*.

- $returnValue_{out}$ is TRUE if the UI_BIGNUM object is less than *number*. Otherwise, *returnValue* is FALSE.

- $number_{in}$ is the other UI_BIGNUM object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_BIGNUM governmentRevenue("10389230299.49");
    UI_BIGNUM governmentSpending("378321783443199.81");
```

```
    if (governmentRevenue == governmentSpending)
        printf("Budget is balanced?\n");
    else if (governmentRevenue < governmentSpending)
        printf("What else is new?!.\n");
    else
        printf("Must be a computer error!\n");
}
```

UI_BIGNUM::operator <=

Syntax

#include <ui_gen.hpp>

int operator <= (const UI_BIGNUM &*number*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The operator overload determines whether the UI_BIGNUM object is less than or equal to the UI_BIGNUM object specified by *number*.

- $returnValue_{out}$ is TRUE if the UI_BIGNUM object is less than or equal to *number*. Otherwise, *returnValue* is FALSE.

- $number_{in}$ is the other UI_BIGNUM object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_BIGNUM totalDays("400");
    UI_BIGNUM daysPerYear("365");

    if (totalDays <= daysPerYear)
        printf("Less than one year has passed.\n");
    else
        printf("One year has passed.\n");
}
```

CHAPTER 4 – UI_DATE

The UI_DATE class is a lower-level class used to store year, month, day and day-of-week date information. It is not a window object. (See "Chapter 49—UIW_DATE" of this manual for information about the date window object.)

NOTE: The **DayOfWeek**, **DaysInMonth** and **DaysInYear** functions may return questionable values for dates before 1753 due to the switch from the Julian calendar to the Gregorian calendar.

The UI_DATE class is declared in **UI_GEN.HPP**. Its public and protected members are:

```
class EXPORT UI_DATE : public UI_INTERNATIONAL
{
public:
    // Members described in UI_DATE reference chapter.
    static char **monthTable;
    static char **dayTable;

    UI_DATE(void);
    UI_DATE(const UI_DATE &date);
    UI_DATE(int year, int month, int day);
    UI_DATE(const char *string, DTF_FLAGS dtFlags = DTF_NO_FLAGS);
    UI_DATE(int packedDate);
    virtual ~UI_DATE(void);
    int DayOfWeek(void);
    int DaysInMonth(void);
    int DaysInYear(void);
    void Export(int *year, int *month, int *day, int *dayOfWeek = NULL);
    void Export(char *string, DTF_FLAGS dtFlags);
    void Export(int *packedDate);
    DTI_RESULT Import(void);
    DTI_RESULT Import(const UI_DATE &date);
    DTI_RESULT Import(int year, int month, int day);
    DTI_RESULT Import(const char *string, DTF_FLAGS dtFlags);
    DTI_RESULT Import(int packedDate);
    long operator=(long days);
    long operator=(const UI_DATE &date);
    long operator+(long days);
    long operator-(long days);
    long operator-(const UI_DATE &date);
    int operator>(const UI_DATE &date);
    int operator>=(const UI_DATE &date);
    int operator<(const UI_DATE &date);
    int operator<=(const UI_DATE &date);
    long operator++(void);
    long operator--(void);
    void operator+=(long days);
    void operator-=(long days);
    int operator==(const UI_DATE& date);
    int operator!=(const UI_DATE& date);
};
```

- *monthTable* is a pointer to the current month table used to give the ASCII representation of a date.

- *dayTable* is a pointer to the current day table used to give the ASCII representation of a date.

UI_DATE::UI_DATE

Syntax

```
#include <ui_gen.hpp>

UI_DATE(void);
    or
UI_DATE(const UI_DATE &date);
    or
UI_DATE(int year, int month, int day);
    or
UI_DATE(const char *string, DTF_FLAGS dtFlags = DTF_NO_FLAGS);
    or
UI_DATE(int packedDate);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors return a pointer to a new UI_DATE object.

The first overloaded constructor takes no arguments. It sets the date information according to the system's date.

The second overloaded constructor is a copy constructor that takes a previously constructed UI_DATE object to specify the default date.

- $date_{in}$ is a reference pointer to a previously constructed UI_DATE object.

The third overloaded constructor uses integer arguments to specify the default date.

- *year*$_{in}$ is the year. This argument must be either 0, if no year value is to be used with the date, or a value in a range from 100 to 32,767.

- *month*$_{in}$ is the month. This argument must be either 0, if no month value is to be used with the date, or a value in a range from 1 (January) to 12 (December).

- *day*$_{in}$ is the day. This argument must be either 0, if no day value is to be used with the date, or a value in a range from 1 to 31 that should be valid for the specified month and year.

The fourth overloaded constructor uses an ASCII string argument to specify the default date. The following algorithm is used to determine the proper order and meaning of date values:

1—Any number greater than 31 is assumed to be the year.

2—If the number is less than 100, 1900 is added to the value. Year values below 100 are not allowed in the UI_DATE class.

3—Any number between 13 and 31 is assumed to be the day. In ambiguous situations where both the day and month values are less than 13, the country code date format (e.g., DTF_US_FORMAT, DTF_JAPANESE_FORMAT) is used to decide the order of date values.

- *string*$_{in}$ is an ASCII string that contains the date information.

- *dtFlags*$_{in}$ gives information on how to interpret the date string. The following flags (declared in **UI_GEN.HPP**) override the country dependent information (supplied by the operating system):

  DTF_EUROPEAN_FORMAT—Forces the date to be interpreted in the European format (i.e., *day/month/year*), regardless of the default country information.

  DTF_JAPANESE_FORMAT—Forces the date to be interpreted in the Japanese format (i.e., *year/month/day*), regardless of the default country information.

  DTF_MILITARY_FORMAT—Forces the date to be interpreted in the U.S. Military format (i.e., *day/month/year* where *month* is a 3 letter abbreviated word), regardless of the default country information.

  DTF_NO_FLAGS—Does not associate any special flags with the UI_DATE object. In this case, the ASCII date will be interpreted using the default country

information. This is the default argument if no other argument is provided. This flag should not be used in conjunction with any other DTF flags.

DTF_SYSTEM—Sets the date value according to the system date if the strin g is blank or NULL. For example, if the DTF_SYSTEM flag were set and a NULL string value was specified, the date would be set to the system date.

DTF_US_FORMAT—Forces the date to be interpreted in the U.S. format (i.e., *month/day/year*), regardless of the default country information.

The fifth overloaded constructor uses a packed integer argument to specify the default date.

- *packedDate*$_{in}$ is a packed representation of the date (whose format is the same as the MS-DOS file dates). This argument is packed according to the following bit pattern:

  bits 0-4 specify the day,
  bits 5-8 specify the month, and
  bits 9-15 specify the year minus 1980 (e.g., a value of 5 means 1985).

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_DATE date1;
    UI_DATE date2(1990, 1, 1);
    UI_DATE *date3 = new UI_DATE("Jan. 1, 1990");
    UI_DATE *date4 = new UI_DATE(date1);
    .
    .
    .

    delete date4;
    delete date3;
    // The destructors for date1 and date2 are automatically called
    // when the scope of this function ends.
}
```

UI_DATE::~UI_DATE

Syntax

#include <ui_gen.hpp>

virtual ~UI_DATE(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_DATE object. Care should be taken to only destroy a UI_DATE class that is not attached to another object.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_DATE date1;
    UI_DATE *date4 = new UI_DATE(date1);
    UI_DATE date2(1990, 1, 1);
    UI_DATE *date3 = new UI_DATE("Jan. 1, 1990");
    .
    .
    .

    delete date3;
    delete date4;
    // The destructors for date1 and date2 are automatically called
    // when the scope of this function ends.
}
```

UI_DATE::DayOfWeek

Syntax

#include <ui_gen.hpp>

int DayOfWeek(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns the day of week (Sunday = 1, Monday = 2, . . . Saturday = 7) according to the specified UI_DATE object.

NOTE: DayOfWeek( ) may return questionable values for dates before 1753 due to the switch from the Julian calendar to the Gregorian calendar.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_DATE date;

    date.dayTable =
    {
        "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
        "Friday", "Saturday"
    };

    // Print the current day of week.
    printf("Today is %s.\n", date.dayTable[date.DayOfWeek() - 1]);
}
```

UI_DATE::DaysInMonth

Syntax

#include <ui_gen.hpp>

int DaysInMonth(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns the number of days in the month according to the specified UI_-DATE object. For example, if the date were December 15, 1993, **DaysInMonth** would return 31.

NOTE: DaysInMonth( ) may return questionable values for dates before 1753 due to the switch from the Julian calendar to the Gregorian calendar.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    // Print the number of days in the current month.
    UI_DATE date;
    printf("This month has %d days.\n", date.DaysInMonth());
}
```

UI_DATE::DaysInYear

Syntax

#include <ui_gen.hpp>

int DaysInYear(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns the number of days in the year according to the specified UI_DATE object. For example, if the date were January 15, 1992, **DaysInYear( )** would return 366 (i.e., 1 extra day for leap year).

NOTE: DaysInYear( ) may return questionable values for dates before 1753 due to the switch from the Julian calendar to the Gregorian calendar.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    // Print the number of days in the year.
    UI_DATE date;
    printf("This year has %d days.\n", date.DaysInYear());
}
```

UI_DATE::Export

Syntax

#include <ui_gen.hpp>

void Export(int **year*, int **month*, int **day*, int **dayOfWeek* = NULL);
or
void Export(char **string*, DTF_FLAGS *dtFlags*);
or
void Export(int **packedDate*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first overloaded function returns date information through four integer arguments.

- *year*$_{in/out}$ is a pointer to the year. If this argument is NULL, no year information is returned. If there is no year associated with the date, this argument will be 0. Otherwise, this argument will be a value within the range 100 to 32,767.

- *month*$_{in/out}$ is a pointer to the month. If this argument is NULL, no month information is returned. If there is no month associated with the date, this argument will be 0. Otherwise, this argument will be a value within the range 1 (January) to 12 (December).

- *day*$_{in/out}$ is a pointer to the day. If this argument is NULL, no day information is returned. If there is no day associated with the date, this argument will be 0. Otherwise, this argument will be a value within the range 1 to 31.

- *dayOfWeek*$_{in/out}$ is a pointer to the day-of-week. If this argument is NULL, no day-of-week information is returned. If the year, month and day values are all present, this argument will be a value within the range 1 (Sunday) to 7 (Saturday). Otherwise, this argument will be 0.

The second overloaded function returns the date information through the *string* argument.

- *string*$_{in/out}$ is a pointer to a string that gets the ASCII formatted date. This string must be long enough to contain the date. (A value of 64 is recommended.)

- *dtFlags*$_{in}$ gives formatting information about the return ASCII date. The following flags (declared in **UI_GEN.HPP**) override the country dependent information (supplied by the operating system):

DTF_ALPHA_MONTH—Formats the month as an ASCII string value.	`March 28, 1990`
	`December 4, 1980`
	`January 3, 2003`

DTF_DASH—Separates each date variable with a dash, regardless of the default country date separator.	`3-28-1990`
	`12-04-1980`
	`1-3-2003`

DTF_DAY_OF_WEEK—Adds an ASCII string day-of-week value to the date.	`Monday May 4, 1992`
	`Friday Dec. 5, 1980`
	`Sunday Jan. 4, 2003`

DTF_EUROPEAN_FORMAT—Forces the date to be formatted in the European format (i.e.,	`28/3/1990`
	`4 December, 1980`
	`3 Jan., 2003`

day/month/year), regardless of the default country information.

DTF_JAPANESE_FORMAT—Forces the date to be formatted in the Japanese format (i.e., *year/month/day*), regardless of the default country information.

```
1990/3/28
1980 December 4
2003 Jan. 3
```

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

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

DTF_NO_FLAGS—Does not associate any special flags with the format function. In this case, the ASCII date will be formatted using the default country information. This is the default argument if no other argument is provided. This flag should not be used in conjunction with any other DTF flags.

```
(European format)
4 December 1989
23 June 2000

(Japanese format)
1989 December 4
2000 June 23
```

DTF_SHORT_DAY—Adds an abbreviated day-of-week to the date.

```
Wed. March 28, 1990
Thurs. Dec. 4, 1980
Sat. January 3, 2003
```

DTF_SHORT_MONTH—Adds an abbreviated alphanumeric month to the date.

```
Mar. 28, 1990
Dec. 4, 1980
Jan. 3, 2003
```

DTF_SHORT_YEAR—Forces the year to be formatted as a two-digit value.

```
3/28/90
December 4, 80
Jan. 3, 89
```

DTF_SLASH—Separates each date value with a slash, regardless of the default country date separator.	3/28/90 12/04/1900 1/3/2003
DTF_SYSTEM—Uses the system date.	3/28/90 12/04/1980 1/3/2003
DTF_UPPER_CASE—Converts the alpha-numeric date to upper-case.	MARCH 28, 1990 DEC. 4, 1980 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.	March 28, 1990 12/4/1980 Jan 3, 2003
DTF_ZERO_FILL—Forces the year, month and day values to be zero filled when their values are less than 10.	March 08, 1990 12/04/1980 01/03/2003

The third overloaded function returns date information through a packed integer.

- *packedDate*$_{in}$ is a packed representation of the date (whose format is the same as the MS-DOS file dates). This argument is packed according to the following bit pattern:

  bits 0-4 specify the day,
  bits 5-8 specify the month, and
  bits 9-15 specify the year minus 1980 (e.g., a value of 5 means 1985).

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_DATE date; // Initialize a system date.

    // Print out the date in various forms.
    int year, month, day;
    date.Export(&year, &month, &day);
    printf("Integer date value: year-%d, month-%d, day-%d\n",
        year, month, day);
    char asciiDate[128];
    date.Export(asciiDate, DTF_NO_FLAGS);
    printf("ASCII date value: %s", asciiDate);

    // The destructor for date is automatically called when the
    // scope of this function ends.
}
```

UI_DATE::Import

Syntax

```
#include <ui_gen.hpp>

DTI_RESULT Import(void);
    or
DTI_RESULT Import(const UI_DATE &date);
    or
DTI_RESULT Import(int year, int month, int day);
    or
DTI_RESULT Import(const char *string, DTF_FLAGS dtFlags);
    or
DTI_RESULT Import(int packedDate);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first overloaded function sets the date information according to the system date.

- *returnValue*$_{out}$ returns one of the DTI_RESULT values listed below:

  DTI_OK—The date value was successfully imported.

  DTI_INVALID—An invalid date format was encountered (e.g., 28 Jan, 1992).

  DTI_AMBIGUOUS—The month name was ambiguous (e.g., "01-JU-92").

  DTI_INVALID_NAME—Either the month name or the day-of-week name was invalid (e.g., "Tuesday **Jaan** 28, 1992" or "**Tyesday** Jan 28, 1992").

  DTI_VALUE_MISSING—The required date value was missing (e.g., "5, 1991").

DTI_OUT_OF_RANGE—The date value was out of range (e.g., "Jan 33, 1992").

The second overloaded function copies the date information from the *date* reference argument.

- *returnValue*$_{out}$ returns one of the DTI_RESULT values listed with the first overloaded function.
- *date*$_{in}$ is a reference pointer to a previously constructed date.

The third overloaded function sets the date information according to specified integer arguments.

- *returnValue*$_{out}$ returns one of the DTI_RESULT values listed with the first overloaded function.
- *year*$_{in}$ is the year. This argument must be 0 if no year value is to be used with the date, or a value in a range from 100 to 32,767.
- *month*$_{in}$ is the month. This argument must be 0 if no month value is to be used with the date, or a value in a range from 1 (January) to 12 (December).
- *day*$_{in}$ is the day. This argument must be 0 if no day value is to be used with the date, or a value in a range from 1 to 31 that should be valid for the specified month and year.

The fourth overloaded function sets the date information according to an ASCII string. The following algorithm is used to determine the proper order and meaning of date values:

1—Any number greater than 31 is assumed to be the year.

2—If the number is less than 100, 1900 is added to the value. Year values below 100 are not allowed in the UI_DATE class.

3—Any number between 13 and 31 is assumed to be the day. In ambiguous situations where both the day and month values are less than 13, the country code date format (e.g., DTF_US_FORMAT, DTF_JAPANESE_FORMAT) is used to determine the order of date values.

- *returnValue*$_{out}$ returns one of the DTI_RESULT values listed with the first overloaded function.

- *string*$_{in}$ is a pointer to the ASCII date string. If this is an empty string (i.e., ""), the UI_DATE will be set to "blank." Passing a blank UI_DATE to the **UIW_DATE::-DataSet( )** function (see the DataSet section of "Chapter 49—UIW_DATE" for more information) will cause the date field to be displayed as blank space.

- *dtFlags*$_{in}$ gives information on how to interpret the date string. The following flags (declared in **UI_GEN.HPP**) override the country dependent information (supplied by the operating system):

  DTF_EUROPEAN_FORMAT—Forces the date to be interpreted in the European format (i.e., *day/month/year*), regardless of the default country information.

  DTF_JAPANESE_FORMAT—Forces the date to be interpreted in the Japanese format (i.e., *year/month/day*), regardless of the default country information.

  DTF_MILITARY_FORMAT—Forces the date to be interpreted in the U.S. Military format (i.e., *day/month/year* where *month* is a 3 letter abbreviated word), regardless of the default country information.

  DTF_NO_FLAGS—Does not associate any special flags with the UI_DATE object. In this case, the ASCII date will be interpreted using the default country information. This is the default argument if no other argument is provided. This flag should not be used in conjunction with any other DTF flag.

  DTF_SYSTEM—Fills a blank date with the system date. For example, if a blank date were specified by the programmer and the DTF_SYSTEM flag were set, the date would be set to the system date.

  DTF_US_FORMAT—Forces the ASCII date to be interpreted in the U.S. format (i.e., *month/day/year*), regardless of the default country information.

The fifth overloaded function sets the date information through a packed integer argument.

- *returnValue*$_{out}$ returns one of the DTI_RESULT values listed with the first overloaded function.

- *packedDate*$_{in}$ is a packed representation of the date (whose format is the same as the MS-DOS file dates). This argument is packed according to the following bit pattern:

bits 0-4 specify the day,
bits 5-8 specify the month, and
bits 9-15 specify the year minus 1980 (e.g., a value of 5 means 1985).

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_DATE date; // Initialize a system date.

    // Import the date in various forms and print out the results.
    char asciiDate[128];
    date.Import(1990, 1, 1);
    date.Export(asciiDate, DTF_NO_FLAGS);
    printf("ASCII date value: %s\n", asciiDate);
    date.Import("1-1-1990", DTF_NO_FLAGS);
    date.Export(asciiDate, DTF_MILITARY_FORMAT);
    printf("ASCII date value: %s\n", asciiDate);

    // The destructor for date is automatically called when the
    // scope of this function ends.
}
```

UI_DATE::operator =

Syntax

#include <ui_gen.hpp>

long operator = (long *days*);
or
long operator = (const UI_DATE &*date*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first overloaded operator assigns the value specified by *days* to the UI_DATE object.

- *returnValue*$_{out}$ is the number of days in the resulting date. This raw value is only used to transfer the current date to another UI_DATE object.

- *days*$_{in}$ is the date, given in the number of days, to be assigned to the UI_DATE object. This value does not necessarily correspond to any calendar year, but could be used to denote a period of time such as 1000 days.

The second overloaded operator assigns the value specified by *date* to the UI_DATE object.

- *returnValue*$_{out}$ is the number of days in the resulting date. This raw value is only used to transfer the current date to another UI_DATE object.

- *date*$_{in}$ is the date, specified by another UI_DATE object, to be assigned to the UI_DATE object.

Example

```
#include <ui_gen.hpp>

AddOneWeek(UI_DATE currentDate, UI_DATE &nextWeek)
{
    long oneWeek = 7;

    // Adding 1 week to the current date gives the next week.
    nextWeek = currentDate + oneWeek;
}
```

UI_DATE::operator +

Syntax

#include <ui_gen.hpp>

long operator + (long *days*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator adds the value *days* to the UI_DATE object.

- $returnValue_{out}$ is the number of days resulting from the addition operation. This raw value is only used to transfer the current date to another UI_DATE object.

- $days_{in}$ is the number of days to be added to the UI_DATE object. This value does not necessarily correspond to any calendar year, but could be used to denote a period of time such as 1000 days.

Example

```
#include <ui_gen.hpp>

AddOneWeek(UI_DATE currentDate, UI_DATE &nextWeek)
{
    long oneWeek = 7;

    // Adding 1 week to the current date gives the next week.
    nextWeek = currentDate + oneWeek;
}
```

UI_DATE::operator -

Syntax

#include <ui_gen.hpp>

long operator - (long *days*);
or
long operator - (const UI_DATE &*date*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first overloaded operator subtracts the value *days* from the UI_DATE object.

- *returnValue*$_{out}$ is the number of days resulting from the subtraction operation. This raw value is only used to transfer the current date to another UI_DATE object.

- *days*$_{in}$ is the number of days to be subtracted from the UI_DATE object. This value does not necessarily correspond to any calendar year, but could be used to denote a period of time such as 1000 days.

The second overloaded operator subtracts the date contained in *date* from the UI_DATE object.

- *returnValue*$_{out}$ is the difference, in days, between the UI_DATE object and the date contained in *date*.

- *date*$_{in}$ is the date to be subtracted from the UI_DATE object.

Example

```
#include <ui_gen.hpp>

SubtractOneWeek(UI_DATE currentDate, UI_DATE &lastWeek)
{
    long oneWeek = 7;

    // Subtracting 1 week from the current date gives the previous week.
    lastWeek = currentDate - oneWeek;
}
```

UI_DATE::operator >

Syntax

#include <ui_gen.hpp>

int operator > (const UI_DATE &*date*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator determines whether the UI_DATE object is chronologically greater than the date specified by *date*.

- *returnValue*$_{out}$ is TRUE if the UI_DATE object is chronologically greater than *date*. Otherwise, *returnValue* is FALSE.

- *date*$_{in}$ is the UI_DATE object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_DATE currentDate; // Initialize a system date.
    UI_DATE twentyFirstCentury("Jan. 1, 2000");

    // Check the dates.
    if (currentDate > twentyFirstCentury ||
        currentDate == twentyFirstCentury)
        printf("The twenty first century has already come.\n");
}
```

UI_DATE::operator >=

Syntax

#include <ui_gen.hpp>

int operator >= (const UI_DATE &*date*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator determines whether the UI_DATE object is chronologically greater than or equal to the date specified by *date*.

- *returnValue*$_{out}$ is TRUE if the UI_DATE object is chronologically greater than or equal to *date*. Otherwise, *returnValue* is FALSE.

- *date*$_{in}$ is the UI_DATE object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_DATE currentDate; // Initialize a system date.
    UI_DATE twentyFirstCentury("Jan. 1, 2000");

    // Check the dates.
    if (currentDate >= twentyFirstCentury)
        printf("The twenty first century has already come.\n");
}
```

UI_DATE::operator <

Syntax

#include <ui_gen.hpp>

int operator < (const UI_DATE &*date*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator determines whether the UI_DATE object is chronologically less than the date specified by *date*.

- *returnValue*$_{out}$ is TRUE if the UI_DATE object is chronologically less than *date*. Otherwise, *returnValue* is FALSE.

- *date*$_{in}$ is the UI_DATE object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_DATE currentDate; // Initialize a system date.
    UI_DATE twentyFirstCentury("Jan. 1, 2000");

    // Check the dates.
    if (currentDate < twentyFirstCentury)
        printf("It's not the twenty first century.\n");
}
```

UI_DATE::operator <=

Syntax

#include <ui_gen.hpp>

int operator <= (const UI_DATE *&date*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator determines whether the UI_DATE object is chronologically less than or equal to the date specified by *date*.

- $returnValue_{out}$ is TRUE if the UI_DATE object is chronologically less than or equal to *date*. Otherwise, *returnValue* is FALSE.

- $date_{in}$ is the UI_DATE object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_DATE currentDate; // Initialize a system date.
    UI_DATE endOfTwentiethCentury("Dec. 31, 1999");
```

```
    // Check the dates.
    if (currentDate <= endOfTwentiethCentury)
        printf("It's not the twenty first century.\n");
}
```

UI_DATE::operator ++

Syntax

#include <ui_gen.hpp>

long operator ++ (void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator increments the UI_DATE by one day.

- *returnValue*$_{out}$ is the number of days after the UI_DATE object has been incremented. This raw value is only used to update the current UI_DATE object.

Example

```
#include <ui_gen.hpp>

AdvanceCurrentDate(UI_DATE &currentDate)
{
    // Advance the current date.
    currentDate++;
}
```

UI_DATE::operator --

Syntax

#include <ui_gen.hpp>

long operator -- (void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator decrements the date in the UI_DATE object by one day.

- *returnValue*$_{out}$ is the number of days after the UI_DATE object has been decremented. This raw value is only used to update the current UI_DATE object.

Example

```
#include <ui_gen.hpp>

DecrementCurrentDate(UI_DATE &currentDate)
{
    // Decrement the current date.
    currentDate--;
}
```

UI_DATE::operator +=

Syntax

#include <ui_gen.hpp>

void operator += (long *days*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator adds *days* to the UI_DATE object and copies the result back into the UI_DATE object.

- $days_{in}$ is the number of days to be added to the UI_DATE object. This value does not necessarily correspond to any calendar year, but could be used to denote a period of time such as 1000 days.

Example

```
#include <ui_gen.hpp>

AddOneWeek(UI_DATE currentDate, UI_DATE &nextWeek)
{
    long oneWeek = 7;

    // Adding 1 week to the current date gives the next week.
    nextWeek = currentDate;
    nextWeek += oneWeek;
}
```

UI_DATE::operator -=

Syntax

#include <ui_gen.hpp>

void operator -= (long *days*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator subtracts *days* from the UI_DATE object and copies the result back into the UI_DATE object.

- $days_{in}$ is the date, given in the number of days, to be subtracted from the UI_DATE object. This value does not necessarily correspond to any calendar year, but could be used to denote a period of time such as 1000 days.

Example

```
#include <ui_gen.hpp>

SubtractWeeks(UI_DATE currentDate, UI_DATE &lastWeek)
{
    long oneWeek = 7;

    // Subtracting 1 week from the current date gives the previous week.
    lastWeek = currentDate;
    lastWeek -= oneWeek;
}
```

UI_DATE::operator ==

Syntax

#include <ui_gen.hpp>

int operator == (const UI_DATE &*date*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator determines whether the UI_DATE object is chronologically equal to the date specified by *date*.

- $returnValue_{out}$ is TRUE if the UI_DATE object is chronologically equal to *date*. Otherwise, *returnValue* is FALSE.

- *date*$_{in}$ is the UI_DATE object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_DATE currentDate; // Initialize a system date.
    UI_DATE newYears1990("Jan. 1, 1990");

    // Check the dates.
    if (currentDate == newYears1990)
        printf("It's new years day 1990.\n")
}
```

UI_DATE::operator !=

Syntax

#include <ui_gen.hpp>

int operator != (const UI_DATE &*date*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator determines whether the UI_DATE object is chronologically not equal to the date specified by *date*.

- *returnValue*$_{out}$ is TRUE if the UI_DATE object is chronologically not equal to *date*. Otherwise, *returnValue* is FALSE.

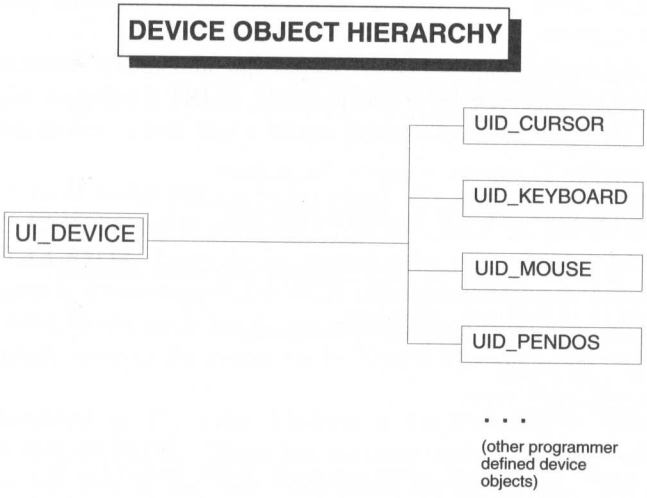
- *date*$_{in}$ is the UI_DATE object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_DATE currentDate; // Initialize a system date.
    UI_DATE newYears1990("Jan. 1, 1990");

    // Check the dates.
    if (currentDate != newYears1990)
        printf("It is not new years day 1990.\n")
}
```

CHAPTER 5 – UI_DEVICE

The UI_DEVICE class is an abstract class that defines basic information associated with input devices (e.g., keyboard, mouse). Since the UI_DEVICE class is abstract, it cannot be used as a constructed class. Rather, derived classes, such as UID_KEYBOARD, UID_CURSOR, UID_MOUSE or UID_PENDOS must be used. The figure below shows the device object hierarchy:



Classes derived from the UI_DEVICE base class include:

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

UID_KEYBOARD—A polled keyboard interface that retrieves keyboard information from the end-user.

UID_MOUSE—A polled mouse device that receives mouse information from the end-user.

UID_PENDOS—A pen-based device that supports handwritten characters (i.e., character recognition) for data entry.

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 UI_DEVICE class is declared in **UI_EVT.HPP**. Its public and protected members are:

```
enum ALT_STATE
{
    ALT_NOT_PRESSED,
    ALT_PRESSED_NO_EVENTS,
    ALT_PRESSED_EVENTS
};

class EXPORT UI_DEVICE : public UI_ELEMENT
{
    friend class EXPORT UI_EVENT_MANAGER;

public:
    // Members described in UI_DEVICE reference chapter.
    static ALT_STATE altState;
    static UI_DISPLAY *display;
    static UI_EVENT_MANAGER *eventManager;

    int installed;
    DEVICE_TYPE type;
    DEVICE_STATE state;

    virtual ~UI_DEVICE(void);
    virtual EVENT_TYPE Event(const UI_EVENT &event) = 0;

    // Members described in UI_ELEMENT reference chapter.
    UI_DEVICE *Next(void);
    UI_DEVICE *Previous(void);

protected:
    // Members described in UI_DEVICE reference chapter.
    UI_DEVICE(DEVICE_TYPE _type, DEVICE_STATE _state);
    static int CompareDevices(void *device1, void *device2);
    virtual void Poll(void) = 0;
};
```

- *ALT_STATE* contains values that are used to indicate the status of the ALT key when an event occurs. The following values are used:

  ALT_NOT_PRESSED—The <Alt> key has not been pressed.

  ALT_PRESSED_NO_EVENTS—The <Alt> key has been pressed, but no other input information has been received since the key was pressed.

  ALT_PRESSED_EVENTS—The <Alt> key continues to be pressed while another event has been received.

- *altState* is a static variable that indicates whether the keyboard <Alt> key is being pressed. It is used by the keyboard and mouse to detect the selection of <Alt> keys or else to send an <Alt> message if the <Alt> key is pressed and then released with no other keyboard or mouse event information.

- *display* is a pointer to a constructed display class. This variable is automatically set when the derived device is added to the Event Manager.

- *eventManager* is a pointer to a constructed Event Manager class. This variable is automatically set when the derived device is added to the Event Manager.

- *installed* indicates whether the input device was able to initialize itself. If installation is successful, *installed* is TRUE. If installation is not successful—for instance, if the mouse input device cannot find a mouse driver—*installed* is FALSE.

- *type* is the type of device that has been created. For example, the keyboard generates a type of E_KEY, the mouse generates a type of E_MOUSE and the cursor generates a type of E_CURSOR. Every device created either has a unique type or else it must have the generic device type E_DEVICE. The Event Manager uses the *type* information in order to recognize the device and send to it its proper messages when the Event Manager receives the events via its **Event( )** member function.

- *state* is described by D_ codes (defined in **UI_EVT.HPP**). These codes are: D_OFF, D_ON, D_HIDE. These act not only as the raw scan codes for event information but also as the state information for the device. For example, if a keyboard is in the D_OFF state, it will not receive any information. If a mouse is in the D_HIDE state, it will still be active, but it will not be visible on the screen.

UI_DEVICE::UI_DEVICE

Syntax

#include <ui_evt.hpp>

UI_DEVICE(DEVICE_TYPE *_type*, DEVICE_STATE *_state*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced constructor initializes the information associated with all devices derived from the UI_DEVICE base class. The UI_DEVICE constructor is protected since UI_DEVICE is an abstract class (i.e., only derived instances of UI_DEVICE should be made.)

- *type*$_{in}$ specifies the type of derived device that is to be initialized. The contents of this argument are copied into the protected *type* member variable. Zinc Application Framework reserves the values 0 though 99 for raw input devices. The following devices are defined within Zinc Application Framework:

  E_CURSOR(50)—Identification for the UID_CURSOR class.

  E_DEVICE(99)—Identification used to define a generic device.

  E_KEY(10)—Identification for the UID_KEYBOARD class.

  E_MOTIF(3)—Identification for Motif events.

  E_MOUSE(30)—Identification for the UID_MOUSE class.

  E_MSWINDOWS(1)—Identification for MS Windows events.

  E_OS2(2)—Identification for OS/2 events.

  The *type* value associated with each device is significant, because the Event Manager polls devices in ascending order. Each derived device class must either use the E_DEVICE type or have a type that is unique and is within the 0-99 value restrictions. The following additional raw device identifications are reserved by Zinc Application Framework for future use: 4, 11-19, 31-39, 51-59, 70-79 and 90-98. The remaining values 5-9, 20-29, 40-49, 60-69 and 80-89 can be used by the programmer.

- *initialState*$_{in}$ specifies the initial state of the derived device. The information contained in this argument depends on the type of device created but should either be D_OFF or D_ON.

Example

```
#include <ui_evt.hpp>

UID_CURSOR::UID_CURSOR(DEVICE_STATE initialState) :
    UI_DEVICE(E_CURSOR, initialState)
{
    .
    .
    .

}
```

UI_DEVICE::~UI_DEVICE

Syntax

#include <ui_evt.hpp>

virtual ~UI_DEVICE(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_DEVICE object. It is used when a derived device class is destroyed.

UI_DEVICE::CompareDevices

Syntax

#include <ui_evt.hpp>

static int CompareDevices(void *_device1_, void *_device2_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function is used to compare two devices according to their types.

- *returnValue*$_{out}$ is 0 if the two devices have the same type. *returnValue* is positive if *device1* has a greater device type than *device2*. *returnValue* is negative if *device1* has a lower device type than *device2*.
- *device1*$_{in}$ is a pointer to a UI_DEVICE object.
- *device2*$_{out}$ is a pointer to a UI_DEVICE object.

Example

```
#include <ui_evt.hpp>

UI_EVENT_MANAGER::UI_EVENT_MANAGER(UI_DISPLAY *_display, int _noOfElements) :
    UI_LIST(UI_DEVICE::CompareDevices), queueBlock(_noOfElements), level(1)
{
    display = _display;
    UI_DEVICE::display = display;
    UI_DEVICE::eventManager = this;
}
```

UI_DEVICE::Event

Syntax

#include <ui_evt.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*) = 0;

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function is a pure virtual function, so it has no declaration. This means that every class which is derived from UI_DEVICE must have an **Event( )** routine.

The **Event( )** routine is used to communicate run-time state information to a particular device. For example, the **UID_KEYBOARD::Event( )** routine can receive event information to turn on or off keyboard input. The type of information required by the **Event( )** function depends on the type of device that is to receive the message. The basic rules of event message passing are:

1—The *event.type* value must be either E_DEVICE or the specific type of device that is to receive the message. The Event Manager looks at the type of message to determine where to route information. For example, if a programmer wanted to send the D_HIDE message to all timer devices in the Event Manager, the following code could be used:

```
event.type = E_TIMER;
event.rawCode = D_HIDE;
event.region.left = 0;          // Define the whole screen.
event.region.top = 0;
event.region.right = display->columns - 1;
event.region.bottom = display->lines - 1;
eventManager->Event(event, E_TIMER);
```

If only the keyboard needed to be hidden, the *event.type* shown above could be changed to contain the E_KEY type. If the second parameter to UI_EVENT_-MANAGER::Event( ) were E_KEY, the Event Manager would only send the event message to the UID_KEYBOARD device, since its type is the only device containing the E_KEY value.

2—The *event.rawCode* must contain the state information. The following general messages can be sent to a device:

D_HIDE—Hides the device while the application paints information to the screen. This advanced message prevents the device from leaving blank areas on the screen when low-level screen painting operations are performed. In general, a programmer should not use this message. Window objects and display classes automatically hide the input devices when they paint information to the screen. If the D_HIDE message is used, the *event.region* information must contain the region that will be re-painted. This allows the affected device to only turn itself off when the affected region overlaps the device's screen position. This message must be used in conjunction with the D_ON message (described below) and should be used in the following order:

```
// Hide all devices before painting information to the screen.
UI_EVENT event;
event.type = E_DEVICE;
event.rawCode = D_HIDE;
event.region.left = 0;          // Define the whole screen.
event.region.top = 0;
event.region.right = display->columns - 1;
event.region.bottom = display->lines - 1;
eventManager->Event(event);
// Paint information directly to the screen.
.
.
.

// Show all devices that may have been shut off.
event.rawCode = D_ON;       // event.region was defined previously
eventManager->Event(event);
```

D_OFF—Turns the device off.

D_ON—Turns the device on.

S_DEINITIALIZE—Serves as a warning that the device is being subtracted from the Event Manager. This allows the device to halt further execution and/or prepare to be deleted.

S_INITIALIZE—Initializes internal information associated with the device. This message can be used when the device cannot initialize all its information at the time that the class constructor is called.

S_POSITION—Changes the screen position of the device. If this message is sent, *event.position.column* and *event.position.line* must contain the run-time display position of the device on the screen. The values of *event.position.column* and *event.position.line* depend on the type of display mode in which the application is running. For example, if a UID_CURSOR object is to be positioned at the center of the screen while the application is running in text mode (i.e., an 80 column by 25 line screen) the position values should be:

```
event.position.column = 40;
event.position.line = 13;
```

If, on the other hand, the application is running in a 640 column by 480 line graphics mode, the position values should be:

```
event.position.column = 320;
event.position.line = 240;
```

If the device is in a D_OFF state, the position change should be reflected when the device is turned back on.

In addition to the messages described above, each device may be sent private messages defined by the programmer. Zinc Application Framework reserves message numbers 0x0000 through 0x00FF. (Programmers may use any unsigned values greater than 0x00FF.)

NOTE: The chapters for the UID_CURSOR and UID_MOUSE classes can give more information about the types of private messages that can be passed in the *event.rawCode* variable.

Example

```
#include <ui_evt.hpp>

main()
{
    // Attach the keyboard to the event manager.
    UI_TEXT_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER eventManager(display);
    eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    .
    .
    .

    // Turn all the devices off.
    UI_EVENT event;
    event.type = E_DEVICE;
    event.rawCode = D_OFF;
    for (UI_DEVICE *device = eventManager.First(); device;
        device = device->Next())
        device->Event(event);
    .
    .
    .

}
```

UI_DEVICE::Poll

Syntax

#include <ui_evt.hpp>

virtual void Poll(void) = 0;

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function is a pure virtual function, so it has no declaration. This means that every class which is derived from UI_DEVICE must have a **Poll** routine.

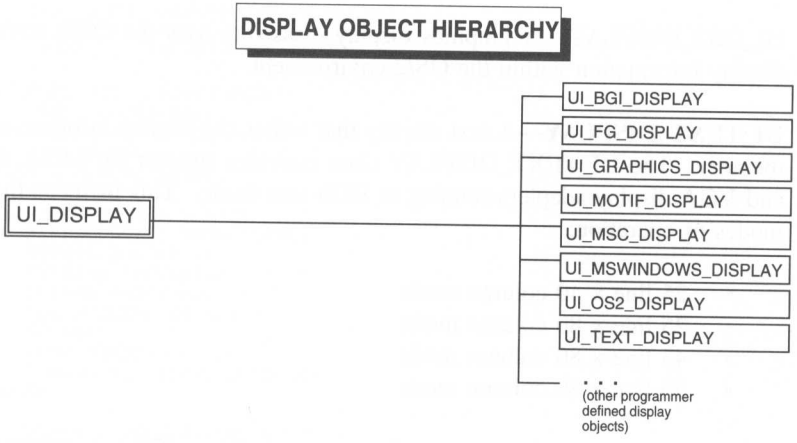
The **Poll( )** routine is used by **UI_EVENT_MANAGER::Get( )** to give input devices time to put event information into the event queue. For example, the **UID_-KEYBOARD::Poll( )** routine gets information from the keyboard BIOS and enters that information, as UI_EVENT information, into the Event Manager's queue of events.

Example

```
#include <ui_evt.hpp>

int UI_EVENT_MANAGER::Get(UI_EVENT &event, Q_FLAGS flags)
{
    // Stay in loop while no event conditions are met.
    int error = -1;
    do
    {
        // Call all the polled devices.
        UI_DEVICE *device;
        if (!FlagSet(flags, Q_NO_POLL))
            for (device = First(); device; device = device->Next())
                device->Poll();
        .
        .
        .

    } while (error);

    // Return the error status.
    return (error);
}
```

CHAPTER 6 – UI_DISPLAY

The UI_DISPLAY class defines basic information associated with the screen display. While the UI_DISPLAY class is not technically abstract (it contains no pure virtual functions), it should not be used as a constructed class. Rather, derived classes, such as UI_BGI_DISPLAY, UI_FG_DISPLAY, UI_MSC_DISPLAY, UI_MSWINDOWS_-DISPLAY or UI_TEXT_DISPLAY must be used. The graphic image below shows the display object hierarchy:



Classes derived from the UI_DISPLAY base class include:

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

UI_FG_DISPLAY—A graphics display that uses the Zortech Flash Graphics library package to display information to the screen. The UI_FG_DISPLAY class provides support for CGA, EGA, VGA, SVGA and Hercules monochrome display adapters running in graphics mode.

UI_GRAPHICS_DISPLAY—A graphics display that uses the GFX graphics routines to display information to the screen. The UI_GRAPHICS_DISPLAY class provides support for CGA, EGA, VGA, SVGA and Hercules monochrome display adapters running in graphics mode.

UI_MOTIF_DISPLAY—A graphics display class that uses the Motif environment to display information within the Motif environment.

UI_MSC_DISPLAY—A graphics display that uses the Microsoft C Graphics library package to display information to the screen. The UI_MSC_DISPLAY class provides support for CGA, EGA, VGA, SVGA and Hercules monochrome display adapters running in graphics mode.

UI_MSWINDOWS_DISPLAY—A graphics display that uses the Microsoft Windows environment to display information within the Windows environment.

UI_OS2_DISPLAY—A graphics display class that uses the OS/2 environment to display information within the OS/2 environment.

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

- 25 line x 40 column mode
- 25 line x 80 column mode
- 43 line x 80 column mode
- 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 definition 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.

NOTE: The UI_DISPLAY class maintains a set of reserved screen regions. When a display member function is called, a screen identification argument (*screenID*) is specified. This argument is matched against the screen's list of reserved regions to ensure that the object can be drawn to the area specified by the arguments. Only those regions with the same *screenID* are updated to the screen.

The UI_DISPLAY class is declared in **UI_DSP.HPP**. Its public and protected members are:

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

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

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

    virtual ~UI_DISPLAY(void);
    virtual void Bitmap(SCREENID screenID, int column, int line,
        int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette = NULL,
        const UI_REGION *clipRegion = NULL,
        HBITMAP *colorBitmap = NULL, HBITMAP *monoBitmap = NULL);
    virtual void BitmapArrayToHandle(SCREENID screenID, int bitmapWidth,
        int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette, HBITMAP *colorBitmap,
        HBITMAP *monoBitmap);
    virtual void BitmapHandleToArray(SCREENID screenID,
        HBITMAP colorBitmap, HBITMAP monoBitmap, int *bitmapWidth,
        int *bitmapHeight, UCHAR **bitmapArray);
    virtual void Ellipse(SCREENID screenID, int column, int line,
        int startAngle, int endAngle, int xRadius, int yRadius,
        const UI_PALETTE *palette, int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void IconArrayToHandle(SCREENID screenID, int iconWidth,
        int iconHeight, const UCHAR *iconArray, const UI_PALETTE *palette,
        HICON *icon);
    virtual void IconHandleToArray(SCREENID screenID, HICON icon,
        int *iconWidth, int *iconHeight, UCHAR **iconArray);
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground);
    virtual void Polygon(SCREENID screenID, int numPoints,
        const int *polygonPoints, const UI_PALETTE *palette,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
```

```
    void Rectangle(SCREENID screenID, const UI_REGION &region,
        const UI_PALETTE *palette, int width = 1, int fill = FALSE,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion, SCREENID screenID = ID_SCREEN);
    void RegionDefine(SCREENID screenID, const UI_REGION &region);
    virtual void RegionDefine(SCREENID screenID, int left, int top,
        int right, int bottom);
    virtual void RegionMove(const UI_REGION &oldRegion, int newColumn,
        int newLine, SCREENID oldScreenID = ID_SCREEN,
        SCREENID newScreenID = ID_SCREEN);
    virtual void Text(SCREENID screenID, int left, int top,
        const char *text, const UI_PALETTE *palette, int length = -1,
        int fill = TRUE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextWidth(const char *string, SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    int VirtualGet(SCREENID screenID, const UI_REGION &region);
    virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
        int bottom);
    virtual int VirtualPut(SCREENID screenID);

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

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

    UI_DISPLAY_IMAGE displayImage[MAX_DISPLAY_IMAGES];

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

- *installed* is a flag that tells whether the display has been installed. This member is set to FALSE by the base UI_DISPLAY class. You should set this variable to be TRUE if the graphics display installs correctly.

- *isText* indicates whether the display is running in text or graphics mode. If *isText* is TRUE, the application is running in text mode. Otherwise, the application is running in graphics mode.

- *isMono* is a flag that tells whether the display is operating in monochrome mode.

- *columns* and *lines* tell how many physical columns or lines are on the display. The following table shows the correlation of some modes of operation with the value for *columns* and *lines*:

Display mode	columns	lines
Text 80 column x 25 line	80	25
Text 40 column x 25 line	40	25
Text 80 column x 43 line	80	43
Text 80 column x 50 line	80	50
CGA 320 column x 200 line	320	200
MCGA 320 column x 200 line	320	200
EGA 640 column x 350 line	640	350
VGA 640 column x 480 line	640	480

- *cellWidth* and *cellHeight* are width and height values of a cell coordinate. If the application is running in text mode, *cellWidth* and *cellHeight* are 1. Otherwise, the value of *cellWidth* and *cellHeight* is determined by the graphics mode and default font size. For example, the UI_BGI_DISPLAY class constructor normally sets *cellWidth* to 7 and *cellHeight* to 23.

- *preSpace* denotes the size (in pixels) of the white space between the top border of a string field and the tallest character.

- *postSpace* denotes the size (in pixels) of the white space between the bottom border of a string field and the lowest character.

- *miniNumeratorX* and *miniDenominatorX* are values used to determine the width of a mini-cell. *miniNumeratorX* is set to 1 and *miniDenominatorX* is set to 10. (These values default to 1/10th of a cellwidth.)

- *miniNumeratorY* and *miniDenominatorY* are values used to determine the height of a mini-cell. *miniNumeratorY* is set to 1 and *miniDenominatorY* is set to 10. (These values default to 1/10th of a cellheight.)

- *backgroundPalette* is a pointer to the background color palette.

- *xorPalette* is a pointer to the XOR color palette.

- *colorMap* is a pointer to a palette table that contains the standard color entries. This palette table can be used to obtain a palette entry for basic colors including the appropriate grayscale or black-and-white colors when running on a grayscale or black-and-white display.

- *hInstance* is an instance handle that identifies the current instance of the application. This variable is only available in the Windows version of the library.

- *hPrevInstance* is an instance handle that identifies if the current instance of the application is the first instance. This variable is only available in the Windows version of the library.

- *nCmdShow* is a pointer to any parameters entered from the command line. This variable is only available in the Windows version of the library.

- *appContext* is the Xt Intrinsics application context. This variable is only available in the Motif version of the library.

- *topShell* is the initial application shell instance returned by **XtAppInitialize( )**.

- *xDisplay* is the X Window display. This variable is only available in the Motif version of the library.

- *xScreen* is a pointer to the X Window screen. This variable is only available in the Motif version of the library.

- *xScreenNumber* is the X Window screen number. This variable is only available in the Motif version of the library.

- *xGc* is the graphics context used in all graphics routines in UI_MOTIF_DISPLAY. This variable is only available in the Motif version of the library.

- *appClass* is the application class name used in the call to XtInitialize. This variable is set to "ZincApp" and is only available in the Motif version of the library. Zinc distributes a resource file called **ZincApp** that should be located in **/usr/lib/X11/app-defaults**. **ZincApp** is used to specify the fonts, background and other resources used by Zinc.

- *interleaveStipple* is a Pixmap specifying the interleave fill pattern. This variable is only available in the Motif version of the library.

UI_DISPLAY::UI_DISPLAY

Syntax

#include <ui_dsp.hpp>

UI_DISPLAY(int *isText*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced constructor returns a pointer to a new UI_DISPLAY object. The UI_DISPLAY constructor is protected, since only derived instances of UI_DISPLAY should be made.

- $isText_{in}$ indicates whether a text or graphics display is being created. This argument sets the member *isText* variable.

Example 1

```
#include <ui_win.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_MSC_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .

    // Restore the system.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

Example 2

```
#include <ui_dsp.hpp>

UI_MSC_DISPLAY::UI_MSC_DISPLAY(int mode):
    UI_DISPLAY(FALSE)
{
    .
    .
    .

}
```

UI_DISPLAY::~UI_DISPLAY

Syntax

#include <ui_dsp.hpp>

virtual ~UI_DISPLAY(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_DISPLAY object.

Example

```
#include <ui_win.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    UI_WINDOW_MANAGER *windowManager = UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
```

```
    // Restore the system.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

UI_DISPLAY::Bitmap

Syntax

#include <ui_dsp.hpp>

virtual void Bitmap(SCREENID *screenID*, int *column*, int *line*, int *bitmapWidth*, int *bitmapHeight*, const UCHAR **bitmapArray*, const UI_PALETTE **palette* = NULL, const UI_REGION **clipRegion* = NULL, HBITMAP *colorBitmap = NULL, HBITMAP *monoBitmap = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function draws a bitmap image to the screen. The bitmap is defined by an array of UCHAR (unsigned char) values where each array element represents a bitmap pixel color.

- $screenID_{in}$ is a screen object identification used to determine the parts of the bitmap that can be updated to the screen. Only those screen locations that match *screenID* are updated.

- $column_{in}$ and $line_{in}$ are the starting position of the bitmap (in pixel coordinates).

- $bitmapWidth_{in}$ and $bitmapHeight_{in}$ are the bitmap's pixel width and height.

- $bitmapArray_{in}$ is the bitmap pattern to be displayed. The bitmap pattern is mapped into an internal palette map (shown below). This ensures that the bitmap can be represented in graphics color modes as well as in various gray scale modes.

- *palette*$_{in}$ is a pointer to a set of palette entries that overrides the default palette map. The default palette map is shown below:

```
static UI_PALETTE _colorMap[16] =
{
    { ' ', attrib(BLACK, BLACK), attrib(MONO_BLACK, MONO_BLACK),
      PTN_SOLID_FILL, BLACK, BLACK, BW_BLACK, BW_BLACK, GS_BLACK,
      GS_BLACK },
    { ' ', attrib(BLUE, BLUE), attrib(MONO_DIM, MONO_DIM),
      PTN_SOLID_FILL, BLUE, BLUE, BW_BLACK, BW_BLACK, GS_GRAY,
      GS_GRAY },
    { ' ', attrib(GREEN, GREEN), attrib(MONO_DIM, MONO_DIM),
      PTN_SOLID_FILL, GREEN, GREEN, BW_BLACK, BW_BLACK, GS_GRAY,
      GS_GRAY },
    { ' ', attrib(CYAN, CYAN), attrib(MONO_DIM, MONO_DIM),
      PTN_SOLID_FILL, CYAN, CYAN, BW_BLACK, BW_BLACK, GS_GRAY,
      GS_GRAY },
    { ' ', attrib(RED, RED), attrib(MONO_DIM, MONO_BLACK),
      PTN_SOLID_FILL, RED, RED, BW_BLACK, BW_BLACK, GS_GRAY,
      GS_GRAY },
    { ' ', attrib(MAGENTA, MAGENTA), attrib(MONO_DIM, MONO_DIM),
      PTN_SOLID_FILL, MAGENTA, MAGENTA, BW_BLACK, BW_BLACK, GS_GRAY,
      GS_GRAY },
    { ' ', attrib(BROWN, BROWN), attrib(MONO_DIM, MONO_DIM),
      PTN_SOLID_FILL, BROWN, BROWN, BW_BLACK, BW_BLACK, GS_GRAY,
      GS_GRAY },
    { ' ', attrib(LIGHTGRAY, LIGHTGRAY), attrib(MONO_DIM, MONO_DIM),
      PTN_SOLID_FILL, LIGHTGRAY, LIGHTGRAY, BW_BLACK, BW_BLACK,
      GS_GRAY,GS_GRAY},
    { ' ', attrib(DARKGRAY, DARKGRAY), attrib(MONO_DIM, MONO_DIM),
      PTN_SOLID_FILL, DARKGRAY, DARKGRAY, BW_BLACK, BW_BLACK, GS_GRAY,
      GS_GRAY },
    { ' ', attrib(LIGHTBLUE, LIGHTBLUE), attrib(MONO_NORMAL, MONO_DIM),
      PTN_SOLID_FILL, LIGHTBLUE, LIGHTBLUE, BW_WHITE, BW_WHITE,
      GS_WHITE, GS_WHITE },
    { ' ', attrib(LIGHTGREEN, LIGHTGREEN), attrib(MONO_NORMAL,
      MONO_DIM),  PTN_SOLID_FILL, LIGHTGREEN, LIGHTGREEN, BW_WHITE,
      BW_WHITE, GS_WHITE, GS_WHITE },
    { ' ', attrib(LIGHTCYAN, LIGHTCYAN), attrib(MONO_NORMAL, MONO_DIM),
      PTN_SOLID_FILL, LIGHTCYAN, LIGHTCYAN, BW_WHITE, BW_WHITE,
      GS_WHITE, GS_WHITE },
    { ' ', attrib(LIGHTRED, LIGHTRED), attrib(MONO_NORMAL, MONO_DIM),
      PTN_SOLID_FILL, LIGHTRED, LIGHTRED, BW_WHITE, BW_WHITE,
      GS_WHITE, GS_WHITE },
    { ' ', attrib(LIGHTMAGENTA, LIGHTMAGENTA), attrib(MONO_NORMAL,
      MONO_DIM),  PTN_SOLID_FILL, LIGHTMAGENTA, LIGHTMAGENTA,
      BW_WHITE, BW_WHITE, GS_WHITE, GS_WHITE },
    { ' ', attrib(YELLOW, YELLOW), attrib(MONO_NORMAL, MONO_DIM),
      PTN_SOLID_FILL, YELLOW, YELLOW, BW_WHITE, BW_WHITE, GS_WHITE,
      GS_WHITE },
    { ' ', attrib(WHITE, WHITE), attrib(MONO_NORMAL, MONO_DIM),
      PTN_SOLID_FILL, WHITE, WHITE, BW_WHITE, BW_WHITE, GS_WHITE,
      GS_WHITE }
};
```

NOTE: If a palette map is provided it must contain entries for all possible bitmap colors.

- *clipRegion*$_{in}$ is a pointer to a region that specifies an additional clipping boundary (in addition to the boundary automatically determined by *screenID*) for the **Bitmap( )** function. If *clipRegion* is NULL, no additional clipping is performed.

- *colorBitmap*$_{in}$ is an operating system-specific bitmap handle that specifies the bitmap to be used when the system is operating in color mode.

- *monoBitmap*$_{in}$ is an operating system-specific bitmap handle that specifies the bitmap to be used when the system is operating in monochrome mode. If the system is operating in color mode and using DOS, the enabled bits of *monoBitmap* serve as a mask for *colorBitmap* and are displayed as transparent.

NOTE: Bitmaps do not have text screen equivalents. Thus, this routine should be used with caution.

Example

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

EVENT_TYPE UIW_ICON::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    int redisplay = FALSE;
    int border = FlagSet(woFlags, WOF_BORDER) ? 1 : 0;
    EVENT_TYPE ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_ICON);
    switch (ccode)
    {
    .
    .
    .

    }

    // Redisplay the object information.
    if (redisplay && ratioWidth == 1 && ratioWidth == 1 && !display->istext)
        display->Bitmap(screenID, iconRegion, bitmapWidth, bitmapHeight,
            bitmapArray);
    .
    .
    .

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

UI_DISPLAY::BitmapArrayToHandle

Syntax

```
#include <ui_dsp.hpp>

virtual void BitmapArrayToHandle(SCREENID screenID, int bitmapWidth,
    int bitmapHeight, const UCHAR *bitmapArray,
    const UI_PALETTE *palette, HBITMAP *colorBitmap,
    HBITMAP *monoBitmap);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function converts a bitmap array to two handles. Handles are pointers to environment specific storage structures that allow the bitmap to be drawn much faster than drawing individual pixels. The bitmap is defined by an array of UCHAR (unsigned char) values where each array element represents a bitmap pixel color.

- *screenID*$_{in}$ is a screen object identification used to determine the parts of the bitmap that can be updated to the screen. Only those screen locations that match *screenID* are updated.
- *bitmapWidth*$_{in}$ and *bitmapHeight*$_{in}$ are the bitmap's pixel width and height.
- *bitmapArray*$_{in}$ is the bitmap pattern to be converted. The bitmap pattern is mapped into an internal palette map. This ensures that the bitmap can be represented in graphics color modes as well as in various gray scale modes.
- *palette*$_{in}$ is a pointer to a set of palette entries that overrides the default palette map.
- *colorBitmap*$_{out}$ is an operating system-specific bitmap handle that specifies the bitmap to be used when the system is operating in color mode.
- *monoBitmap*$_{out}$ is an operating system-specific bitmap handle that specifies the bitmap to be used when the system is operating in monochrome mode. If the system is

operating in color mode and using DOS, the enabled bits of *monoBitmap* serve as a mask for *colorBitmap* and are displayed as transparent.

NOTE: Bitmaps do not have text screen equivalents. Thus, this routine should be used with caution.

Example

```
#include <ui_dsp.hpp>

void UI_OS2_DISPLAY::Bitmap(SCREENID screenID, int left, int top,
    int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
    const UI_PALETTE *palette, const UI_REGION *, HBITMAP *_colorBitmap,
    HBITMAP *_monoBitmap)
{
    .
    .
    .
    // Convert the bitmap array then draw the bitmap.
    if (!colorBitmap)
        BitmapArrayToHandle(screenID, bitmapWidth, bitmapHeight, bitmapArray,
            palette, &colorBitmap, &monoBitmap);
    .
    .
    .
}
```

UI_DISPLAY::BitmapHandleToArray

Syntax

#include <ui_dsp.hpp>

virtual void BitmapHandleToArray(SCREENID *screenID*, HBITMAP *colorBitmap*, HBITMAP *monoBitmap*, int **bitmapWidth*, int **bitmapHeight*, UCHAR ***bitmapArray*)

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function converts an environment-specific bitmap handle to an array of unsigned characters (UCHAR) values where each array element represents a bitmap pixel color.

- *screenID*$_{in}$ is a screen object identification used to determine the parts of the bitmap that can be updated to the screen. Only those screen locations that match *screenID* are updated.
- *colorBitmap*$_{in}$ is an operating system-specific color bitmap handle.
- *monoBitmap*$_{in}$ is an operating system-specific monochrome bitmap handle.
- *bitmapWidth*$_{out}$ and *bitmapHeight*$_{out}$ are the pixel width and height of the bitmap array.
- *bitmapArray*$_{out}$ is a pointer to an array of unsigned char (UCHAR) values where each array element represents a bitmap pixel color.

UI_DISPLAY::Ellipse

Syntax

#include <ui_dsp.hpp>

virtual void Ellipse(SCREENID *screenID*, int *column*, int *line*, int *startAngle*,
int *endAngle*, int *xRadius*, int *yRadius*, const UI_PALETTE **palette*,
int *fill* = FALSE, int *xor* = FALSE, const UI_REGION **clipRegion* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function draws and/or fills an arc, circle or ellipse on the screen.

- *screenID*$_{in}$ is a screen object identification used to determine the parts of the ellipse that can be updated to the screen. Only those screen locations that match *screenID* are updated.

- *column*$_{in}$ and *line*$_{in}$ are the center of the ellipse on the screen (in pixel coordinates).

- *startAngle*$_{in}$ and *endAngle*$_{in}$ are starting and ending angles of the ellipse. If a complete ellipse is desired, *startAngle* should be 0 and *endAngle* should be 360.

- *xRadius*$_{in}$ and *yRadius*$_{in}$ are the horizontal and vertical pixel radii of the ellipse.

- *palette*$_{in}$ is a pointer to the palette argument used when drawing the graphics line. The palette's foreground color is used to draw the border of the ellipse. The palette's background color is used to fill the ellipse (if *fill* is TRUE).

- *fill*$_{in}$ is a value that tells whether to fill the ellipse. If this value is TRUE, the ellipse is filled according to the specified palette's fill pattern and background color.

- *xor*$_{in}$ is a flag that tells whether the ellipse should be displayed according to an XOR attribute. If this value is TRUE, the ellipse is drawn using an XOR attribute.

- *clipRegion*$_{in}$ is a pointer to a region that specifies an additional clipping boundary (in addition to the boundary automatically determined by *screenID*) for the **Ellipse( )** function. If *clipRegion* is NULL, no additional clipping is performed.

NOTE: Ellipses do not have text screen equivalents. Thus, this routine should be used with caution.

Example

```
#include <ui_dsp.hpp>

class CIRCLE : public UI_WINDOW_OBJECT
{
public:
    .
    .
    .

    int fill;
};

EVENT_TYPE CIRCLE::Event(const UI_EVENT &event)
{
    EVENT_TYPE ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_CIRCLE);

    switch (ccode)
    {
    case S_DISPLAY_INACTIVE:
```

```
    case S_DISPLAY_ACTIVE:
        if (display->isText || !UI_WINDOW_OBJECT::NeedsUpdate(event, ccode))
            break;

        // Set up a temporary clip region then draw the circle.
        display->RegionDefine(screenID | 0x1000, true);
        int column = true.left + (relative.right - relative.left) / 2;
        int line = true.top + (relative.bottom - relative.top) / 2;
        int radius = (relative.bottom - relative.top) / 2;
        if (radius < (relative.right - relative.left) / 2)
            radius = (relative.right - relative.left) / 2;
        display->Ellipse(screenID | 0x1000, column, line, 0, 360, radius,
            radius, lastPalette, fill);
        // Restore the normal region.
        display->RegionDefine(screenID, true);
        break;


    .
    .
    .

    }

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

UI_DISPLAY::IconArrayToHandle

Syntax

#include <ui_dsp.hpp>

virtual void IconArrayToHandle(SCREENID *screenID*, int *iconWidth*, int *iconHeight*, const UCHAR **iconArray*, const UI_PALETTE **palette*, HICON **icon*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function converts a bitmap array to an icon handle. A handle is a pointer to an environment specific storage structure that allows the icon to be drawn much faster than drawing individual pixels. The bitmap is defined by an array of UCHAR (unsigned char) values where each array element represents a bitmap pixel color.

- *screenID*$_{in}$ is a screen object identification used to determine the parts of the icon that can be updated to the screen. Only those screen locations that match *screenID* are updated.

- *iconWidth*$_{in}$ and *iconHeight*$_{in}$ are the icon's pixel width and height.

- *iconArray*$_{in}$ is the icon pattern. The icon pattern is mapped into an internal palette map. This ensures that the icon can be represented in graphics color modes as well as in various gray scale modes.

- *palette*$_{in}$ is a pointer to a set of palette entries that overrides the default palette map.

- *icon*$_{in}$ is an operating system-specific icon that specifies the icon image.

NOTE: Icons do not have text screen equivalents. Thus, this routine should be used with caution.

UI_DISPLAY::IconHandleToArray

Syntax

#include <ui_dsp.hpp>

virtual void IconHandleToArray(SCREENID *screenID*, HICON *icon*, int **iconWidth*, int **iconHeight*, UCHAR ***iconArray*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function converts an icon image to an array of unsigned characters (UCHAR) values where each array element represents a bitmap pixel color.

- *screenID*$_{in}$ is a screen object identification used to determine the parts of the icon that can be updated to the screen. Only those screen locations that match *screenID* are updated.

- *icon*$_{in}$ is the icon whose image is to be converted.
- *iconWidth*$_{out}$ and *iconHeight*$_{out}$ are the pixel width and height of the icon array.
- *iconArray*$_{out}$ is a pointer to an array of UCHAR (unsigned char) values where each array element represents a bitmap pixel color.

UI_DISPLAY::Line

Syntax

```
#include <ui_dsp.hpp>

virtual void Line(SCREENID screenID, int column1, int line1, int column2, int line2,
    const UI_PALETTE *palette, int width = 1, int xor = FALSE,
    const UI_REGION *clipRegion = NULL);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function draws a line between two points on the screen.

- *screenID*$_{in}$ is a screen object identification used to determine the parts of the line that can be updated to the screen. Only those screen locations that match *screenID* are updated.
- *column1*$_{in}$ and *line1*$_{in}$ are the starting position of the line on the screen (in pixel coordinates).
- *column2*$_{in}$ and *line2*$_{in}$ are the ending position of the line on the screen (in pixel coordinates).
- *palette*$_{in}$ is a pointer to the palette argument used when drawing the graphics line. The palette's foreground color is used to draw the line.

- $width_{in}$ is the width (in pixels) of the line.

- xor_{in} is a flag that tells whether the line should be displayed according to an XOR attribute. If this value is TRUE, the line is shown using an XOR attribute.

- $clipRegion_{in}$ is a pointer to a region that specifies an additional clipping boundary (in addition to the boundary automatically determined by *screenID*) for the **Line( )** function. If *clipRegion* is NULL, no additional clipping is performed.

Example

```
void UI_WINDOW_OBJECT::Border(EVENT_TYPE ccode, UI_REGION &region,
    const UI_PALETTE *palette)
{
    // Determine the border and update the region.
    region = true;
    .
    .
    .

    int displayBorder = (ccode == S_DISPLAY_ACTIVE ||
        ccode == S_NON_CURRENT || ccode == S_DISPLAY_INACTIVE ||
        ccode == S_CURRENT) ? TRUE : FALSE;


    if (displayBorder && palette)
    {
        UI_PALETTE tPalette = *palette;
        tPalette.colorForeground = tPalette.colorBackground;
        tPalette.bwForeground = tPalette.bwBackground;
        tPalette.grayScaleForeground = tPalette.grayScaleBackground;
        display->Line(screenID, region.left, region.top, region.left,
            region.bottom, &tPalette);
        display->Line(screenID, region.right, region.top, region.right,
            region.bottom, &tPalette);
        .
        .
        .

    }
    region.left++;
    region.right--;
    .
    .
    .

}
```

UI_DISPLAY::MapColor

Syntax

#include <ui_dsp.hpp>

virtual COLOR MapColor(const UI_PALETTE **palette*, int *isForeground*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function gives the actual color according to the type of display and access. For example, if the application is running in full color mode and the foreground is requested, this function returns the palette's foreground color attribute. If, however, the application is running in a gray-scale mode, the same call returns the gray scale foreground color.

- *returnValue*$_{out}$ is the actual color to be set.
- *palette*$_{in}$ is the palette whose colors are to be determined.
- *isForeground*$_{in}$ is a flag that tells whether a foreground or background color is desired. If this value is TRUE, the foreground color will be returned.

UI_DISPLAY::Polygon

Syntax

#include <ui_dsp.hpp>

virtual void Polygon(SCREENID *screenID*, int *numPoints*, const int **polygonPoints*,
const UI_PALETTE **palette*, int *fill* = FALSE, int *xor* = FALSE,
const UI_REGION **clipRegion* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function draws and/or fills a polygon on the screen.

- *screenID*$_{in}$ is a screen object identification used to determine the parts of the polygon that can be updated to the screen. Only those screen locations that match *screenID* are updated.

- *numPoints*$_{in}$ is the number of points in the polygon.

- *polygonPoints*$_{in}$ is a pointer to a sequence of integers (i.e., *numPoints* x 2). Each integer pair gives a column and line point on the polygon.

- *palette*$_{in}$ is a pointer to the palette argument used when drawing the polygon. The palette's foreground color is used to draw the border of the polygon. The palette's background color is used to fill the polygon (if *fill* is TRUE).

- *fill*$_{in}$ is a value that tells whether to fill the polygon. If this value is TRUE, the polygon is filled according to the specified palette's fill pattern and background color.

- *xor*$_{in}$ is a flag that tells whether the polygon should be displayed according to an XOR attribute. If this value is TRUE, the polygon is shown using an XOR attribute.

- *clipRegion*$_{in}$ is a pointer to a region that specifies an additional clipping boundary (in addition to the boundary automatically determined by *screenID*) for the **Polygon( )** function. If *clipRegion* is NULL, no additional clipping is performed.

NOTE: Polygons do not have text screen equivalents. Thus, this routine should be used with caution.

Example

```
#include <ui_dsp.hpp>

class TRIANGLE : public UI_WINDOW_OBJECT
{
public:
    .
    .
    .

    int fill;
};

EVENT_TYPE TRIANGLE::Event(const UI_EVENT &event)
{
    EVENT_TYPE ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_TRIANGLE);
    switch (ccode)
    {
    case S_DISPLAY_INACTIVE:
    case S_DISPLAY_ACTIVE:
        if (display->isText || !UI_WINDOW_OBJECT::NeedsUpdate(event, ccode))
            break;

        // Set up a temporary clip region then draw the triangle.
        display->RegionDefine(screenID | 0x1000, true);
        int triangle[8];
        triangle[0] = triangle[6] =
            true.left + (relative.right - relative.left) / 2;
        triangle[1] = triangle[7] = true.top;
        triangle[2] = true.left;
        triangle[3] = triangle[5] = true.top + relative.bottom;
        triangle[4] = true.left + relative.right;
        display->Polygon(screenID | 0x1000, 4, triangle, lastPalette, fill);
        // Restore the normal region.
        display->RegionDefine(screenID, true);
        break;
    .
    .
    .

    }

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

UI_DISPLAY::Rectangle

Syntax

#include <ui_dsp.hpp>

void Rectangle(SCREENID *screenID*, const UI_REGION &*region*,
const UI_PALETTE **palette*, int *width* = 1, int *fill* = FALSE, int *xor* = FALSE,
const UI_REGION **clipRegion* = NULL);
or

virtual void Rectangle(SCREENID *screenID*, int *left*, int *top*, int *right*, int *bottom*, const UI_PALETTE **palette*, int *width* = 1, int *fill* = FALSE, int *xor* = FALSE, const UI_REGION **clipRegion* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded functions draw and/or fill a bar or rectangle on the screen.

The first virtual function draws a rectangular box according to the region argument.

- $screenID_{in}$ is a screen object identification used to determine the parts of the rectangle that can be updated to the screen. Only those screen locations that match *screenID* are updated.

- $region_{in}$ is the region whose box is to be drawn.

- $palette_{in}$ is a pointer to the palette argument used when drawing the rectangle. The palette's foreground color is used to draw the border of the rectangle. The palette's background color is used to fill the rectangle (if *fill* is TRUE).

- $width_{in}$ specifies the width of the rectangle's border. If the application is running in text mode, *width* is given in cell widths. Otherwise, *width* is given in pixel coordinates.

- $fill_{in}$ is a value that tells whether to fill the rectangle. If this value is TRUE, the rectangle is filled according to the specified palette's fill pattern and background color.

- xor_{in} is a flag that tells whether the rectangle should be displayed according to an XOR attribute. If this value is TRUE, the rectangle is shown using an XOR attribute.

- $clipRegion_{in}$ is a pointer to a region that specifies an additional clipping boundary (in addition to the boundary automatically determined by *screenID*) for the **Rectangle( )** function. If *clipRegion* is NULL, no additional clipping is performed.

The second virtual function draws a rectangular box according to two points (left-top and right-bottom) that form the rectangular region.

- *screenID*$_{in}$ is a screen object identification used to determine the parts of the rectangle that can be updated to the screen. Only those screen locations that match *screenID* are updated.

- *left*$_{in}$ and *top*$_{in}$ are the starting position of the rectangle.

- *right*$_{in}$ and *bottom*$_{in}$ are the ending position of the rectangle.

- *palette*$_{in}$ is a pointer to the palette argument used when drawing the rectangle. The palette's foreground color is used to draw the border of the rectangle. The palette's background color is used to fill the rectangle (if *fill* is TRUE).

- *width*$_{in}$ specifies the width of the rectangle's border. If the application is running in text mode, *width* is given in cell widths. Otherwise, *width* is given in pixel coordinates.

- *fill*$_{in}$ is a value that tells whether to fill the rectangle. If this value is TRUE, the rectangle is filled according to the specified palette's fill pattern and background color.

- *xor*$_{in}$ is a flag that tells whether the rectangle should be displayed according to an XOR attribute. If this value is TRUE, the rectangle is shown using an XOR attribute.

- *clipRegion*$_{in}$ is a pointer to a region that specifies an additional clipping boundary (in addition to the boundary automatically determined by *screenID*) for the **Rectangle** function. If *clipRegion* is NULL, no additional clipping is performed.

Example

```
#include <ui_win.hpp>

EVENT_TYPE UIW_BORDER::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    UI_REGION region;
    EVENT_TYPE ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_BORDER);
    switch (ccode)
    {
    case S_DISPLAY_INACTIVE:
    case S_DISPLAY_ACTIVE:
        // Draw the borders around the object.
        if (!display->isText && !UI_WINDOW_OBJECT::NeedsUpdate(event, ccode))
            break;
        UI_PALETTE *palette = UI_WINDOW_OBJECT::LogicalPalette(ccode);
        lastPalette = palette;
```

```
        if (display->isText)
            display->Rectangle(screenID, true, palette,
                (ccode == S_DISPLAY_ACTIVE) ? 2 : 1);
        else
        {
            region = parent->true;
            eventManager->DevicesHide(parent->true);
            UI_PALETTE *outlinePalette = MapPalette(paletteMapTable,
                PM_ACTIVE, ID_BLACK_SHADOW);
            display->Rectangle(screenID, true, outlinePalette);
            display->Rectangle(screenID, region, outlinePalette);
            UI_WINDOW_OBJECT::Shadow(region, 1);

            // Display the top and bottom lines.
            int temp = region.bottom;
            region.bottom = true.top - 1;
            display->Rectangle(screenID, region, palette, 0, TRUE);
            region.bottom = temp;

            temp = region.top;
            region.top = true.bottom + 1;
            display->Rectangle(screenID, region, palette, 0, TRUE);
            region.top = temp;
        .
        .
        .

    }

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

UI_DISPLAY::RectangleXORDiff

Syntax

#include <ui_dsp.hpp>

virtual void RectangleXORDiff(const UI_REGION *&oldRegion*,
const UI_REGION *&newRegion*, SCREENID *screenID* = ID_SCREEN);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced virtual function draws the difference of two XOR rectangles to the screen.

The two parameters allow the programmer to specify an old XOR rectangle that has been set previously by **Rectangle( )** where its *xor* flag was set to TRUE.

- *oldRegion*$_{in}$ is the old XOR region.
- *newRegion*$_{in}$ is the new XOR region.
- *screenID*$_{in}$ is a screen object identification used to determine the parts of the rectangle that can be updated to the screen. Only those screen locations that match *screenID* are updated.

Example

```
#include <ui_win.hpp>

void UI_WINDOW_MANAGER::Modify(UI_WINDOW_OBJECT *object,
    const UI_EVENT &event)
{
    UI_REGION newRegion = object->true;
    UI_REGION oldRegion = newRegion;

    .
    .
    .

    // Update the new region.
    if (oldRegion.left != newRegion.left ||
        oldRegion.top != newRegion.top ||
        oldRegion.right != newRegion.right ||
        oldRegion.bottom != newRegion.bottom)
    {
        // Compute the lower-right coordinates.
        newRegion.right = newRegion.left + width - 1;
        newRegion.bottom = newRegion.top + height - 1;

        // Remove the old region and update the new region.
        if (eventManager->Get(tEvent, Q_NO_BLOCK | Q_NO_DESTROY) != 0 ||
            MapEvent(eventMapTable, tEvent, ID_WINDOW_OBJECT, ID_WINDOW_OBJECT)
            != L_CONTINUE_SELECT)
        {
            display->RectangleXORDiff(oldRegion, newRegion);
            oldRegion = newRegion;
        }
    }
    .
    .
    .

}
```

UI_DISPLAY::RegionDefine

Syntax

#include <ui_dsp.hpp>

void RegionDefine(SCREENID *screenID*, const UI_REGION &*region*);
or
virtual void RegionDefine(SCREENID *screenID*, int *left*, int *top*, int *right*, int *bottom*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These advanced overloaded functions are used to reserve a specified region of the screen with a particular identification value.

The first function defines the region according to a rectangular box.

- $screenID_{in}$ is the identification to use to define the region. Once a region has been defined, only those objects with the same screen identification will be allowed to write information to the screen region bound by *region*.

- $region_{in}$ is the region whose boundary is to be reserved. The coordinates 0, 0, 0xFFFF and 0xFFFF reserve the whole screen.

The second virtual function defines the region according to two points (left-top, right-bottom) that form the rectangular region.

- $screenID_{in}$ is the identification to use to define the region. Once a region has been defined, only those objects with the same screen identification will be allowed to write information to the screen region bound by the region points.

- $left_{in}$, top_{in}, $right_{in}$ and $bottom_{in}$ are values that define the region to be reserved. The coordinates 0, 0, 0xFFFF and 0xFFFF reserve the whole screen.

Example

```
#include <ui_win.hpp>

EVENT_TYPE UIW_ICON::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    int redisplay = FALSE;
    int border = FlagSet(woFlags, WOF_BORDER) ? 1 : 0;
    EVENT_TYPE ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_ICON);
    switch (ccode)
    {
    case S_DEFINE_REGION:
        if (!parent)
        {
            if (!display->isText)
                display->RegionDefine(screenID, iconRegion);
            if (string)
                display->RegionDefine(screenID, stringRegion);
        }
        break;
    .
    .
    .

    }

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

UI_DISPLAY::RegionMove

Syntax

#include <ui_dsp.hpp>

virtual void RegionMove(const UI_REGION &*region*, int *newColumn*, int *newLine*,
    SCREENID *oldScreenID* = ID_SCREEN,
    SCREENID *newScreenID* = ID_SCREEN);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function copies the screen contents of a particular region to a new area on the screen.

- *region*$_{in}$ is a reference pointer to the region whose screen contents are to be moved.
- *newColumn*$_{in}$ and *newLine*$_{in}$ are the new starting position where the old region information is to be moved.
- *oldScreenID*$_{in}$ and *newScreenID*$_{in}$ are advanced arguments used by Microsoft Windows to identify the source and destination window handles.

Example

```
#include <ui_win.hpp>

void UI_WINDOW_OBJECT::Modify(UI_WINDOW_OBJECT *object,
    const UI_EVENT &event)
{
    UI_REGION newRegion = object->true;
    UI_REGION oldRegion = newRegion;
    .
    .
    .

    // Move the region.
    if (!sizeObject)
        display->RegionMove(object->true, newRegion.left, newRegion.top);
}
```

UI_DISPLAY::Text

Syntax

#include <ui_dsp.hpp>

virtual void Text(SCREENID *screenID*, int *left*, int *top*, const char **text*,
const UI_PALETTE **palette*, int *length* = -1, int *fill* = TRUE, int *xor* = FALSE,
const UI_REGION **clipRegion* = NULL,
LOGICAL_FONT *font* = FNT_DIALOG_FONT);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function draws a text string to the screen.

- *screenID*$_{in}$ is a screen object identification used to determine the parts of the text string that can be updated to the screen. Only those screen locations that match *screenID* are updated.

- *left*$_{in}$ and *top*$_{in}$ are the starting position of the text on the screen. If the application is running in text mode, *left* and *top* are given in cell widths. Otherwise, *left* and *top* are given in pixel coordinates.

- *text*$_{in}$ is a pointer to the text to be displayed to the screen.

- *palette*$_{in}$ is a pointer to the palette argument used when drawing the text. The palette's foreground color is used to draw the characters of the text. The palette's background color is used to fill the region behind the text (if *fill* is TRUE).

- *length*$_{in}$ is the number of characters to display to the screen. If *length* is -1, the string is displayed until the '\0' character is found.

- *fill*$_{in}$ is a value that tells whether to fill the rectangular space occupied by *text*. If this value is TRUE, the region is filled according to the specified palette's background color.

- *xor*$_{in}$ is a flag that tells whether the rectangular space occupied by *text* should be displayed according to an XOR attribute. If this value is TRUE, *text* is displayed according to the XOR attribute.

- *clipRegion*$_{in}$ is a pointer to a region that specifies an additional clipping boundary (in addition to the boundary automatically determined by *screenID*) for the **Text( )** function. If *clipRegion* is NULL, no additional clipping is performed.

- *font*$_{in}$ is the font to be used when drawing the text string to the screen.

Example

```
#include <ui_win.hpp>

EVENT_TYPE UI_WINDOW_OBJECT::DrawText(SCREENID screenID, UI_REGION &tRegion,
    const char *text, UI_PALETTE *palette)
{
    int length = ui_strlen(text);
    WORD wFormat = DT_SINGLELINE | DT_VCENTER;
```

```
        if (FlagSet(woFlags, WOF_JUSTIFY_CENTER))
            wFormat |= DT_CENTER;
        else if (FlagSet(woFlags, WOF_JUSTIFY_RIGHT))
            wFormat |= DT_RIGHT;
        COLORREF oldForeground = SetTextColor(hDC, foreground);
        COLORREF oldBackground = SetBkColor(hDC, background);
        ::DrawText(hDC, (LPSTR)text, length, &region, wFormat);
        SetTextColor(hDC, oldForeground);
        SetBkColor(hDC, oldBackground);
        if (screenID != ID_DIRECT)
            EndPaint(screenID, &ps);
        return (TRUE);
    }
```

UI_DISPLAY::TextHeight

Syntax

#include <ui_dsp.hpp>

virtual int TextHeight(const char *_string_, SCREENID _screenID_ = ID_SCREEN, LOGICAL_FONT _font_ = FNT_DIALOG_FONT);

Portability

This virtual function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns the height of a specified string.

- $returnValue_{out}$ is the height of the string. If the application is running in text mode, this value is always 1. Otherwise, *returnValue* is the pixel height of the string.
- $string_{in}$ is a pointer to the string whose height is to be determined.
- $screenID_{in}$ is a screen object identification used to identify regions of the screen.
- $font_{in}$ is the font to be used when measuring the text string.

Example

```
#include <ui_win.hpp>

void UI_WINDOW_OBJECT::Text(char *string, int depth, int ccode,
    const UI_PALETTE *palette)
{
    // Display the text to the screen.
    .
    .
    .

    // Make sure it is a valid string.
    if (string == 0 || string[0] == '\0')
        return;

    // See if the string will fit.
    int height = display->TextHeight(string);
    if (region.bottom - region.top + 1 < height)
        return;

    char scrapBuffer[128];
    strncpy(scrapBuffer, string, 128);
    scrapBuffer[127] = '\0';
    char *hotKey = strchr(scrapBuffer, '~');
    if (hotKey)
        strcpy(hotKey, hotKey + 1);
            .
    int width = display->TextWidth(scrapBuffer);
    if (width > region.right - region.left + 1)
    {
        width = region.right - region.left;
        scrapBuffer[width / display->cellWidth] = '\0';
    }
    .
    .
    .

}
```

UI_DISPLAY::TextWidth

Syntax

#include <ui_dsp.hpp>

virtual int TextWidth(const char **string*, SCREENID *screenID* = ID_SCREEN,
    LOGICAL_FONT *font* = FNT_DIALOG_FONT);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function returns the width of a specified string.

- *returnValue*$_{out}$ is the width of the string. If the application is running in text mode, this value is the character width of the string. Otherwise, *returnValue* is the pixel width of the string.

- *string*$_{in}$ is a pointer to the string whose width is to be determined.

- *screenID*$_{in}$ is a screen object identification used to identify regions of the screen.

- *font*$_{in}$ is the font to be used when measuring the text string.

Example

```
#include <ui_win.hpp>

void UI_WINDOW_OBJECT::Text(char *string, int depth, int ccode,
    const UI_PALETTE *palette)
{
    // Display the text to the screen.
    .
    .
    .

    // Make sure it is a valid string.
    if (string == 0 || string[0] == '\0')
        return;


    // See if the string will fit.
    int height = display->TextHeight(string);
    if (region.bottom - region.top + 1 < height)
        return;

    char scrapBuffer[128];
    strncpy(scrapBuffer, string, 128);
    scrapBuffer[127] = '\0';
    char *hotKey = strchr(scrapBuffer, '~');
    if (hotKey)
        strcpy(hotKey, hotKey + 1);
    int width = display->TextWidth(scrapBuffer);
    if (width > region.right - region.left + 1)
    {
        width = region.right - region.left;
        scrapBuffer[width / display->cellWidth] = '\0';
    }
    .
    .
    .

}
```

UI_DISPLAY::VirtualGet

Syntax

```
#include <ui_dsp.hpp>

int VirtualGet(SCREENID screenID, const UI_REGION &region);
    or
virtual int VirtualGet(SCREENID screenID, int left, int top, int right, int bottom);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded functions attempt to optimize multiple successive drawing calls. In DOS text mode, this function will copy a section of the display to a video buffer (defined by the library) in memory. When a certain area of the screen is going to be drawn or re-drawn many times, it may be advantageous to copy that portion of the screen into memory, do all of the drawing, and then copy the modified 'virtual screen' back onto the actual screen.

In graphics modes, this function will suppress drawing of any device images. This will allow for successive drawing calls to be made without re-drawing the device images every time.

The first overloaded function uses the region specified by *region* to copy that portion of the screen into the virtual screen buffer or to suppress device image re-drawing within that region, as appropriate for the environment.

- $screenID_{in}$ is a screen object identification used to identify regions of the screen.
- $region_{in/out}$ specifies the region of the screen which will be copied into the virtual screen buffer or for which device image re-drawing is to be suppressed.

The second overloaded function uses the region specified by *left*, *top*, *right* and *bottom*, to copy that portion of the screen into the virtual screen buffer or to suppress device image re-drawing within that region, as appropriate for the environment.

- *left*$_{in}$ specifies the left boundary of the screen region which will be copied into the virtual screen buffer or for which device image re-drawing will be suppressed.

- *top*$_{in}$ specifies the top boundary of the screen region which will be copied into the virtual screen buffer or for which device image re-drawing will be suppressed.

- *right*$_{in}$ specifies the right boundary of the screen region which will be copied into the virtual screen buffer or for which device image re-drawing will be suppressed.

- *bottom*$_{in}$ specifies the bottom boundary of the screen region which will be copied into the virtual screen buffer or for which device image re-drawing will be suppressed.

Example

```
WO_GRAPH::DrawX(UI_DISPLAY *display)
{
    // Copy the screen into the virtual buffer.
    display->VirtualGet(screenID, 0, 0, 100, 100);

    display->Line(screenID, 0, 0, 100, 100);
    display->Line(screenID, 0, 100, 100, 0);

    // Copy the virtual buffer back to the screen.
    display->VirtualPut(screenID);
}
```

UI_DISPLAY::VirtualPut

Syntax

#include <ui_dsp.hpp>

virtual int VirtualPut(SCREENID *screenID*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function copies the virtual screen buffer back on to the screen location from where it was copied by the call to **VirtualGet( )**, if in DOS text mode. If in graphics modes this function allows device image re-drawing to take place again.

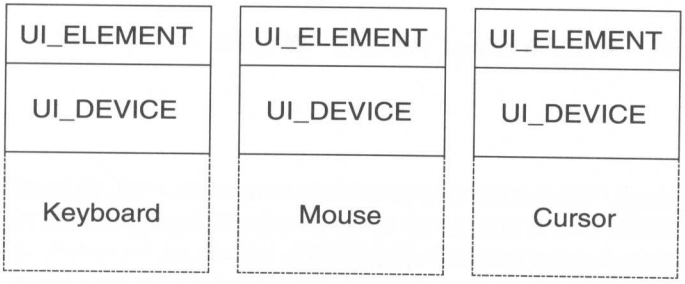
- *screenID*$_{in}$ is a screen object identification used to identify regions of the screen.

Example

```
WO_GRAPH::DrawX(UI_DISPLAY *display)
{
    // Copy the screen into the virtual buffer.
    display->VirtualGet(0, 0, 100, 100);

    display->line(0, 0, 100, 100);
    display->line(0, 100, 100, 0);

    // Copy the virtual buffer back to the screen.
    display->VirtualPut(screenID);
}
```

CHAPTER 7 – UI_ELEMENT

The UI_ELEMENT class serves as the base class to all window object classes, all input device classes and several other specialized classes in Zinc Application Framework. List elements allow the programmer to tie related objects together using UI_ELEMENT pointers to each object. UI_ELEMENT pointers allow different types of derived objects to be referred to by their base class (i.e., UI_ELEMENT). This allows different objects to be managed in the same linked-list. Classes derived from the UI_ELEMENT base class can be viewed in the following manner:



NOTE: In the figure above, the solid line denotes the base class (i.e., UI_ELEMENT) and the dotted line shows the possible logical extensions of a derived class.

The UI_ELEMENT class is declared in **UI_GEN.HPP**. Its public and protected members are:

```
class EXPORT UI_ELEMENT
{
    friend class EXPORT UI_LIST;
public:
    virtual ~UI_ELEMENT(void);
    virtual void *Information(INFO_REQUEST request, void *data, OBJECTID
        objectID = 0);
    int ListIndex(void);
    UI_ELEMENT *Next(void);
    UI_ELEMENT *Previous(void);

protected:
    // Members described in UI_ELEMENT reference chapter.
    UI_ELEMENT *previous, *next;

    UI_ELEMENT(void);
};
```

- *previous* and *next* are pointers to additional elements (or derived class elements) stored in doubly-linked lists.

UI_ELEMENT::UI_ELEMENT

Syntax

#include <ui_gen.hpp>

UI_ELEMENT(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced constructor returns a pointer to a new UI_ELEMENT object. The UI_ELEMENT class is of little use by itself; rather, Zinc Application Framework uses the element constructor to initialize specific class objects.

Example 1

```
#include <ui_evt.hpp>

UI_DEVICE::UI_DEVICE(RAW_EVENT _type, DEVICE_STATE _state) : UI_ELEMENT(),
    installed(FALSE), enabled(TRUE), type(_type), state(_state),
    display(NULL), eventManager(NULL)
{
    .
    .
    .

}
```

Example 2

```
#include <ui_win.hpp>

UI_WINDOW_OBJECT::UI_WINDOW_OBJECT(int left, int top, int width, int height,
    WOF_FLAGS _woFlags, WOAF_FLAGS _woAdvancedFlags) : UI_ELEMENT(),
    woFlags(_woFlags), woAdvancedFlags(_woAdvancedFlags)
{
    .
    .
    .

}
```

UI_ELEMENT::~UI_ELEMENT

Syntax

#include <ui_gen.hpp>

virtual ~UI_ELEMENT(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_ELEMENT object. The destructor is declared virtual so that derived list element destructors can be called. (If the destructor for the UI_ELEMENT class were not declared virtual, the programmer would need to call the destroy function associated with each derived class.)

Example

```
#include <ui_gen.hpp>

ElementFunction()
{
    UI_ELEMENT element1;
    UI_ELEMENT *element2;
    .
    .
    .
    // The element1 destructor is automatically called when the function ends.
    delete element2;
}
```

UI_ELEMENT::Information

Syntax

#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST *request*, void **data*,
    OBJECTID *objectID* = 0);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is a stub. Since all objects derived from UI_ELEMENT use a virtual **Information( )** function, this stub was necessary. This function simply returns NULL.

UI_ELEMENT::ListIndex

Syntax

```
#include <ui_win.hpp>

int ListIndex(void);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns an integer corresponding to the position of the element in its parent UI_LIST.

UI_ELEMENT::Next

Syntax

```
#include <ui_win.hpp>

UI_ELEMENT *Next(void);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a pointer to the next element, if one exists, in the list of elements.

NOTE: The **Next( )** function is also used by window objects and input devices. In each case, the function is overloaded to return an object pointer typecast according to the context. For example, window objects generally return a UI_WINDOW_OBJECT pointer when **Next( )** is called:

```
UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE("Window 1")
    + new UIW_TEXT(0, 0, 0, 0, TXF_NO_FLAGS, WOF_NON_FIELD_REGION);

for (UI_WINDOW_OBJECT *object = window->First(); object;
    object = object->Next())
    .
    .
    .
```

Input devices, however, return a UI_DEVICE pointer:

```
UI_DISPLAY *display = new UI_TEXT_DISPLAY;
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
*eventManager
    + new UID_KEYBOARD
    + new UID_MOUSE
    + new UID_CURSOR;

for (UI_DEVICE *device = eventManager->First(); device;
    device = device->Next());
    .
    .
    .
```

Some other class objects return specific element pointers (e.g., UI_QUEUE_ELEMENT, UI_REGION_ELEMENT, UIW_POP_UP_ITEM). Refer to each class definition for information about the return value of **Next( )**.

UI_ELEMENT::Previous

Syntax

#include <ui_win.hpp>

UI_ELEMENT *Previous(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a pointer to the previous element, if one exists, in the list of elements.

NOTE: The **Previous( )** function is also used by window objects and input devices. In each case, the function is overloaded to return an object pointer typecast according to the context. For example, window objects generally return a UI_WINDOW_OBJECT pointer when **Previous( )** is called:

```
UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
*window
    + new UIW_BORDER
    + new UIW_MAXIMIZE_BUTTON
    + new UIW_MINIMIZE_BUTTON
    + new UIW_SYSTEM_BUTTON
    + new UIW_TITLE("Window 1")
    + new UIW_TEXT(0, 0, 0, 0, TXF_NO_FLAGS, WOF_NON_FIELD_REGION);

for (UI_WINDOW_OBJECT *object = window->Last(); object;
    object = object->Previous())
    .
    .
    .
```

Input devices, however, return a UI_DEVICE pointer:

```
UI_DISPLAY *display = new UI_TEXT_DISPLAY;
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
*eventManager
    + new UID_KEYBOARD
    + new UID_MOUSE
    + new UID_CURSOR;

for (UI_DEVICE *device = eventManager->Last(); device;
    device = device->Previous());
```

.
.
.

Some other class objects return specific element pointers (e.g., UI_QUEUE_ELEMENT, UI_REGION_ELEMENT, UIW_POP_UP_ITEM). Refer to each class definition for information about the return value of **Previous( )**.

CHAPTER 8 – UI_ERROR_SYSTEM

The UI_ERROR_SYSTEM class implements an error system class that Zinc Application Framework and programmers use to report run-time errors. If the environment where the application is running (e.g., Windows) has a native error system, then UI_ERROR_-SYSTEM will call that error system. It is the default error system if no other error system is specified.

The UI_ERROR_SYSTEM class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UI_ERROR_SYSTEM : public UIW_WINDOW
{
public:
    // Members described in UI_ERROR_SYSTEM reference chapter.
    static char **errorPrompts;
    static char **statusPrompts;

    UI_ERROR_SYSTEM(void);
    virtual ~UI_ERROR_SYSTEM(void);
    static void Beep(void);
    virtual UIS_STATUS ReportError(UI_WINDOW_MANAGER *windowManager,
        UIS_STATUS errorStatus, char *format, ...);
};
```

- *errorPrompts* is a pointer to an array of text strings that will be placed as options in the error window's system button pop-up menu. The array is defined and assigned to this static member in the **G_PERROR.CPP** module. Placing these literal strings in a lookup table allows the programmer to easily change the text that will be displayed (e.g., the text could be changed to a language other than English).

- *statusPrompts* is a pointer to an array of text strings that will be placed on buttons on the error window. The array is defined and assigned to this static member in the **G_PERROR.CPP** module. Placing these literal strings in a lookup table allows the programmer to easily change the text that will be displayed (e.g., the text could be changed to a language other than English).

UI_ERROR_SYSTEM::UI_ERROR_SYSTEM

Syntax

#include <ui_win.hpp>

UI_ERROR_SYSTEM(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UI_ERROR_SYSTEM class object.

Example

```
#include <ui_win.hpp>

main()
{
    .
    .
    .

    // Install the error system.
    UI_ERROR_SYSTEM *errorSystem = new UI_ERROR_SYSTEM();
    UI_WINDOW_OBJECT::errorSystem = errorSystem;
    .
    .
    .

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

UI_ERROR_SYSTEM::~UI_ERROR_SYSTEM

Syntax

#include <ui_win.hpp>

virtual ~UI_ERROR_SYSTEM(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_ERROR_-SYSTEM object.

Example

```
#include <ui_win.hpp>

main()
{
    .
    .
    .

    // Install the error system.
    UI_ERROR_SYSTEM *errorSystem = new UI_ERROR_SYSTEM();
    UI_WINDOW_OBJECT::errorSystem = errorSystem;
    .
    .
    .

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

UI_ERROR_SYSTEM::Beep

Syntax

#include <ui_win.hpp>

static void Beep(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function sounds the computer's speaker indicating to the user that an error has occurred.

Example

```
#include <ui_win.hpp>

UIS_STATUS UI_ERROR_SYSTEM::ReportError(UI_WINDOW_MANAGER *_windowManager,
        UIS_STATUS errorStatus, char *format, ...)
{
    // Beep then display the error.
    Beep();
    .
    .
    .

}
```

UI_ERROR_SYSTEM::ReportError

Syntax

#include <ui_win.h>

virtual UIS_STATUS ReportError(const UI_WINDOW_MANAGER **windowManager*,
UIS_STATUS *errorStatus*, const char **format*[, *argument*, ...]);

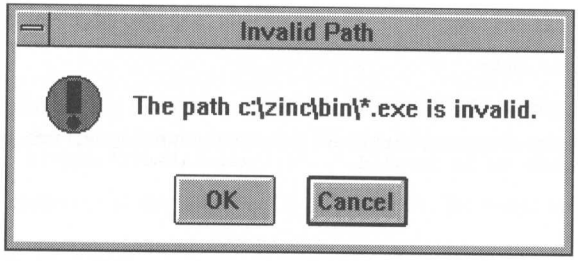
Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function is used to report an error via the error system. It does this by sounding a single beep and displaying a window with the error message, *format*, when the error occurs. This function is declared virtual so that additional error systems can be written to override this function. The figure below shows the graphic UI_ERROR_-SYSTEM presentation window:



- *returnValue*$_{out}$ is WOS_INVALID if the "OK" button is pressed or WOS_NO_-STATUS if the "Cancel" button is pressed.

- *windowManager*$_{in}$ is a pointer to the Window Manager.

- *errorStatus*$_{in}$ indicates the type of action to take place when the error occurs. *errorStatus* can be set to one of the following:

  WOS_INVALID—If this status is set, the error window will contain text information and an "OK" and "CANCEL" buttons. Selecting "OK" causes the error window to be deleted and the field's value to be restored to the value it contained before the invalid entry was made. Pressing the "CANCEL" button causes the error window to be deleted and the invalid field entry to remain.

  WOS_NO_STATUS—If this status is set, the error window will contain text information and an "OK" button. Pressing the "OK" button causes the error window to be deleted and the field's value to be restored to the value it contained before the invalid entry was made.

- *format*$_{in}$ is the **printf** style format that controls how the string is to be converted.

- *argument*$_{in}$ and ... are the **printf** style arguments that are used by the *format* string.

NOTE: The **ReportError( )** function looks for an icon called "ASTERISK." This icon must be in the *UI_WINDOW_OBJECT::defaultStorage* **.DAT** file or linked in as a resource (if applicable to the environment) in order for the error icon to be displayed. This icon can be copied from the **P_DESIGN.ZNC** file for the Designer.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_OBJECT *item)
{
    item->errorSystem->ReportError(item->windowManager, WOS_NO_STATUS,
        "The path %s is invalid.", "c:\\zinc\\bin\\*.exe");
    .
    .
    .
}
```

CHAPTER 9 – UI_EVENT

The UI_EVENT structure is used to store all information passed through Zinc Application Framework (from the Event Manager to the Window Manager).

The UI_EVENT structure is declared in **UI_EVT.HPP**. Its fields are shown below:

```
struct EXPORT UI_EVENT
{
    // Members described in UI_EVENT reference chapter.
    EVENT_TYPE type;
    RAW_CODE rawCode;
    RAW_CODE modifiers;
#if defined(ZIL_MSWINDOWS)
    MSG message;
#elif defined(ZIL_OS2)
    QMSG message;
#elif defined(ZIL_MOTIF)
    XEvent message;
#endif

    union
    {
        UI_KEY key;
        UI_REGION region;
        UI_POSITION position;
        UI_SCROLL_INFORMATION scroll;
        void *data;
    };

    UI_EVENT(void);
    UI_EVENT(EVENT_TYPE _type, RAW_CODE _rawCode = 0);
    UI_EVENT(EVENT_TYPE _type, RAW_CODE _rawCode, const UI_KEY &_key);
    UI_EVENT(EVENT_TYPE _type, RAW_CODE _rawCode, const UI_REGION &_region);
    UI_EVENT(EVENT_TYPE _type, RAW_CODE _rawCode,
        const UI_POSITION &_position);
    UI_EVENT(EVENT_TYPE _type, RAW_CODE _rawCode,
        const UI_SCROLL_INFORMATION &_scroll);
#if defined(ZIL_MSWINDOWS) && defined(WIN32)
    UI_EVENT(EVENT_TYPE type, HWND hWnd, UINT wMsg, WPARAM wParam,
        LPARAM lParam);
#elif defined(ZIL_MSWINDOWS)
#if defined(__BCPLUSPLUS__) | defined(__TCPLUSPLUS__)
#if __BORLANDC__ >= 0x0410      // Set by Borland C++ version 3.1
    UI_EVENT(EVENT_TYPE type, HWND hWnd, UINT wMsg, UINT wParam,
        LONG lParam);
#else
    UI_EVENT(EVENT_TYPE type, HWND hWnd, WORD wMsg, WORD wParam,
        LONG lParam);
#endif
#else
    UI_EVENT(EVENT_TYPE type, HWND hWnd, WORD wMsg, WORD wParam,
        LONG lParam);
#endif
#elif defined(ZIL_OS2)
    UI_EVENT(EVENT_TYPE type, HWND hWnd, ULONG msg, MPARAM mp1, MPARAM mp2);
#elif defined(ZIL_MOTIF)
    UI_EVENT(EVENT_TYPE _type, XEvent &xevent);
#endif
};
```

- *type* is the type of event. Events are numbered as follows:

  -32,767 to -1,000—Reserved by Zinc Application Framework for future use.

  -999 to -1—Reserved by Zinc Application Framework for system messages. System messages are declared in **UI_EVT.HPP**. A full description of these messages is given in "Appendix B–System Events."

  0 to 99—Reserved for raw device identifications. The following constants (declared in **UI_EVT.HPP**) are pre-defined:

    E_CURSOR(50)—Identification for the UID_CURSOR class.

    E_DEVICE(99)—Identification used to define a generic device.

    E_KEY(10)—Identification for the UID_KEYBOARD class.

    E_MOTIF(3)—Identification for Motif events.

    E_MOUSE(30)—Identification for the UID_MOUSE class.

    E_MSWINDOWS(1)—Identification for MS Windows events.

    E_OS2(2)—Identification for OS/2 events.

  The following additional raw device identifications are reserved by Zinc Application Framework for future use: 4, 11-19, 31-39, 51-59, 70-79 and 90-98. The remaining values 5-9, 20-29, 40-49, 60-69 and 80-89 can be used by the programmer.

  100 to 9,999—Reserved by Zinc Application Framework for logical events. Logical messages are declared in **UI_EVT.HPP**. A full description of these messages is given in "Appendix C–Logical Events."

  10,000 to 32,767—Available to the programmer for private use. These values are not used by Zinc Application Framework.

- *rawCode* is the raw code value associated with the event. The following devices (declared in **UI_EVT.HPP**) use the *rawCode* event field:

  UID_KEYBOARD—The *rawCode* for the keyboard device is the raw scan code associated with the key. For example, pressing <F1> generates a raw scan code

of 0x3B00. In this case, the UI_EVENT structure would contain the following values:

```
event.type = E_KEY;
event.rawCode = 0x3B00;
event.key.value = 0;                    // low 8 bits of rawCode
event.key.shiftState = 0;
```

UID_MOUSE—The *rawCode* for the mouse device is the keyboard shift state (low 8 bits) and the mouse button state (high 8 bits). For example, pressing the left mouse button while holding the <Left-shift> key generates a raw code of 0x0102 (0x0002 for the <Left-shift> key and 0x0100 for the left-mouse button). In this case, the UI_EVENT structure would contain the following values:

```
event.type = E_MOUSE;
event.rawCode = 0x0102;
event.position.column = <current mouse column position>;
event.position.line = <current mouse row position>;
```

UID_PENDOS—The *rawCode* for the PenDOS device is the same as that for the mouse device.

- *modifiers* is a bit field that indicates which modifier keys (i.e., shift keys, meta keys, etc.) were pressed at the time the event occurred.

- *message* is the message received if the application is running in Windows, OS/2 or Motif.

- *key*, *region*, *position*, *scroll* and *data* are types of specific information associated with the event.

UI_EVENT::UI_EVENT

Syntax

UI_EVENT(void);
or
UI_EVENT(EVENT_TYPE *_type*, RAW_CODE *_rawCode* = 0);
or
UI_EVENT(EVENT_TYPE *_type*, RAW_CODE *_rawCode*, const UI_KEY &*_key*);
or

UI_EVENT(EVENT_TYPE *_type*, RAW_CODE *_rawCode*,
const UI_REGION &*_region*);
or
UI_EVENT(EVENT_TYPE *_type*, RAW_CODE *_rawCode*,
const UI_POSITION &*_position*);
or
UI_EVENT(EVENT_TYPE *_type*, RAW_CODE *_rawCode*,
const UI_SCROLL_INFORMATION &*_scroll*);
or
UI_EVENT(EVENT_TYPE *type*, HWND *hWnd*, UINT *wMsg*, WPARAM *wParam*,
LPARAM *lParam*);
or
UI_EVENT(EVENT_TYPE *type*, HWND *hWnd*, WORD *wMsg*, WORD *wParam*,
LONG *lParam*);
or
UI_EVENT(EVENT_TYPE *type*, HWND *hWnd*, UINT *wMsg*, UINT *wParam*,
LONG *lParam*);
or
UI_EVENT(EVENT_TYPE *type*, HWND *hWnd*, ULONG *msg*, MPARAM *mp1*,
MPARAM *mp2*);
or
UI_EVENT(EVENT_TYPE *_type*, XEvent &*xevent*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first overloaded constructor takes no arguments. It creates a basic event structure with no special initialization.

The second overloaded constructor initializes an event structure with the following arguments:

- $\_type_{in}$ contains a valid EVENT_TYPE.
- $\_rawCode_{in}$ contains a valid RAW_CODE.

The third overloaded constructor initializes an event structure with the following arguments:

- $\_type_{in}$ contains a valid EVENT_TYPE.
- $\_rawCode_{in}$ contains a valid RAW_CODE.
- $\_key_{in}$ contains the address of a UI_KEY structure.

The fourth overloaded constructor initializes an event structure with the following arguments:

- $\_type_{in}$ contains a valid EVENT_TYPE.
- $\_rawCode_{in}$ contains a valid RAW_CODE.
- $\_region_{in}$ contains the constant address of a UI_REGION structure.

The fifth overloaded constructor initializes an event structure with the following arguments:

- $\_type_{in}$ contains a valid EVENT_TYPE.
- $\_rawCode_{in}$ contains a valid RAW_CODE.
- $\_position_{in}$ contains the constant address of a UI_POSITION structure.

The sixth overloaded constructor initializes an event structure with the following arguments:

- $\_type_{in}$ contains a valid EVENT_TYPE.
- $\_rawCode_{in}$ contains a valid RAW_CODE.
- $\_scroll_{in}$ contains the constant address of a UI_SCROLL_INFORMATION structure.

The seventh, eighth and ninth overloaded constructors are used only for Windows programming. They initialize an event structure with the following arguments:

- $\_type_{in}$ contains a valid EVENT_TYPE.
- $hWnd_{in}$ contains a handle to a window.

- *wMsg*$_{in}$ contains an input message.
- *wParam*$_{in}$ is a WORD (or a UINT for Borland C++ version 3.1 and Microsoft C/C++ 7.0, or WPARAM for WIN32) value containing specific message information.
- *lParam*$_{in}$ is a LONG (or LPARAM for WIN32) value containing specific message information.

The tenth overloaded constructor is used only for OS/2 programming. It initializes an event structure with the following arguments:

- *_type*$_{in}$ contains a valid EVENT_TYPE.
- *hWnd*$_{in}$ contains a handle to a window.
- *msg*$_{in}$ contains an input message.
- *mp1*$_{in}$ is a MPARAM value containing specific message information.
- *mp2*$_{in}$ is a MPARAM value containing specific message information.

The eleventh overloaded constructor is used only for Motif programming. It initializes an event structure with the following arguments:

- *_type*$_{in}$ contains a valid EVENT_TYPE.
- *xevent*$_{in}$ is a XEvent value containing specific message information.

Example 1

```
#include <ui_win.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANAGER(display,
        eventManager);
    .
    .
    .
    EVENT_TYPE ccode;
```

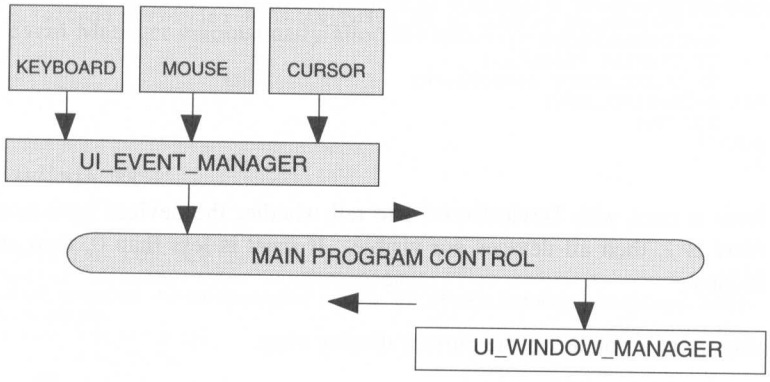
```
    do
    {
        // Get an event from the event manager.
        UI_EVENT event;
        eventManager->Get(event, Q_NORMAL);

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

}

static void Exit(UI_WINDOW_OBJECT *item, UI_EVENT &event, EVENT_TYPE ccode)
{
    // Send an L_EXIT message through the system.
    event.type = L_EXIT;
    UI_EVENT_MANAGER *eventManager = ((UIW_POP_UP_ITEM *)item)->eventManager;
    eventManager->Put(event, Q_BEGIN);
}
```

Example 2

```
void UI_WINDOW_MANAGER::Add(UI_WINDOW_OBJECT *object)
{
    .
    .
    .

    // Flag the old object as non-current.
    if (firstObject && object != firstObject)
        firstObject->Event(UI_EVENT(S_NON_CURRENT));
    .
    .
    .

    // Flag the new object as current.
    if (object != firstObject && UI_LIST::Index(object) != -1)
    {
        UI_LIST::Subtract(object);
        UI_LIST::Add(firstObject, object);
    }
    else if (object != firstObject)
    {
        UI_LIST::Add(firstObject, object);
        object->Event(UI_EVENT(S_INITIALIZE));
        object->Event(UI_EVENT(S_CREATE));
    }
    display->RegionDefine(object->screenID, object->true);
    .
    .
    .

}
```

CHAPTER 10 – UI_EVENT_MANAGER

The UI_EVENT_MANAGER class serves as the control unit for input devices and as the storage unit for event information that is processed by Zinc Application Framework 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:



The controlling portion of the UI_EVENT_MANAGER class contains a list of input devices that are either polled by the Event Manager (e.g., a keyboard device) or are automatically interrupted by the device's interrupt service routine (e.g., a mouse device).

The storage portion of the UI_EVENT_MANAGER class is implemented as an array of UI_EVENT structures. The size of this array is specified by the programmer when the Event Manager class is constructed. Input devices feed to the Event Manager when they are polled or when their interrupt routine is activated.

The UI_EVENT_MANAGER class is declared in **UI_EVT.HPP**. Its public and protected members are:

```
class EXPORT UI_EVENT_MANAGER : public UI_LIST
{
public:
    // Members described in UI_EVENT_MANAGER reference chapter.
    UI_EVENT_MANAGER(UI_DISPLAY *display, int noOfElements = 100);
    virtual ~UI_EVENT_MANAGER(void);
    EVENT_TYPE DevicePosition(DEVICE_TYPE deviceType, int column, int line);
    EVENT_TYPE DeviceState(DEVICE_TYPE deviceType,
        DEVICE_STATE deviceState);
    EVENT_TYPE DeviceImage(DEVICE_TYPE deviceType,
        DEVICE_IMAGE deviceImage);
```

```
    virtual EVENT_TYPE Event(const UI_EVENT &event,
        DEVICE_TYPE deviceType = E_DEVICE);
    virtual int Get(UI_EVENT &event, Q_FLAGS flags = Q_NORMAL);
    virtual void Put(const UI_EVENT &event, Q_FLAGS flags = Q_END);

    // Members described in UI_LIST reference chapter.
    void Add(UI_DEVICE *device);
    UI_DEVICE *Current(void);
    UI_DEVICE *First(void);
    UI_DEVICE *Last(void);
    void Subtract(UI_DEVICE *device);
    UI_EVENT_MANAGER &operator+(UI_DEVICE *device);
    UI_EVENT_MANAGER &operator-(UI_DEVICE *device);

protected:
    // Members described in UI_EVENT_MANAGER reference chapter.
    int level;
    UI_DISPLAY *display;
    UI_QUEUE_BLOCK queueBlock;
#if defined(ZIL_OS2)
    HMQ hmq;
#endif
};
```

- *level* is used with **DeviceState( )** to tell whether the devices have been hidden. If *level* is 1, then all devices are visible. If *level* is less than 0, then all devices are hidden.

- *display* is a pointer to the current display class.

- *queueBlock* is a pointer to the event queue, which contains all of the messages sent by the devices as well as all other communication within Zinc Application Framework.

- *hmq* is a pointer to the OS/2 message queue.

UI_EVENT_MANAGER::UI_EVENT_MANAGER

Syntax

#include <ui_evt.hpp>

UI_EVENT_MANAGER(UI_DISPLAY **display*, int *noOfElements* = 100);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UI_EVENT_MANAGER class object. It must be called after the display class constructor has been called.

- *display*$_{in}$ is a pointer to the screen display. This pointer is used by input devices when they display their information to the screen display (e.g., the blinking cursor of the UID_CURSOR class object).

- *noOfElements*$_{in}$ tells the maximum number of elements to reserve in the event queue. The Event Manager automatically allocates space for *noOfElements*.

Example

```
#include <ui_win.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display, 100);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANAGER(display,
        eventManager);
    .
    .
    .

    // Restore the system.  We must explicitly call the destructor for the
    // window manager and event manager to preserve the creation order.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

UI_EVENT_MANAGER::~UI_EVENT_MANAGER

Syntax

#include <ui_evt.hpp>

virtual ~UI_EVENT_MANAGER(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_EVENT_-MANAGER object and destroys the class information of any input device that remains attached to the Event Manager.

Example

```
#include <ui_win.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANAGER(display,
        eventManager);
    .
    .
    .
    // Restore the system.  We must explicitly call the destructor for the
    // window manager and event manager to preserve the creation order.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

UI_EVENT_MANAGER::DeviceImage

Syntax

#include <ui_evt.hpp>

EVENT_TYPE DeviceImage(DEVICE_TYPE *deviceType*,
    DEVICE_IMAGE *deviceImage*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function sends a programmer specified state message to all input devices that match *deviceType*.

- *returnValue*$_{out}$ is the new state of the device (i.e., event.type.)

- *deviceType*$_{in}$ is the device identification where the state message is to be sent. The following device types (declared in **UI_EVT.HPP**) may be specified:

  E_CURSOR—Sends the state information to the UID_CURSOR class object (if it is in the device list).

  E_DEVICE—Sends the state information to all input devices in the Event Manager's device list.

  E_MOUSE—Sends the state information to the UID_MOUSE class object (if it is in the device list.)

- *deviceImage*$_{in}$ is the new image of the device. For mouse and cursor devices, the allowable image changes (declared in **UI_EVT.HPP**) are:

  E_CURSOR—The UID_CURSOR class recognizes the following image information:

  DC_INSERT—Changes the cursor to an insert cursor (e.g., a thick vertical bar).

  DC_OVERSTRIKE—Changes the cursor to an overstrike cursor (e.g., a thin vertical bar).

  E_MOUSE—These must be recognized by *deviceType*. For example, the UID_MOUSE class recognizes the following image information:

  DM_EDIT—Displays a '|' mouse pointer.

DM_DIAGONAL_ULLR—Displays a upper-left to lower-right diagonal mouse pointer.

DM_DIAGONAL_LLUR—Displays a lower-left to upper-right diagonal mouse pointer.

DM_MOVE—Displays a four-way arrow mouse pointer.

DM_HORIZONTAL—Displays a '↔' mouse pointer.

DM_VERTICAL—Displays a '↕' mouse pointer.

DM_VIEW—Displays a normal '↖'mouse pointer.

DM_WAIT—Displays an hour-glass mouse pointer.

DM_POSITION—Displays a cross-hair mouse pointer.

Example

```
#include <ui_evt.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;

    // Display an hour glass until the program is ready to receive user input.
    eventManager->DeviceImage(E_MOUSE, DM_WAIT);
    .
    .
    .

}
```

UI_EVENT_MANAGER::DevicePosition

Syntax

#include <ui_evt.hpp>

void DevicePosition(DEVICE_TYPE *deviceType*, int *column*, int *line*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function sets the position for a receiving device (e.g., mouse or cursor).

- *deviceType*$_{in}$ is the type of device (e.g., E_CURSOR or E_MOUSE) for which the message is intended.

- *column*$_{in}$ and *line*$_{in}$ is the position to where the device will be moved. The value of *column* and *line* depends on the type of display mode in which the application is running. For example, if the cursor is to be positioned at the center of the screen while the application is running in text mode (i.e., an 80 column by 25 line screen) the position values should be:

```
column = 40;
line = 13;
eventManager->DevicePosition(E_CURSOR, column, line);
```

If on the other hand, the application is running in a 640 column by 480 line graphics mode, the position values should be:

```
column = 320;
line = 240;
eventManager->DevicePosition(E_CURSOR, column, line);
```

Example

```
#include <ui_evt.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    .
    .
    .
```

```
    // Reposition the cursor at the top-left side of the screen.
    eventManager->DevicePosition(E_CURSOR, 0, 0);
    .
    .
    .
}
```

UI_EVENT_MANAGER::DeviceState

Syntax

#include <ui_evt.hpp>

EVENT_TYPE DeviceState(DEVICE_TYPE *deviceType*, DEVICE_STATE *deviceState*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function sends a programmer specified state message to all input devices that match *deviceType*.

- $returnValue_{out}$ is the new state of the device.
- $deviceType_{in}$ is the device identification where the state message is to be sent. The following device types (declared in **UI_EVT.HPP**) may be specified:

  E_CURSOR—Sends the state information to the UID_CURSOR class object (if it is in the device list).

  E_DEVICE—Sends the state information to all input devices in the Event Manager's device list.

  E_MOUSE—Sends the state information to the UID_MOUSE class object (if it is in the device list).

  E_KEY—Sends the state information to the UID_KEYBOARD class object (if it is in the device list).

- *deviceState*$_{in}$ is the new state of the device. Allowable state changes (declared in **UI_EVT.HPP**) are:

  D_HIDE—Hides the specified device's image. If *deviceType* is E_DEVICE, all devices in the Event Manager's device list are sent the D_HIDE message.

  D_ON—Turns the specified device on. If *deviceType* is E_DEVICE, all devices in the Event Manager's device list are sent the D_ON message.

  D_OFF—Turns the specified device off. If *deviceType* is E_DEVICE, all devices in the Event Manager's device list are sent the D_OFF message.

  D_STATE—Gets the state information associated with the specified device. If *deviceType* is E_DEVICE, only the state of the last device in the Event Manager's device list is returned.

  Other device states—These must be recognized by the device whose type is *deviceType*. For example, the UID_MOUSE class also recognizes the following state information:

  DM_EDIT—Displays a '|' mouse pointer.

  DM_DIAGONAL_ULLR—Displays a upper-left diagonal mouse pointer.

  DM_DIAGONAL_LLUR—Displays a lower-left diagonal mouse pointer.

  DM_MOVE—Displays a four-way arrow mouse pointer.

  DM_HORIZONTAL—Displays a '↔' mouse pointer.

  DM_VERTICAL—Displays a '↕' mouse pointer.

  DM_VIEW—Displays a normal '↖'mouse pointer.

  DM_WAIT—Displays an hour-glass mouse pointer.

Example

```
#include <ui_evt.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
```

```
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;

    // Display an hour glass until the program is ready to receive user input.
    eventManager->DeviceState(E_MOUSE, DM_WAIT);
    .
    .
    .
}
```

UI_EVENT_MANAGER::Event

Syntax

#include <ui_evt.hpp>

virtual EVENT_TYPE Event(const UI_EVENT *&event*,
DEVICE_TYPE *deviceType* = E_DEVICE);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual function allows the programmer to communicate with devices through the Event Manager. This permits the programmer to change the interaction of input devices without having a pointer to the device.

- $event_{in}$ is the message to be passed to an input device.
- $deviceType_{in}$ is the type of device to which the message will be passed. DEVICE_TYPE values used by the library include: E_CURSOR, E_KEY and E_MOUSE.

Example 1

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    // Attach the keyboard to the event manager.
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    .
    .
    .

    // Turn all the devices off.
    UI_EVENT event;
    event.type = E_DEVICE;
    event.rawCode = D_OFF;
    eventManager->Event(event, E_DEVICE);
    .
    .
    .

}
```

Example 2

```
#include <ui_win.hpp>

EVENT_TYPE UIW_TITLE::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    EVENT_TYPE ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_TITLE);
    switch (ccode)
    {
    .
    .
    .

    case L_VIEW:
        if (!FlagSet(parent->woAdvancedFlags, WOAF_NO_MOVE) &&
            UI_WINDOW_OBJECT::Overlap(event.position))
        {
            // Setting tEvent equal to event sends the message to same device
            // as where the message originated.
            UI_EVENT tEvent = event;
            tEvent.rawCode = DM_MOVE;
            eventManager->Event(event, E_DEVICE);
            break;
        }
        // Continue to default.

    default:
        ccode = UIW_BUTTON::Event(event);
        break;
    }

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

UI_EVENT_MANAGER::Get

Syntax

#include <ui_evt.hpp>

virtual int Get(UI_EVENT &*event*, Q_FLAGS *flags* = Q_NORMAL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function gets an event from the Event Manager's input queue, if one is available. (When input devices receive information, they feed the information to the Event Manager. The Event Manager in turn dispatches the information when the **UI_EVENT_-MANAGER::Get( )** function is called.)

- $returnValue_{out}$ is set to 0 if an event was available and copied to the *event* argument. Otherwise, a negative value is returned, indicating that an event was not available for the type of request made.

- $event_{in/out}$ is a reference pointer to the event. This argument is a copy of the event information.

- $flags_{in}$ indicates the type of read operation to perform with the devices in the Event Manager's list of devices. The following flags (declared in **UI.EVT_HPP**) specify the read operation:

  Q_BEGIN—Retrieves the event from the beginning of the input queue. Setting this flag forces the Event Manager to return the oldest event in the event queue.

  Q_BLOCK—Remains in the **UI_EVENT_MANAGER::Get( )** function polling the devices until there is an event on the queue.

  Q_DESTROY—Destroys the event information from the Event Manager after it is copied to *event*. **NOTE:** The Q_NO_DESTROY flag takes precedence over this flag.

Q_END—Retrieves the event from the end of the input queue. Setting this flag forces the Event Manager to return the most recent event in the event queue.

Q_NO_BLOCK—Polls the devices and then immediately returns from the **UI_EVENT_MANAGER::Get( )** function, even if there is not an event in the event queue.

Q_NO_DESTROY—Does not destroy the event information from the input queue. If this flag is set, the next call to **UI_EVENT_MANAGER::Get( )** will return the same event.

Q_NO_POLL—Does not poll the devices before checking the event queue. This is an advanced flag that should only be used by UI_DEVICE class objects when they communicate with the Event Manager. It prevents UI_DEVICE class objects from being recursively called by the **UI_EVENT_MANAGER::Get( )** routine.

Q_NORMAL—Performs a standard read operation. This flag is equivalent to setting the Q_BLOCK, Q_BEGIN, Q_DESTROY and Q_POLL flags.

Q_POLL—Ensures that all devices in the Event Manager's device list are called before information is retrieved from the event queue. This enables the device objects (e.g., the keyboard and cursor) to perform any polling operations (e.g., BIOS calls for the keyboard device) before the event queue is examined.

Example

```
#include <ui_win.hpp>

main()
{
    .
    .
    .

    EVENT_TYPE ccode;
    do
    {
        // Get an event from the event manager.
        UI_EVENT event;
        eventManager->Get(event, Q_NORMAL);

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

}
```

UI_EVENT_MANAGER::Put

Syntax

#include <ui_evt.hpp>

virtual void Put(const UI_EVENT *&event*, Q_FLAGS *flags* = Q_END);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine puts an event into the event queue.

- $event_{in}$ is a reference pointer to the event. This argument has the event information that is put in the input queue.

- $flags_{in}$ indicates the order in which to insert the event into the event queue. The following flags (declared in **UI_EVT.HPP**) are recognized by the **UI_EVENT_-MANAGER::Put( )** function:

  Q_BEGIN—Puts the event information at the beginning of the input queue (i.e., before the oldest event in the input queue.)

  Q_END—Puts the event information at the end of the input queue (i.e., after the most recent event in the input queue.)

Example

```
#include <ui_win.hpp>

static void Exit(UI_WINDOW_OBJECT *item, UI_EVENT &event, EVENT_TYPE ccode)
{
    // Send an L_EXIT message through the system.
    event.type = L_EXIT;
    UI_EVENT_MANAGER *eventManager = item->eventManager;
    eventManager->Put(event, Q_BEGIN);
}
```

CHAPTER 11 – UI_EVENT_MAP

The UI_EVENT_MAP structure is used to map raw input device events to logical events. For example, Zinc Application Framework declares default event mapping for the UID_-KEYBOARD and UID_MOUSE (including the UID_PENDOS device) class objects. Some of their mapped values are:

<F1> — Mapped to L_HELP; a message that causes the system to generate context-sensitive help information about the current window object.

<Ctrl C> — Mapped to L_EXIT_FUNCTION; a message that causes program execution to end.

<Ctrl F5> — Mapped to L_BEGIN_MARK for editable window objects (e.g., UIW_DATE, UIW_STRING). This key allows end-users to begin marked regions that can be cut or copied for later use.

<Left-mouse-button drag> — Mapped to L_CONTINUE_MARK for editable objects. This is equivalent to the <Ctrl F5> key that begins a marked region.

<Left-mouse-button click> — Mapped to L_BEGIN_SELECT; an option that selects a new window field.

The UI_EVENT_MAP structure is declared in **UI_WIN.HPP**. Its fields are:

```
struct EXPORT UI_EVENT_MAP
{
    // Members described in UI_EVENT_MAP reference chapter.
    OBJECTID objectID;
    LOGICAL_EVENT logicalValue;
    EVENT_TYPE eventType;
    RAW_CODE rawCode;
    RAW_CODE modifiers;

    static LOGICAL_EVENT MapEvent(UI_EVENT_MAP *mapTable,
        const UI_EVENT &event,
        OBJECTID id1 = ID_WINDOW_OBJECT, OBJECTID id2 = ID_WINDOW_OBJECT,
        OBJECTID id3 = ID_WINDOW_OBJECT, OBJECTID id4 = ID_WINDOW_OBJECT,
        OBJECTID id5 = ID_WINDOW_OBJECT);
};
```

- *objectID* is the object identification for which the match applies. (A full list of object identifications is given in **UI_EVT.HPP**.) Each window identification has an "ID_" prefix. Some example window object identifications are:

ID_WINDOW_OBJECT—This identification is a default identification associated with all class objects derived from the UI_WINDOW_OBJECT base class.

ID_BORDER—This identification is associated with the UIW_BORDER class object.

ID_STRING—This identification is associated with the UIW_STRING object or with any class object derived from the UIW_STRING base class (e.g., UIW_DATE, UIW_TIME).

- *logicalValue* is the logical event to map. (A full list of logical values is given in **UI_EVT.HPP**.) Each logical value has an "L_" prefix. Some example logical values are:

  L_EXIT—Exits the application program.

  L_BEGIN_MARK—Begins a mark region. This logical event is understood by all editable window objects (e.g, UIW_STRING, UIW_FORMATTED_STRING, UIW_TEXT).

- *eventType* is the raw device identification. The following event types (declared in **UI_EVT.HPP**) are pre-defined by Zinc Application Framework:

  E_CURSOR—Identification for the UID_CURSOR object.

  E_KEY—Identification for the UID_KEYBOARD object. This device generates keyboard input information.

  E_MOUSE—Identification for the UID_MOUSE object. This device generates mouse input information.

- *rawCode* is the raw scan code or button state (depending on the type of device) of the event.

UI_EVENT_MAP::MapEvent

Syntax

```
#include <ui_evt.hpp>

static LOGICAL_EVENT MapEvent(UI_EVENT_MAP *mapTable,
    const UI_EVENT &event,
    OBJECTID id1 = ID_WINDOW_OBJECT,
    OBJECTID id2 = ID_WINDOW_OBJECT,
    OBJECTID id3 = ID_WINDOW_OBJECT,
    OBJECTID id4 = ID_WINDOW_OBJECT,
    OBJECTID id5 = ID_WINDOW_OBJECT);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function provides the logical mapping (if any) of a raw event.

- *returnValue*$_{out}$ is the logical event that matches the event and identification parameters. If no match occurs, this value is *event.type* (the event type passed into the match function).

- *mapTable*$_{in}$ is a pointer to the event map table to be used by the event mapping function.

- *event*$_{in}$ is the raw event to be mapped. The *event.type* and *event.rawCode* values are used by the event mapping function.

- *id1*$_{in}$, *id2*$_{in}$, *id3*$_{in}$, *id4*$_{in}$ and *id5*$_{in}$ are hierarchal identification values to use while interpreting the raw event. For example, the UIW_TEXT class object uses the following identification values when it looks for a logical mapping:

  id1—ID_TEXT
  id2—ID_WINDOW
  id3—ID_WINDOW_OBJECT

id4—unused
id5—unused

Example 1

```
#include <ui_evt.hpp>

static UI_EVENT_MAP _myHotKeyMapTable[] =
{
    { ID_WINDOW_OBJECT, HOT_KEY_SYSTEM, E_KEY, ALT_PERIOD },
    { ID_WINDOW_OBJECT, HOT_KEY_SYSTEM, E_KEY, ALT_SPACE },
    { ID_WINDOW_OBJECT, HOT_KEY_MINIMIZE, E_KEY, ALT_WHITE_MINUS },
    { ID_WINDOW_OBJECT, HOT_KEY_MINIMIZE, E_KEY, ALT_GRAY_MINUS },
    { ID_WINDOW_OBJECT, HOT_KEY_MAXIMIZE, E_KEY, ALT_WHITE_PLUS },
    { ID_WINDOW_OBJECT, HOT_KEY_MAXIMIZE, E_KEY, ALT_GRAY_PLUS },
    { ID_END, 0, 0, 0 }
};

const int MY_EVENT = 10000;
static UI_EVENT_MAP _myEventMapTable[] =
{
    { MY_EVENT, L_EXIT, E_KEY, ALT_F10 },
    { MY_EVENT, L_EXIT, E_KEY, ALT_X },
    { ID_END, 0, 0, 0 }
};

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANAGER(display,
        eventManager);
    .
    .
    .
    // Simplify the global hot key table.
    eventManager->hotKeyTable = _myHotKeyMapTable;
    UI_EVENT event;
    do
    {
        // Get an event from the event manager.
        event.type = MapEvent(_myEventMapTable, event, MY_EVENT, MY_EVENT);
        if (event.type != L_EXIT)
            event.type = windowManager->Event(event);
    } while (event.type != L_EXIT);
    .
    .
    .
}
```

Example 2

```
#include <ui_evt.hpp>

class EXPORT UI_WINDOW_OBJECT : public UI_ELEMENT
{
protected:
    EVENT_TYPE LogicalEvent(const UI_EVENT &event, OBJECTID currentID)
        { return (UI_EVENT_MAP::MapEvent(eventMapTable, event, currentID,
            windowID[0], windowID[1], windowID[2], windowID[3], windowID[4])); }
    .
    .
    .

}

EVENT_TYPE UIW_TITLE::Event(const UI_EVENT &event)
{
    EVENT_TYPE ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_TITLE);
    switch (ccode)
    {
    .
    .
    .

    }
    return (ccode);
}
```

CHAPTER 12 – UI_FG_DISPLAY

The UI_FG_DISPLAY class implements a graphics display that uses the Zortech Flash Graphics library package to display information to the screen. Since the UI_FG_-DISPLAY class is derived from UI_DISPLAY, only details specific to the UI_FG_-DISPLAY class are given in this chapter. For descriptions and examples regarding virtual or inherited display members, see "Chapter 6—UI_DISPLAY."

The UI_FG_DISPLAY class is declared in **UI_DSP.HPP**. Its public and protected members are:

```
class EXPORT UI_FG_DISPLAY : public UI_DISPLAY, public UI_REGION_LIST
{
public:
    struct FGFONT
    {
        char *fontptr;
        char *fontptr2;
        int maxWidth, maxHeight;
    };
    typedef unsigned short FGPATTERN[16];

    static UI_PATH *searchPath;
    static FGFONT fontTable[MAX_LOGICAL_FONTS];
    static FGPATTERN patternTable[MAX_LOGICAL_PATTERNS];

    UI_FG_DISPLAY(int mode = 0);
    virtual ~UI_FG_DISPLAY(void);
    virtual void Bitmap(SCREENID screenID, int column, int line,
        int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette = NULL,
        const UI_REGION *clipRegion = NULL, HBITMAP *colorBitmap = NULL,
        HBITMAP *monoBitmap = NULL);
    virtual void BitmapArrayToHandle(SCREENID screenID, int bitmapWidth,
        int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette, HBITMAP *colorBitmap,
        HBITMAP *monoBitmap);
    virtual void BitmapHandleToArray(SCREENID screenID, HBITMAP colorBitmap,
        HBITMAP monoBitmap, int *bitmapWidth, int *bitmapHeight,
        UCHAR **bitmapArray);
    virtual void Ellipse(SCREENID screenID, int column, int line,
        int startAngle, int endAngle, int xRadius, int yRadius,
        const UI_PALETTE *palette, int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void IconArrayToHandle(SCREENID screenID, int iconWidth,
        int iconHeight, const UCHAR *iconArray, const UI_PALETTE *palette,
        HICON *icon);
    virtual void IconHandleToArray(SCREENID screenID, HICON icon,
        int *iconWidth, int *iconHeight, UCHAR **iconArray);
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground);
    virtual void Polygon(SCREENID screenID, int numPoints,
        const int *polygonPoints, const UI_PALETTE *palette,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
```

```
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion, SCREENID screenID = ID_SCREEN);
    virtual void RegionDefine(SCREENID screenID, int left, int top,
        int right, int bottom);
    virtual void RegionMove(const UI_REGION &oldRegion, int newColumn,
        int newLine, SCREENID oldScreenID = ID_SCREEN,
        SCREENID newScreenID = ID_SCREEN);
    virtual void Text(SCREENID screenID, int left, int top,
        const char *text, const UI_PALETTE *palette, int length = -1,
        int fill = TRUE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN, LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextWidth(const char *string, SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
        int bottom);
    virtual int VirtualPut(SCREENID screenID);

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

protected:
    int maxColors;
    int _virtualCount;
    UI_REGION _virtualRegion;
    char _stopDevice;
    int _fillPattern;
    int _backgroundColor;
    int _foregroundColor;
    int _xor;

    void SetFont(LOGICAL_FONT logicalFont);
    void SetPattern(const UI_PALETTE *palette, int xor);
};
```

- *FGFONT* is a structure that contains the following font information:

  fontptr contains the font information for the first 128 characters.

  fontptr2 contains the font information for the second 128 characters.

- *FGPATTERN* is an array of 16 bytes that make up the bitmap pattern. The patterns defined by Zinc are: PTN_SOLID_FILL, PTN_INTERLEAVE_FILL and PTN_-BACKGROUND_FILL. For more information see the *Zortech C++ Library Reference*.

- *searchPath* contains the path to be searched for font files.

- *fontTable* is an array of *FGFONT*. The default array contains space for 10 *FGFONT* entries. The following entries are pre-defined by Zinc:

FNT_SMALL_FONT—is a font that is used to display an icon's text string.

FNT_DIALOG_FONT—is a font that is used when text is displayed on window objects (e.g., UIW_BUTTON, UIW_STRING, UIW_TEXT, etc.)

FNT_SYSTEM_FONT—is a sans-serif style font that is used to display a window's title.

NOTE: When these three fonts are used, no font files are needed since they are linked into Zinc Application Framework. However, if other fonts are added to this table, the proper font files must either be in the current path or be linked into the application.

- *patternTable* is an array of *FGPATTERN*. The default array contains space for 15 *FGPATTERN* entries. The following entries are pre-defined by Zinc:

  PTN_SOLID_FILL—Solid fill.

  PTN_INTERLEAVE_FILL—Interleaving line fill.

  PTN_BACKGROUND_FILL—Background fill style.

- *maxColors* tells the maximum number of colors supported by the display. For example, an EGA display supports sixteen colors.

- *_virtualCount* is a count of the number of virtual screen operations that have taken place. For example, when the **VirtualGet( )** function is called, *_virtualCount* is decremented. Additionally, when the **VirtualPut( )** function is called, *_virtualCount* is incremented.

- *_virtualRegion* is the region affected by either **VirtualGet( )** or **VirtualPut( )**.

- *_stopDevice* is a variable used to disable the update of the display. If *_stopDevice* is TRUE, no drawing will be done to the screen. Otherwise, drawing will be made directly to the screen display.

- *_fillPattern* is an index into the *patternTable* specifying the current fill pattern.

- *_backgroundColor* is the current background drawing color.

- *_foregroundColor* is the current foreground drawing color.

- *_xor* is TRUE if screen drawing is to be XOR'ed onto the screen. If *_xor* is FALSE, no XOR operation is done.

UI_FG_DISPLAY::UI_FG_DISPLAY

Syntax

#include <ui_dsp.hpp>

UI_FG_DISPLAY(int *mode* = 0);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This constructor returns a pointer to a new UI_FG_DISPLAY object. When a new UI_FG_DISPLAY class is constructed, the system finds the associated Zortech device driver, then clears the screen display to the background color and pattern specified by the member variable *backgroundPalette* that was inherited from UI_DISPLAY.

- $mode_{in}$ is used to determine which graphics driver to initialize. For example, if *mode* were set to EGACOLOR, the Zortech **fg_init_egacolor( )** function would be called. If *mode* is 0, the routine auto-detects the current display mode. For more information on this argument, see **fg_init( )** in the *Zortech C++ Function Reference*.

The default settings of the UI_FG_DISPLAY class are given below:

font—The default font is the Zortech system font. This is an 8x8 bit-mapped font.

mode—The default mode is determined by the *mode* argument discussed above.

background—The background color is determined by the values contained in the *backgroundPalette* member variable.

cell size—The default cell size (i.e., *cellWidth* and *cellHeight*) is 8 pixels wide by 16 pixels high. Thus, when objects are constructed, they are placed on 8x16 cell boundaries.

colors—The run-time colors are determined by the colors defined in the UI_-PALETTE_MAP member variables, or by those defined within an individual application.

Example

```
#include <ui_win.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_FG_DISPLAY;
    if (!display->installed)
    {
        delete display;
        display = new UI_TEXT_DISPLAY;
    }

    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;

    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .

    // Restore the system.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

CHAPTER 13 – UI_GRAPHICS_DISPLAY

The UI_GRAPHICS_DISPLAY class implements a graphics display that uses the GFX graphics routines to display information to the screen. Since the UI_GRAPHICS_-DISPLAY class is derived from UI_DISPLAY, only details specific to the UI_-GRAPHICS_DISPLAY class are given in this chapter. For descriptions and examples regarding virtual or inherited display members, see "Chapter 6—UI_DISPLAY."

The UI_GRAPHICS_DISPLAY class is declared in **UI_DSP.HPP**. Its public and protected members are:

```
class UI_GRAPHICS_DISPLAY : public UI_DISPLAY, public UI_REGION_LIST
{
public:
    struct GRAPHICSFONT
    {
        int font;
        int maxWidth, maxHeight;
    };
    typedef unsigned char GRAPHICSPATTERN[10];

    static GRAPHICSFONT fontTable[MAX_LOGICAL_FONTS];
    static GRAPHICSPATTERN patternTable[MAX_LOGICAL_PATTERNS];

    UI_GRAPHICS_DISPLAY(int mode = 4);
    virtual ~UI_GRAPHICS_DISPLAY(void);
    virtual void Bitmap(SCREENID screenID, int column, int line,
        int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette = NULL,
        const UI_REGION *clipRegion = NULL, HBITMAP *colorBitmap = NULL,
        HBITMAP *monoBitmap = NULL);
    virtual void BitmapArrayToHandle(SCREENID screenID, int bitmapWidth,
        int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette, HBITMAP *colorBitmap,
        HBITMAP *monoBitmap);
    virtual void BitmapHandleToArray(SCREENID screenID, HBITMAP colorBitmap,
        HBITMAP monoBitmap, int *bitmapWidth, int *bitmapHeight,
        UCHAR **bitmapArray);
    virtual void Ellipse(SCREENID screenID, int column, int line,
        int startAngle, int endAngle, int xRadius, int yRadius,
        const UI_PALETTE *palette, int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void IconArrayToHandle(SCREENID screenID, int iconWidth,
        int iconHeight, const UCHAR *iconArray, const UI_PALETTE *palette,
        HICON *icon);
    virtual void IconHandleToArray(SCREENID screenID, HICON icon,
        int *iconWidth, int *iconHeight, UCHAR **iconArray);
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground);
    virtual void Polygon(SCREENID screenID, int numPoints,
        const int *polygonPoints, const UI_PALETTE *palette,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
```

```
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion, SCREENID screenID = ID_SCREEN);
    virtual void RegionDefine(SCREENID screenID, int left, int top,
        int right, int bottom);
    virtual void RegionMove(const UI_REGION &oldRegion, int newColumn,
        int newLine, SCREENID oldScreenID = ID_SCREEN,
        SCREENID newScreenID = ID_SCREEN);
    virtual void Text(SCREENID screenID, int left, int top,
        const char *text, const UI_PALETTE *palette, int length = -1,
        int fill = TRUE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN, LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextWidth(const char *string, SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
        int bottom);
    virtual int VirtualPut(SCREENID screenID);

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

protected:
    int maxColors;
    int _fillPattern;
    int _backgroundColor;
    int _foregroundColor;
    int _fillAttributes;
    int _outlineAttributes;
    char _virtualCount;
    UI_REGION _virtualRegion;
    char _stopDevice;

    void SetFont(LOGICAL_FONT logicalFont);
    void SetPattern(const UI_PALETTE *palette, int xor);
};
```

- *GRAPHICSFONT* is a structure that contains the following font information:

  font contains the value of the font. FNT_SMALL_FONT (font is 0), FNT_-DIALOG_FONT (font is 1) and FNT_SYSTEM_FONT (font is 2) are predefined by Zinc.

  maxHeight is the height of the tallest character.

  maxWidth is the width of the widest character.

- *GRAPHICSPATTERN* is an array of 10 bytes that make up the 8x8 bitmap pattern. The first two bytes indicate the number of rows and columns defined in the pattern. The remaining 8 bytes define the pattern. Each byte (8 bits) corresponds to 8 pixels in the pattern. The patterns defined by Zinc are: PTN_SOLID_FILL, PTN_INTERLEAVE_FILL and PTN_BACKGROUND_FILL.

- *fontTable* is an array of *GRAPHICSFONT*. The default array contains space for 10 *ZINCFONT* entries. The following entries are pre-defined by Zinc:

  FNT_SMALL_FONT—a font used to display an icon's text string.

  FNT_DIALOG_FONT—a font used when text is displayed on window objects (e.g., UIW_BUTTON, UIW_STRING, UIW_TEXT, etc.)

  FNT_SYSTEM_FONT—a sans-serif style font used to display a window's title.

- *patternTable* is an array of *GRAPHICSPATTERN*. The default array contains space for 15 *GRAPHICSPATTERN* entries. The following entries are pre-defined by Zinc:

  PTN_SOLID_FILL—Solid fill.

  PTN_INTERLEAVE_FILL—Interleaving line fill.

  PTN_BACKGROUND_FILL—Background fill style.

- *maxColors* tells the maximum number of colors supported by the display. For example, an EGA display supports sixteen colors.

- *_fillPattern* is an index into the *patternTable* specifying the current fill pattern.

- *_backgroundColor* is the current background drawing color.

- *_foregroundColor* is the current foreground drawing color.

- *_fillAttributes* is the type of filling that takes place (e.g., is the shape filled?, is a line drawn around the filled area?, etc.). This field is only used by the UI_GRAPHICS_-DISPLAY when calling the GFX graphics routines.

- *_outlineAttributes* is the current line style. This field is only used by the UI_-GRAPHICS_DISPLAY when calling the GFX graphics routines.

- *_virtualCount* is a count of the number of virtual screen operations that have taken place. For example, when the **VirtualGet( )** function is called, *_virtualCount* is decremented. Additionally, when the **VirtualPut( )** function is called, *_virtualCount* is incremented.

- *_virtualRegion* is the region affected by either **VirtualGet( )** or **VirtualPut( )**.

- *_stopDevice* is a variable used to disable the update of the display. If *_stopDevice* is TRUE, no drawing will be done to the screen. Otherwise, drawing will be made directly to the screen display.

UI_GRAPHICS_DISPLAY::UI_GRAPHICS_DISPLAY

Syntax

#include <ui_dsp.hpp>

UI_GRAPHICS_DISPLAY(int *mode* = 4);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This constructor returns a pointer to a new UI_GRAPHICS_DISPLAY object. When a new UI_GRAPHICS_DISPLAY class is constructed, the system finds the associated ZINC device driver and sets the screen display to the background color and pattern specified by the inherited variable *backgroundPalette*.

- $mode_{in}$ determines the display mode initialized (e.g., CGA, EGA, VGA, SVGA, etc.).

Example

```
#include <ui_win.hpp>
main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
    .
    .
    .

    return (0);
}
```

CHAPTER 14 – UI_HELP_SYSTEM

The UI_HELP_SYSTEM class is a windowed help system which is used to display help to the end-user during an application. It is the default help class if no other help class has been specified.

The UI_HELP_SYSTEM class is declared in **UI_WIN.HPP**. Its public members are:

```
class EXPORT UI_HELP_SYSTEM
{
public:
    // Members described in UI_HELP_SYSTEM reference chapter.
    static char **helpPrompts;
    UI_HELP_SYSTEM(char *fileName, UI_WINDOW_MANAGER *windowManager = NULL,
        UI_HELP_CONTEXT helpContext = NO_HELP_CONTEXT);
    virtual ~UI_HELP_SYSTEM(void);
    virtual void DisplayHelp(UI_WINDOW_MANAGER *windowManager,
        UI_HELP_CONTEXT helpContext = NO_HELP_CONTEXT);

protected:
    // Members described in UI_HELP_SYSTEM reference chapter.
    UI_STORAGE *storage;
    UIW_WINDOW *helpWindow;
    UIW_TITLE *titleField;
    UIW_TEXT *messageField;
    UI_HELP_CONTEXT defaultHelpContext;
};
```

- *helpPrompts* is a pointer to an array of text strings that will be placed as options in the help window's system button pop-up menu. This array is defined and assigned to this static member in the **G_PHELP.CPP** module. Placing these literal strings in a lookup table allows the programmer to easily change the text that will be displayed (e.g., the text could be changed to a language other than English).

- *storage* is a pointer to the UI_STORAGE that contains the help data.

- *helpWindow* is a pointer to the UIW_WINDOW used to display help messages.

- *titleField* is a pointer to the UIW_TITLE object on the help window.

- *messageField* is a pointer to the UIW_TEXT field used to display the help text.

- *defaultHelpContext* contains the default help context to be used if no other help context is specified.

Generating help files

The help context information is read from a binary help file on the disk when needed.

This file is created either in Zinc Designer or from an ASCII text file using the **GENHELP.EXE** utility which is supplied with Zinc Application Framework. For example, the text file **WINMGR.TXT** below was first created using a text editor and then converted into a binary help file using **GENHELP.EXE**.

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

Press <F3> to continue...

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

Mark  - <Ctrl+F5> or <Left-drag> on the mouse                    \
Cut   - <Shift+Del> or <Right-down-click> on the mouse           \
Copy  - <Ctrl+Ins> or <Left-down><Right-down-click> the mouse    \
Paste - <Shift+Ins> or <Right-down-click> on the mouse           \

Press <F3> to continue...
```

There are two help contexts in the example above. Each one is preceded by the help context name, set off by three dashes on the left side. 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.

Typing **genhelp winmgr.txt** at the command line generates two files. Be sure that the path to the file **GENHELP.EXE**, located in the **BIN** directory, is included in the environment PATH variable.

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

```
#define HELP_GENERAL      0x0001  // General Help
#define HELP_WINMGR       0x0002  // Winmgr 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 international versions of the application require different help files.

UI_HELP_SYSTEM::UI_HELP_SYSTEM

Syntax

#include <ui_win.h>
#include "*filename*.hpp"

UI_HELP_SYSTEM(char **fileName*,
    UI_WINDOW_MANAGER **windowManager* = NULL,
    UI_HELP_CONTEXT *helpContext* = NO_HELP_CONTEXT);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UI_HELP_SYSTEM class object.

- *fileName*$_{in}$ is a pointer to a string containing the name of the binary help file. This file is generated in Zinc Designer or from an ASCII text file using the **GENHELP.EXE** utility.

- *windowManager*$_{in}$ is a pointer to the Window Manager. It is used by the help system to display the help window on the screen display.

- *helpContext*$_{in}$ is the help context to present when no specific help context is available.

Example

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

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_MSC_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
```

```
    UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANAGER(display,
        eventManager);
    UI_WINDOW_OBJECT::errorSystem = new UI_ERROR_SYSTEM;
    UI_WINDOW_OBJECT::helpSystem = new UI_HELP_SYSTEM("demo.dat",
         windowManager, HELP_GENERAL);
    .
    .
    .

    // Restore the system.
    delete UI_WINDOW_OBJECT::helpSystem;
    delete UI_WINDOW_OBJECT::errorSystem;
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

UI_HELP_SYSTEM::~UI_HELP_SYSTEM

Syntax

#include <ui_win.h>
#include "*filename*.hpp"

virtual ~UI_HELP_SYSTEM(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_HELP_-SYSTEM object.

UI_HELP_SYSTEM::DisplayHelp

Syntax

```
#include <ui_win.h>
#include "filename.hpp"

virtual void DisplayHelp(UI_WINDOW_MANAGER *windowManager,
    UI_HELP_CONTEXT helpContext = NO_HELP_CONTEXT);
```

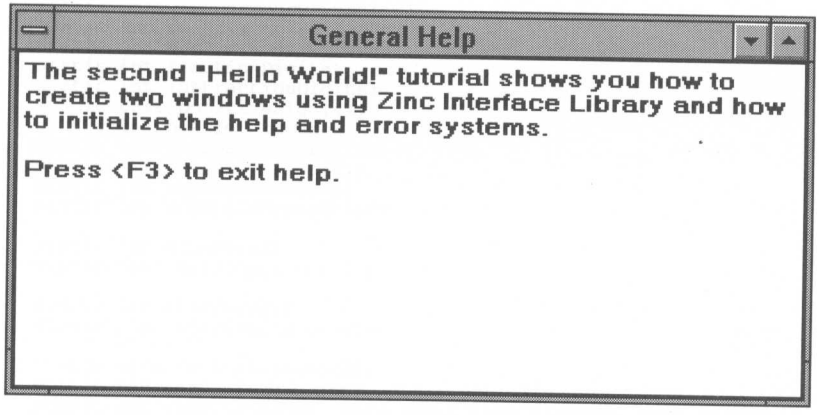
Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to present help information via the help system. The picture below shows a graphic representation of the UI_HELP_SYSTEM presentation window:



- *windowManager*$_{in}$ is a pointer to the Window Manager where the help window is to be presented.

- *helpContext*$_{in}$ is the help context to present. If this value is NO_HELP_CONTEXT, the help window system will use the default help context provided in the UI_HELP_-SYSTEM constructor.

Example

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

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MS_MOUSE
        + new UID_CURSOR;
    UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANAGER(display,
        eventManager);
    UI_WINDOW_OBJECT::errorSystem = new UI_ERROR_SYSTEM;
    UI_WINDOW_OBJECT::helpSystem = new UI_HELP_SYSTEM("demo.dat",
         windowManager, HELP_GENERAL);
    .
    .
    .
    // Call the help system to display general help.
    windowManager->helpSystem->DisplayHelp(windowManager, HELP_GENERAL);
    .
    .       .
    .

}
```

CHAPTER 15 – UI_INTERNATIONAL

The UI_INTERNATIONAL class is used to allow data to be displayed or input according to the default country settings. This allows data to be automatically displayed differently when the country settings in the system setup file (e.g., CONFIG.SYS) are changed. Library classes derived from UI_INTERNATIONAL include: UI_BIGNUM, UI_DATE and UI_TIME.

The UI_INTERNATIONAL class structure is declared in **UI_GEN.HPP**. Its members are:

```
class EXPORT UI_INTERNATIONAL
{
    friend class EXPORT UI_ERROR_SYSTEM;
public:
    static int initialized;
    static int countryCode;

    static char languageName[64];
    static char localeName[64];
    static char decimalSeparator[4];
    static char monDecimalSeparator[4];
    static char thousandsSeparator[4];
    static char monThousandsSeparator[4];
    static char currencySymbol[8];
    static char grouping[10];
    static char monGrouping[10];
    static char intCurrencySymbol[8];
    static int posCurrencyPrecedes;
    static int negCurrencyPrecedes;
    static int fractionDigits;
    static int intFractionDigits;
    static char positiveSign[4];
    static int posSignPrecedes;
    static int posSpaceSeparation;
    static char negativeSign[4];
    static int negSignPrecedes;
    static int negSpaceSeparation;

    static int dateFormat;
    static char dateSeparator[4];

    static int timeFormat;
    static char timeSeparator[4];

    static void Initialize(void);
    static int sprintf(char *dst, const char *format, ...);
    static int vsprintf(char *dst, const char *format, void *args);
protected:
    static void *rearrangeArgs(const char *format, const void *args,
       char *newFormat, void *newArgs);
};
```

NOTE: The following examples use the United States format unless otherwise specified.

- *initialized* shows if the UI_INTERNATIONAL class has been initialized. This variable is initially set to FALSE.

- *countryCode* is the system's current country code. This member is available in DOS only.

- *languageName* is a string that specifies the language used for displaying all text in the program.

- *localeName* is a string that specifies which country's style of formatting (such as on monetary units) is to be performed.

- *decimalSeparator* is the system's decimal separator (e.g., 100.00).

- *monDecimalSeparator* is the system's currency decimal separator (e.g., $100.00).

- *thousandsSeparator* is the system's thousands separator (e.g., 100,000).

- *monThousandsSeparator* is the system's currency thousands separator (e.g., $100,000.00).

- *currencySymbol* is the system's currency symbol (e.g., '$').

- *grouping* is a string that indicates the format to be used for grouping digits in a number. For more specific information, see the ANSI Standard Specification for the C Programming Language.

- *monGrouping* is a string that indicates the format to be used for grouping digits in a monetary number. For more specific information, see the ANSI Standard Specification for the C Programming Language.

- *intCurrencySymbol* is the system's international currency symbol. (e.g., '**USD**')

- *posCurrencyPrecedes* is TRUE if the currency symbol precedes the currency amount (e.g., '$100,000.00') for positive currency values. Otherwise, *posCurrencyPrecedes* is FALSE and the currency symbol will be displayed following the currency amount.

- *negCurrencyPrecedes* is TRUE if the currency symbol precedes the currency amount (e.g., '-$100,000.00') for negative currency values. Otherwise, *negCurrencyPrecedes* is FALSE and the currency symbol will be displayed following the currency amount.

- *fractionDigits* specifies the number of fractional digits to display after the decimal point on currency values (e.g., $100.**00**).

- *intFractionDigits* specifies the number of fractional digits to display after the decimal point on currency values in international format (e.g., USD100.**00**).

- *positiveSign* specifies the symbol for positive values (e.g., +100 or 100).

- *posSignPrecedes* is TRUE if the symbol for positive amounts is displayed before the currency symbol. Otherwise, *posSignPrecedes* is FALSE and the symbol will be displayed following the amount.

- *posSpaceSeparation* is TRUE if there is a space separator between the currency symbol and the currency amount when the currency is positive. Otherwise, *posSpaceSeparation* is FALSE.

- *negativeSign* specifies the symbol for negative values (e.g., -100).

- *negSignPrecedes* is TRUE if the symbol for negative amounts is displayed before the currency symbol. Otherwise, *negSignPrecedes* is FALSE and the symbol will be displayed following the amount.

- *negSpaceSeparation* is TRUE if there is a space separator between the currency symbol and the currency amount when the currency is negative. Otherwise, *negSpaceSeparation* is FALSE.

- *dateFormat* is the system's current date format. (e.g., **12/25/91**)

- *dateSeparator* is the system's date separator. (e.g., 12/25/91)

- *timeFormat* is the system's current time format. (e.g., **12:00a**)

- *timeSeparator* is the system's time separator. (e.g., 12:00)

UI_INTERNATIONAL::Initialize

Syntax

```
static void Initialize(void);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

Initialize( ) sets up the default country information according to the country-specific information obtained from the operating system.

Example 1

```
void UI_INTERNATIONAL::Initialize(void)
{
    // Make sure the country information has not already been initialized.
    if (initialized)
        return;
    initialized = TRUE;
            .
    // Get the proper operating system version.
    REGS regs;
    DOS2_COUNTRY dos2Info;
    DOS3_COUNTRY dos3Info;
    extern unsigned char _osmajor;
    if (_osmajor == 2)
        regs.x.dx = (USHORT)&dos2Info;
    else
        regs.x.dx = (USHORT)&dos3Info;

    // Determine the country code.
    regs.x.ax = 0x3800;
    int86(0x21, &regs, &regs);
    countryCode = (regs.h.al == 0xFF) ? regs.x.bx : regs.h.al;
    if (_osmajor == 2)
    {
        dateFormat = dos2Info.dateFormat;
        strcpy(currencySymbol, dos2Info.currencySymbol);
        strcpy(intCurrencySymbol, dos2Info.currencySymbol);
        strcpy(thousandsSeparator, dos2Info.thousandsSeparator);
        strcpy(monThousandsSeparator, dos2Info.thousandsSeparator);
        strcpy(decimalSeparator, dos2Info.decimalSeparator);
        strcpy(monDecimalSeparator, dos2Info.decimalSeparator);
    }
    else
    {
        dateFormat = dos3Info.dateFormat;
        strcpy(currencySymbol, dos3Info.currencySymbol);
        strcpy(intCurrencySymbol, dos3Info.currencySymbol);
        strcpy(thousandsSeparator, dos3Info.thousandsSeparator);
        strcpy(monThousandsSeparator, dos3Info.thousandsSeparator);
        strcpy(decimalSeparator, dos3Info.decimalSeparator);
        strcpy(monDecimalSeparator, dos3Info.decimalSeparator);
        strcpy(dateSeparator, dos3Info.dateSeparator);
        strcpy(timeSeparator, dos3Info.timeSeparator);
        posSignPrecedes = negSignPrecedes = 1;
        switch (dos3Info.currencyStyle)
        {
```

```
            case 0:
                posSpaceSeparation = negSpaceSeparation = 0;
                posCurrencyPrecedes = negCurrencyPrecedes = 1;
                break;

            case 1:
                posSpaceSeparation = negSpaceSeparation = 0;
                posCurrencyPrecedes = negCurrencyPrecedes = 0;
                break;

            case 2:
                posSpaceSeparation = negSpaceSeparation = 1;
                posCurrencyPrecedes = negCurrencyPrecedes = 1;
                break;

            case 3:
                posSpaceSeparation = negSpaceSeparation = 1;
                posCurrencyPrecedes = negCurrencyPrecedes = 0;
                break;
            }
            intFractionDigits = fractionDigits = dos3Info.significantDigits;
            timeFormat = dos3Info.timeFormat;
        }

        // Get the locale information.
        struct lconv *conv = localeconv();
        if (*conv->decimal_point)
            strcpy(decimalSeparator, conv->decimal_point);
        if (*conv->mon_decimal_point)
            strcpy(monDecimalSeparator, conv->mon_decimal_point);
        if (*conv->thousands_sep)
            strcpy(thousandsSeparator, conv->thousands_sep);
        .
        .
        .
    }
```

Example 2

```
void UI_DATE::DataSet(const char *string, DTF_FLAGS dtFlags)
{
    // Make sure the country information is initialized.
    if (!UI_INTERNATIONAL::initialized)
        UI_INTERNATIONAL::Initialize();
}
```

CHAPTER 16 – UI_ITEM

The UI_ITEM structure is used to store **item** information in the form of an array. The actual type of object created depends upon the type of object to which the UI_ITEM is added. For example, when a UI_ITEM is added to a UIW_PULL_DOWN_MENU, a UIW_PULL_DOWN_ITEM is created. However, if the UI_ITEM is added to a UIW_-COMBO_BOX, UIW_HZ_LIST or UIW_VT_LIST, a UIW_STRING object is created.

The UI_ITEM structure is declared in **UI_WIN.HPP**. Its fields are:

```
struct EXPORT UI_ITEM
{
    // Fields described in UI_ITEM reference chapter.
    EVENT_TYPE value;
    void *data;
    char *text;
    UIF_FLAGS flags;
};
```

- *value* is a number associated with the item. It is used to either identify a particular item or it can be the type of event to be put on the event queue when the item is selected (i.e., if the MNIF_SEND_MESSAGE flag is set.) For example, if the following items were specified:

```
UI_ITEM _menuFlag[] =
{
    { ONE,      NULL,    " Item &one ",   MNIF_NO_FLAGS },
    { TWO,      NULL,    " Item &two ",   MNIF_NON_SELECTABLE },
    { L_EXIT,   NULL,    " E&xit",        MNIF_SEND_MESSAGE },
    { 0,        NULL,    NULL,            NULL }
};
```

  and the "Exit" option is selected, an event with type set to L_EXIT would be put on the event queue.

- *data* may contain the function to associate with the item. When the object is selected, the function pointed to by *data* will be called. For example, the **_menuFlag** array defined above could be modified to contain a user function for each menu item:

```
UI_ITEM _menuFlag[] =
{
    { BTF_NO_TOGGLE,    CheckFlag,   " No toggle ",    WOF_NO_FLAGS },
    { BTF_DOWN_CLICK,   CheckFlag,   " Down click ",   WOF_NO_FLAGS },
    { BTF_CHECK_MARK,   CheckFlag,   " Check mark ",   WOF_NO_FLAGS },
    { BTF_AUTO_SIZE,    CheckFlag,   " Auto size ",    WOF_NO_FLAGS },
    { 0,                NULL,        NULL,             NULL }
};

static EVENT_TYPE CheckFlag(UI_WINDOW_OBJECT *data,
    const UI_EVENT &event, EVENT_TYPE ccode)
{
```

```
    // Typecast a pointer to the pop-up-item.
    UIW_POP_UP_ITEM *item = (UIW_POP_UP_ITEM *)data;
    switch (item->value)
    {
    case BTF_NO_TOGGLE:
        .
        .
        .
        break;

    case BTF_DOWN_CLICK:
        .
        .
        .
        break;
    }
    return ccode;
}
```

- *text* is the text that is displayed to the screen.

- *flags* is the UIF_FLAGS that are used when constructing the class object. If the item constructed is a UIW_POP_UP_ITEM, *flags* is interpreted to be MNIF_FLAGS as the default menu-item flags. (See "Chapter 57—UIW_POP_UP_ITEM" for the available MNIF_FLAGS.) If the item constructed is a UIW_STRING, then *flags* is interpreted to be STF_FLAGS as the default string flags. (See "Chapter 64—UIW_STRING" for the available STF_FLAGS.)

NOTE: The end-of-array indicator is provided by a UI_ITEM object that has 0 and NULL as the *value*, *userFunction*, *text* and *flags*. This field must be provided since no array size argument is provided:

```
UI_ITEM _buttonFlag[] =
{
    .
    .
    .
    { 0,    NULL,    NULL,    NULL }  // End of array.
};
```

Portability

This structure is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    UI_ITEM listItems[] =
    {
        { 11,       NULL,       "Item 1.1",     STF_NO_FLAGS },
        { 12,       NULL,       "Item 1.2",     STF_NO_FLAGS },
        { 21,       NULL,       "Item 2.1",     STF_NO_FLAGS },
        { 22,       NULL,       "Item 2.2",     STF_NO_FLAGS },
        { 0,        NULL,       NULL,           NULL }
    };

    // Create the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample List ")
        + new UIW_PROMPT(2, 1, "List:")
        + new UIW_VT_LIST(10, 1, 20, 6, NULL, 0, listItems);
    *windowManager + window;
    .
    .
    .

}
```

CHAPTER 17 – UI_KEY

The UI_KEY structure is used to store keyboard information passed through the library in the UI_EVENT structure.

The UI_KEY structure class is declared in **UI_EVT.HPP**. Its fields are:

```
struct EXPORT UI_KEY
{
    // Fields described in UI_KEY reference chapter.
    RAW_CODE shiftState;
    RAW_CODE value;
};
```

- *shiftState* is the shift state of the keyboard. The shift state may contain one or more of the following flags (declared in **UI_EVT.HPP**):

  S_ALT—The <Alt> key is pressed.

  S_CAPS_LOCK—The <Caps-Lock> key is on.

  S_CTRL—The <Ctrl> key is pressed.

  S_INSERT—The <Ins> key is on.

  S_LEFT_SHIFT—The <Left-Shift> key is pressed.

  S_NUM_LOCK—The <Num-Lock> key is on.

  S_RIGHT_SHIFT—The <Right-Shift> key is pressed.

  S_SCROLL_LOCK—The <Scroll-Lock> key is on.

- *value* is the low eight bits of the scan code. If a non-function key is pressed, this gives the ASCII value of the key. For example, the character 'a' produces a scan code of 0x1E61 but has an associated ASCII value of 0x61 (i.e., the character 'a' in the ASCII character set).

Portability

This structure is available on the following environments:

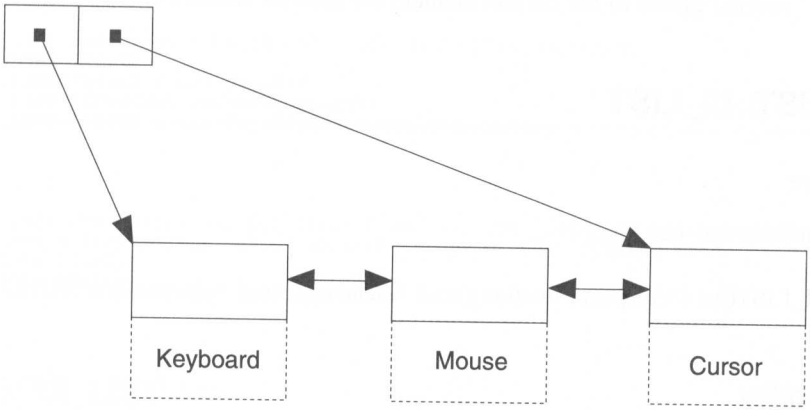
■ DOS ■ MS Windows ■ OS/2 ■ Motif

Example

```
#include <ui_evt.hpp>

void UID_KEYBOARD::Poll()
{
    // See if a keystroke is already waiting.
    .
    .
    .

    // Get the key from the keyboard bios using INT 16H, 10H(or 00H if not
    // enhanced).
    UI_EVENT event;
    event.type = E_KEY;
    inregs.h.ah = _enhancedBios;
    int86(0x16, &inregs, &outregs);
    event.rawCode = outregs.x.ax;
    event.key.value = outregs.h.al;

    // Get the shift state using INT 16H, 12H(or 02H if not enhanced).
    inregs.h.ah = 0x02 + _enhancedBios;
    int86(0x16, &inregs, &outregs);
    event.key.shiftState = outregs.h.al;

    // Place event on the queue.
    if (state != D_OFF && eventManager)
        eventManager->Put(event, Q_END);
}
```

CHAPTER 18 – UI_LIST

The UI_LIST class is used to store doubly-linked list elements derived from the UI_-ELEMENT base class. It also serves as the base class to all Zinc Application Framework management classes (e.g., UI_EVENT_MANAGER, UI_REGION_LIST) and many control objects (e.g., UIW_WINDOW, UI_DISPLAY). All elements in a list must be derived from the UI_ELEMENT class since all list member functions act upon UI_-ELEMENT class objects. The figure below illustrates how elements are linked together in a list:



The UI_LIST class is declared in **UI_GEN.HPP**. Its public and protected members are:

```
class EXPORT UI_LIST
{
    friend class EXPORT UI_LIST_BLOCK;

public:
    // Members described in UI_LIST reference chapter.
    int (*compareFunction)(void *element1, void *element2);

    UI_LIST(int (*_compareFunction)(void *element1, void *element2) = NULL);
    virtual ~UI_LIST(void);
    UI_ELEMENT *Add(UI_ELEMENT *newElement);
    UI_ELEMENT *Add(UI_ELEMENT *element, UI_ELEMENT *newElement);
    int Count(void);
    UI_ELEMENT *Current(void);
    virtual void Destroy(void);
    UI_ELEMENT *First(void);
    UI_ELEMENT *Get(int index);
    UI_ELEMENT *Get(int (*findFunction)(void *element1, void *matchData),
        void *matchData);
    int Index(UI_ELEMENT const *element);
    UI_ELEMENT *Last(void);
    void SetCurrent(UI_ELEMENT *element);
    void Sort(void);
    UI_ELEMENT *Subtract(UI_ELEMENT *element);
    UI_LIST &operator+(UI_ELEMENT *element);
```

```
    UI_LIST &operator-(UI_ELEMENT *element);

protected:
    // Members described in UI_LIST reference chapter.
    UI_ELEMENT *first, *last, *current;
};
```

- *compareFunction* is used by the list to determine the order of each element when the list is sorted. The two void element pointer arguments must be typecast when the specified compare function is called.

- *first* and *last* point to the first and last elements in the list.

- *current* points to the current element (or derived element class) in the list.

UI_LIST::UI_LIST

Syntax

#include <ui_gen.hpp>

UI_LIST(int (**compareFunction*)(void **element1*, void **element2*) = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UI_LIST object.

- *compareFunction*$_{in}$ is a programmer-defined function that is used to determine the order of list elements. The following arguments are passed to *compare*:

  element1$_{in}$—A pointer to the first argument to compare. This argument must be typecast by the programmer.

  element2$_{in}$—A pointer to the second argument to compare. This argument must be typecast by the programmer.

The compare function's *returnValue* should be 0 if the two elements exactly match. A negative value should be returned if *element1* is less than *element2*. Otherwise, a positive value should be returned if *element1* is greater than *element2*.

Example

```
#include <ui_gen.hpp>

int ButtonValueCompare(void *button1, void *button2)
{
    return(((UIW_BUTTON *)button1)->value - ((UIW_BUTTON *)button2)->value);
}

ExampleFunction()
{
    // Each declaration below calls the UI_LIST constructor.
    UI_LIST list1;
    UI_LIST *list2 = new UI_LIST;
    UI_LIST list3(ButtonValueCompare);
    UI_LIST *list4 = new UI_LIST(ButtonValueCompare);
    .
    .
    .

    // Call the destructor for lists 2 and 4.  The list1 and list3 destructors
    // are automatically called when the scope of this routine ends.
    delete list2;
    delete list4;
}
```

UI_LIST::~UI_LIST

Syntax

#include <ui_gen.hpp>

virtual ~UI_LIST(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_LIST object. It calls the destructor associated with each element in the list.

Example

```
#include <ui_gen.hpp>

int ButtonValueCompare(void *button1, void *button2)
{
    return(((UIW_BUTTON *)button1)->value - ((UIW_BUTTON *)button2)->value);
}

ExampleFunction()
{
    // Each declaration below calls the UI_LIST constructor.
    UI_LIST list1;
    UI_LIST *list2 = new UI_LIST;
    UI_LIST list3(ButtonValueCompare);
    UI_LIST *list4 = new UI_LIST(ButtonValueCompare);
    .
    .
    .

    // Call the destructor for lists 2 and 4.  The list1 and list3 destructors
    // are automatically called when the scope of this routine ends.
    delete list2;
    delete list4;
}
```

UI_LIST::Add
UI_LIST::operator +

Syntax

#include <ui_gen.hpp>

UI_ELEMENT *Add(UI_ELEMENT **newElement*);
or
UI_ELEMENT *Add(UI_ELEMENT **element*, UI_ELEMENT **newElement*);
or
UI_LIST &operator + (UI_ELEMENT **element*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded functions are used to add a new element to the UI_LIST object.

The first overloaded function adds a new element to the UI_LIST object into a position specified by the list's *compareFunction*. The new element must be a class object derived from the UI_ELEMENT base class. If no compare function is specified when the list is constructed, *newElement* is added to the end of the list.

- *returnValue*$_{out}$ is a pointer to *newElement* if the addition was successful. Otherwise, *returnValue* is NULL.

- *newElement*$_{in}$ is a pointer to the element to be added to the list. This argument must be a class object derived from the UI_ELEMENT base class.

The second overloaded function overrides the list's *compareFunction* by inserting *newElement* directly before *element*. The new element must be a class object derived from the UI_ELEMENT base class. The **UI_LIST::Sort( )** routine may be called to sort the list when this routine is used.

- *returnValue*$_{out}$ is a pointer to *newElement* if the addition was successful. Otherwise, *returnValue* is NULL.

- *element*$_{in}$ is a pointer to an element before which the new element is to be placed. If this variable is NULL, the routine adds *newElement* to the end of the list.

- *newElement*$_{in}$ is a pointer to the element to be added to the list. This argument must be a class object derived from the UI_ELEMENT base class.

The third overloaded operator adds an element to the UI_LIST object. This operator overload is equivalent to calling the **UI_LIST::Add( )** routine, except that it allows the chaining of list element additions to the UI_LIST object.

- *returnValue*$_{out}$ is the UI_LIST reference. Returning the reference to the UI_LIST object allows chaining of the **UI_LIST::operator+** overload operator.

- *element*$_{in}$ is a pointer to the UI_ELEMENT class element or to the object derived from the UI_ELEMENT base class that is to be added to the list.

NOTE: The **Add( )** function and the **+ operator** are also used by windows, by the Window Manager and by the Event Manager. The UIW_WINDOW class adds a window object to its list by using **Add( )** or the **+ operator**. UI_WINDOW_MANAGER uses **Add( )** or the **+ operator** to add a window object to the Window Manager. UI_-EVENT_MANAGER uses these functions to add a device to the Event Manager.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a new window and attach it to the window manager.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1");
    *windowManager + window;
}
```

UI_LIST::Count

Syntax

#include <ui_gen.hpp>

int Count(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine counts the number of elements in the UI_LIST object.

- *returnValue*$_{out}$ is the number of elements in the list.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_LIST list1;
    list1.Add(NULL, new UI_ELEMENT);
    list1.Add(NULL, new UI_ELEMENT);
    .
    .
    .
```

```
    // Count the number of elements in the list.
    int count = list1.Count();
    .
    .
    .

}
```

UI_LIST::Current

Syntax

#include <ui_gen.hpp>

UI_ELEMENT *Current(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a pointer to the current element, if one exists, in the list.

NOTE: The **Current( )** function is also used by window objects and devices. In each case, the function is overloaded to return an object pointer typecast according to the context. For example, window objects generally return a UI_WINDOW_OBJECT pointer when **Current( )** is called:

```
// Load a resource from SAMPLE.DAT and find the current object.
UIW_WINDOW *window = (UIW_WINDOW *)
    UI_WINDOW_OBJECT::New("sample.dat~WINDOW1");
*windowManager + window;

UI_WINDOW_OBJECT *currentObject = window->Current();
    .
    .
    .
```

Input devices, however, return a UI_DEVICE pointer:

```
UI_DISPLAY *display = new UI_TEXT_DISPLAY;
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);

*eventManager
    + new UID_KEYBOARD
```

```
        + new UID_MOUSE
        + new UID_CURSOR;
    UI_DEVICE *device = eventManager->Current();
        .
        .
        .
```

Some other class objects return specific element pointers (e.g., UI_QUEUE_ELEMENT, UI_REGION_ELEMENT, UIW_POP_UP_ITEM). Refer to each class object for information about the return value of **Current( )**.

UI_LIST::Destroy

Syntax

#include <ui_gen.hpp>

virtual void Destroy(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine calls the destructor associated with each element in the UI_LIST object, then clears the *first*, *last* and *current* members. The list's *compareFunction* remains unchanged. Because the destructor of objects derived from UI_ELEMENT is virtual, the destructor of the most-derived class will be called first.

Example 1

```
#include <ui_gen.hpp>

ExampleFunction1(UI_ELEMENT *element1)
{
    UI_LIST list1;
    list1.Add(element1);
    .
    .
    .
```

```
    // Destroy all the elements of the list.
    list1.Destroy();
    .
    .
    .

}
```

Example 2

```
ExampleFunction2(UI_ELEMENT *element1, UI_ELEMENT *element2)
{
    UI_LIST *list2 = new LIST;
    *list2 + element1 + element2;
    .
    .
    .

    // Destructively remove all items from the list.  The
    // element destructor is called for each item in the list.
    // Notice we have to also call delete on the list, since it was
    // dynamically constructed.
    list2->Destroy();
    delete list2;
    .
    .
    .

}
```

UI_LIST::First

Syntax

#include <ui_gen.hpp>

UI_ELEMENT *First(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a pointer to the first element, if one exists, in the list.

NOTE: The **First( )** function is also used by window objects and devices. In each case, the function is overloaded to return an object pointer typecast according to the context. For example, window objects generally return a UI_WINDOW_OBJECT pointer when **First( )** is called:

```
// Load a resource from SAMPLE.DAT and find the first object.
UIW_WINDOW *window = (UIW_WINDOW *)
    UI_WINDOW_OBJECT::New("sample.dat~WINDOW1");
*windowManager + window;

UI_WINDOW_OBJECT *currentObject = window->First();
    .
    .
    .
```

Input devices, however, return a UI_DEVICE pointer:

```
UI_DISPLAY *display = new UI_TEXT_DISPLAY;
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
*eventManager
    + new UID_KEYBOARD
    + new UID_MOUSE
    + new UID_CURSOR;
for (UI_DEVICE *device = eventManager->First(); device;
    device = device->Next());
    .
    .
    .
```

Some other class objects return specific element pointers (e.g., UI_QUEUE_ELEMENT, UI_REGION_ELEMENT, UIW_POP_UP_ITEM). Refer to each class object for information about the return value of **First( )**.

UI_LIST::Get

Syntax

```
#include <ui_gen.hpp>

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

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded functions are used to get a specific list element.

The first overloaded function returns the list element specified by *index*. The first element in the list has an index value of 0. If the index value is invalid, NULL is returned.

- *returnValue*$_{out}$ is a pointer to the matching element of the list. This value is NULL if no element matched the index value.

- *index*$_{in}$ is the index of the list element to find. List element indexes are zero based (i.e., the first element in a list has an index value of 0).

The second overloaded function searches the UI_LIST object for a pattern matched by *find*.

- *returnValue*$_{out}$ is a pointer to the matching list element. This value is NULL if no element matches *match*.

- *findFunction*$_{in}$ is a pointer to a programmer supplied function that compares a specified element with the typecast *match*. If an exact match is made this function must return a 0. Any non-zero value indicates that no match was made.

- *matchData*$_{in}$ is a pointer to the data to be matched. This can point to any data the programmer desires to match. The **Get( )** routine will call the *find* routine with this argument as the *matchData* parameter.

Example 1

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_LIST list;
    list + new ITEM("Item1") + new ITEM("Item2");
    .
    .
    .
```

```
    // Get the 2nd element in the list.
    UI_ELEMENT *element = list.Get(1);
    // Get the element that matches the "Item2" pattern.
    ITEM *item = (ITEM *)list.Get(ITEM::Find, "Item2");
    .
    .
    .

}
```

Example 2

```
FindElement(void *element1, void *element2)
{
    return ((element1 == element2) ? 0 : -1);
}

ExampleFunction2()
{
    UI_LIST list2;
    ITEM *item;
    *list2
        + new ITEM("Item3")
        + (item = new ITEM("Item1"))
        + new ITEM("Item2");
    .       .
    .
    .

    // Get the first element in the list.
    ITEM *item = (ITEM *)list.Get(0);

    // See if item is still in the list.
    if (list.Get(FindElement, item))
        cout << "Item1 was found in the list.";
    else
        cout << "Item1 was NOT found in the list.";
}
```

UI_LIST::Index

Syntax

#include <ui_gen.hpp>

int Index(UI_ELEMENT const **element*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine returns the index value of the specified element. If no element matches the specified element, -1 is returned.

- *returnValue*$_{out}$ gives the index of the element in the UI_LIST object. List element indexes are zero based (i.e., the first element in a list has an index value of 0). If *element* is not found in the UI_LIST object, -1 is returned.

- *element*$_{in}$ is a pointer to the list element to find. This element must either be UI_ELEMENT or be derived from the UI_ELEMENT class.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_LIST list;
    ITEM *item3 = new ITEM("Item3");
    ITEM *item1 = new ITEM("Item1");
    ITEM *item2 = new ITEM("Item2");
    list + item3 + item1 + item2;
    .
    .
    .

    list.Sort();
    // Get the index number of an element in a sorted list.
    cout << "Item1 is item #" << list.Index(item1) + 1 << "in the list.";
}
```

UI_LIST::Last

Syntax

#include <ui_gen.hpp>

UI_ELEMENT *Last(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a pointer to the last element, if one exists, in the list.

NOTE: The **Last( )** function is also used by window objects and devices. In each case, the function is overloaded to return an object pointer typecast according to the context. For example, window objects generally return a UI_WINDOW_OBJECT pointer when **Last( )** is called:

```
// Load a resource from SAMPLE.DAT and find the last object.
UIW_WINDOW *window = (UIW_WINDOW *)
    UI_WINDOW_OBJECT::New("sample.dat~WINDOW1");
*windowManager + window;

UI_WINDOW_OBJECT *currentObject = window->Last();
    .
    .
    .
```

Input devices, however, return a UI_DEVICE pointer:

```
UI_DISPLAY *display = new UI_TEXT_DISPLAY;
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
*eventManager
    + new UID_KEYBOARD
    + new UID_MOUSE
    + new UID_CURSOR;
for (UI_DEVICE *device = eventManager->Last(); device;
    device = device->Previous());
    .
    .
    .
```

Some other class objects return specific element pointers (e.g., UI_QUEUE_ELEMENT, UI_REGION_ELEMENT, UIW_POP_UP_ITEM). Refer to each class object for information about the return value of **Last( )**.

UI_LIST::SetCurrent

Syntax

#include <ui_gen.hpp>

void SetCurrent(UI_ELEMENT **element*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to set the current item in the list.

- *element*$_{in}$ is a pointer to the element in the list that will become current.

NOTE: *element* must be a member of the list (i.e., it must have been previously added to the list.)

Example

```
EVENT_TYPE COMPORT_SETUP::ResetDefaults(UI_WINDOW_OBJECT *object,
    UI_EVENT &event, EVENT_TYPE ccode)
{
    if (ccode != L_SELECT)
        return ccode;

    for (UI_WINDOW_OBJECT *window = object; window->parent;
        window = window->parent)
            ;

    COMPORT_SETUP *parentWindow = (COMPORT_SETUP *)window;
    .
    .
    .

    // Reset the combo box's default current items.
    UI_EVENT tEvent;
    tEvent = event;
    event.type = S_CREATE;
    tEvent.type = S_REDISPLAY;
    parentWindow->portField->list.SetCurrent(parentWindow->defaultPort);
    parentWindow->portField->Event(event);
    parentWindow->portField->Event(tEvent);
    .
    .
    .

    return ccode;
}
```

UI_LIST::Sort

Syntax

#include <ui_gen.hpp>

void Sort(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine sorts the UI_LIST object according to the *compareFunction*. If the list has no compare function, no sort occurs.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_LIST list(ITEM::Compare);
    ITEM *item3 = new ITEM("Item3");
    ITEM *item1 = new ITEM("Item1");
    ITEM *item2 = new ITEM("Item2");
    list + item3 + item1 + item2;
    .
    .
    .

    // Sort a list of items.
    list.Sort();
}
```

UI_LIST::Subtract
UI_LIST::operator –

Syntax

```
#include <ui_gen.hpp>

UI_ELEMENT *Subtract(UI_ELEMENT *element);
    or
UI_LIST &operator – (UI_ELEMENT *element);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These functions remove an element from the UI_LIST object.

The first function removes an element from the UI_LIST object but does not call the destructor associated with the element. The programmer is responsible for deletion of each object explicitly subtracted from a list.

- *returnValue*$_{out}$ is a pointer to the next element in the list. This value is NULL if there are no more elements after the deleted element.

- *element*$_{in}$ is a pointer to the element to be deleted from the list. The *element* class destructor is not called by **Subtract( )**.

The second overloaded operator removes an element from the UI_LIST object. This operator overload is equivalent to calling the **Subtract( )** routine, except that it allows the chaining of list element removals from the UI_LIST object.

- *returnValue*$_{out}$ is the UI_LIST reference. Returning the reference to the list allows chaining of the **operator–** overload operator.

- *element*$_{in}$ is a pointer to the UI_ELEMENT class element or to the object derived from the UI_ELEMENT base class that is to be removed from the list.

NOTE: The **Subtract( )** function and the **- operator** are also used by windows, by the Window Manager and by the Event Manager. The UIW_WINDOW class removes a window object from its list by using **Subtract( )** or **- operator**. UI_WINDOW_-MANAGER uses **Subtract( )** or the **- operator** to remove a window object from the Window Manager. UI_EVENT_MANAGER uses these functions to remove a device from the list of devices.

Example 1

```
#include <ui_gen.hpp>

ExampleFunction1(UI_ELEMENT *element1)
{
    // Construct a list, then add elements to it.
    UI_LIST list1;
    list1.Add(element1);
    .
    .
    .

    // Delete a particular element from a list.
    list1.Subtract(element1);
    delete element1;
    .
    .
    .

}
```

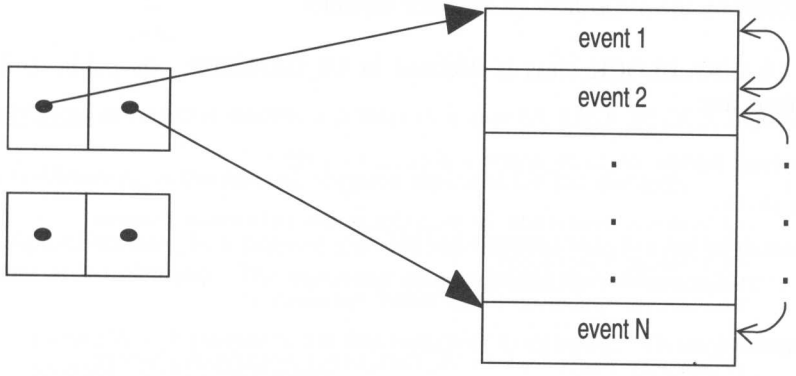
Example 2

```
ExampleFunction2(UI_ELEMENT *element1, UI_ELEMENT *element2)
{
    // Construct a list, then add elements to it using the
    // + operator overload.
    UI_LIST *list1 = new UI_LIST;
    *list1 + element1 + element2;
    .
    .
    .

    // Move elements from list1 to list2.
    UI_LIST *list2 = new UI_LIST;

    while (list1->First())
    {
        UI_ELEMENT *element = list1->First();
        *list1 - element;
        *list2 + element;
    }
    .
    .
    .

}
```

CHAPTER 19 – UI_LIST_BLOCK

The UI_LIST_BLOCK class is used to represent an array of items in list form. It is an advanced class and in general should not be used by programmers.

Since Zinc Application Framework uses lists and list elements exclusively, a UI_LIST_-BLOCK array is structured to behave like a list so that it can access pre-existing Zinc Application Framework functions. This is accomplished by establishing two pointers, one for the normal list and another for what is called the free list. When a list block is initialized, an array of items is created, and the beginning of the free list points to the first element in the array. Each of the elements in the array is derived from the UI_ELEMENT base class; therefore, each has a *previous* and a *next* pointer. The figure below illustrates this arrangement:



For example, the Event Manager uses an array of event elements to store input information. This array is essentially a block of UI_EVENT structures. In the case of UI_EVENT, however, which is not derived from UI_ELEMENT, a Q_ELEMENT class is constructed that is derived from UI_ELEMENT:

```
class EXPORT UI_QUEUE_ELEMENT : public UI_ELEMENT
{
public:
    // Members described in UI_QUEUE_ELEMENT reference chapter.
    UI_QUEUE_ELEMENT(void);
    UI_EVENT event;

    // Members described in UI_ELEMENT reference chapter.
    UI_QUEUE_ELEMENT *Next(void);
    UI_QUEUE_ELEMENT *Previous(void);
};
```

This class is actually only an event with a *previous* and a *next* pointer, which allows an array to be set up that behaves like a list.

Every time that a list is added to or a new list is created, instead of having to allocate memory for it, the space is taken from the array. Originally the free list points to the first element in the array, but if the array/list needs a new element, one will be taken from the free list and put into the array. Now the normal list points to the first element of the array and the free list points to the second. When elements are removed, instead of memory being made free, space will be opened up in the free list.

Constructing arrays that act as lists allows for greater speed and efficiency within Zinc Application Framework. For example, the Event Manager is continually feeding information into the system. If space had to be allocated for each of these events, the application would be extremely inefficient. Instead, a large block of memory is initially allocated by the Event Manager constructor (i.e., the size of the block allocated is controlled by the *noOfElements* member variable).

The UI_LIST_BLOCK class is declared in **UI_GEN.HPP**. Its public and protected members are:

```
class EXPORT UI_LIST_BLOCK : public UI_LIST
{
public:
    // Members described in UI_LIST_BLOCK reference chapter.
    virtual ~UI_LIST_BLOCK(void);
    UI_ELEMENT *Add(void);
    UI_ELEMENT *Add(UI_ELEMENT *element);
    UI_ELEMENT *Subtract(UI_ELEMENT *element);

protected:
    // Members described in UI_LIST_BLOCK reference chapter.
    int noOfElements;
    void *elementArray;
    UI_LIST freeList;

    UI_LIST_BLOCK(int noOfElements,
        int (*compareFunction)(void *element1, void *element2) = NULL);
};
```

- *noOfElements* is an integer that indicates how many elements are in the list. This must be a static number.

- *elementArray* is a pointer to the allocated block. This array must be allocated by the programmer through a derived class of UI_LIST_BLOCK.

- *freeList* contains pointers to the elements not currently in use.

UI_LIST_BLOCK::UI_LIST_BLOCK

Syntax

#include <ui_gen.hpp>

UI_LIST_BLOCK(int *noOfElements*,
int (**compareFunction*)(void **element1*, void **element2*) = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced constructor returns a pointer to a new UI_LIST_BLOCK object.

- $noOfElements_{in}$ is the amount of space allocated for list elements.
- $compareFunction_{in}$ is a programmer defined function that is used to determine the order of list elements. The following arguments are passed to *compare*:

  $element1_{in}$—A pointer to the first argument to compare. This argument must be typecast by the programmer.

  $element2_{in}$—A pointer to the second argument to compare. This argument must be typecast by the programmer.

The compare function's *returnValue* should be 0 if the two elements exactly match. A negative value should be returned if *element1* is less than *element2*. Otherwise, a positive value should be returned if *element1* is greater than *element2*.

Example

```
#include <ui_evt.hpp>

UI_QUEUE_BLOCK::UI_QUEUE_BLOCK(int _noOfElements) :
    UI_LIST_BLOCK(_noOfElements)
{
    // Initialize the queue block.
    UI_QUEUE_ELEMENT *queueBlock = new UI_QUEUE_ELEMENT[_noOfElements];
```

```
    elementArray = queueBlock;
    for (int i = 0; i < _noOfElements; i++)
        freeList.Add(NULL, &queueBlock[i]);
}
```

UI_LIST_BLOCK::~UI_LIST_BLOCK

Syntax

#include <ui_gen.hpp>

virtual ~UI_LIST_BLOCK(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_LIST_-BLOCK object. It calls the destructor associated with each element in the list.

Example

```
#include <ui_evt.hpp>

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

UI_LIST_BLOCK::Add

Syntax

#include <ui_gen.hpp>

UI_ELEMENT *Add(void);

or

UI_ELEMENT *Add(UI_ELEMENT *element*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded functions are used to add a new element to the UI_LIST_BLOCK object.

The first overloaded function adds a new element, retrieved from the free list, to the UI_LIST_BLOCK object into a position specified by the list's *compareFunction*. If no compare function is specified when the list is constructed, the element is added to the end of the UI_LIST_BLOCK object. The new element is retrieved from the free list.

The second overloaded function overrides the list's *compareFunction* by inserting the new element directly before *element*.

- *returnValue* is a pointer to the new element if the addition was successful. Otherwise, *returnValue* is NULL.

- *element* is a pointer to an element before which the new element is to be placed. If this variable is NULL, the routine adds the new element to the end of the list.

Example

```
#include "ui_evt.hpp"

void UI_EVENT_MANAGER::Put(const UI_EVENT &event, Q_FLAGS flags)
{
    // Place the event back in the event queue.
    UI_QUEUE_ELEMENT *element =
        (UI_QUEUE_ELEMENT *)queueBlock.Add(FlagSet(flags, Q_END) ? NULL :
        queueBlock.First());
    if (element)
        element->event = event;
}
```

UI_LIST_BLOCK::Subtract

Syntax

#include <ui_gen.hpp>

UI_ELEMENT *Subtract(UI_ELEMENT **element*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function removes an element from the UI_LIST_BLOCK object and puts it back into the free list. This routine does not call the destructor associated with the element.

- *returnValue*$_{out}$ is a pointer to the next element in the list. This value is NULL if there are no more elements after the removed element.
- *element*$_{in}$ is a pointer to the element to be removed from the list.

Example

```
#include "ui_evt.hpp"

int UI_EVENT_MANAGER::Get(UI_EVENT &event, Q_FLAGS flags)
{
    UI_DEVICE *device;
    UI_QUEUE_ELEMENT *element;
    int error = -1;

    // Stay in loop while no event conditions are met.
    do
    {
        // Call all the polled devices.
        if (!FlagSet(flags, Q_NO_POLL))
        {
            .
            .
            .
            for (device = First(); device; device = device->Next())
                device->Poll();
        }
```

```
        // Get the event.
        element = FlagSet(flags, Q_END) ? queueBlock.Last() :
            queueBlock.First();
        if (element)
        {
            event = element->event;
            if (!FlagSet(flags, Q_NO_DESTROY))
                queueBlock.Subtract((UI_ELEMENT *)element);
            error = 0;
        }
        else if (FlagSet(flags, Q_NO_BLOCK))
            return (-2);
    } while (error);

    // Return the error status.
    return (error);
}
```

CHAPTER 20 – UI_MOTIF_DISPLAY

The UI_MOTIF_DISPLAY class implements a graphics display that uses the OSF/Motif Toolkit version 1.1 with X11.R4 to display information to the screen. Since the UI_-MOTIF_DISPLAY class is derived from UI_DISPLAY, only details specific to the UI_-MOTIF_DISPLAY class are given in this chapter. For descriptions and examples regarding virtual or inherited display members, see "Chapter 6—UI_DISPLAY."

Applications using the UI_MOTIF_DISPLAY class are true X Window, OSF/Motif programs. Therefore, the following features apply:

- The X resource database is used to specify such resources as default colors, widget fonts, etc. While the default application class name is "ZincApp", users can create their own class names and files that specify the application defaults. The X foreground and background resources currently override *UI_WINDOW_-OBJECT::paletteMapTable*.

- The UI_MOTIF_DISPLAY assumes that the Motif Window Manager is the current window manager. If another X window manager is used, such as the OPEN LOOK, the program should still work, but decorations such as UIW_SYSTEM_BUTTON and UIW_TITLE might not behave as documented.

- The Motif widget name for each Zinc window object is the *UI_WINDOW_-OBJECT::stringID*. These widget names can be used in resource files.

- Programs are Motif programs first and Zinc Application Framework programs second. Where conflicts exist between the definitions, Zinc has tried to adhere to the Motif Style Guide.

NOTE: All member functions use the standard Zinc screen pixel coordinates with (0,0) being the top-left corner of the display.

The UI_MOTIF_DISPLAY class is declared in **UI_DSP.HPP**. Its public members are:

```
class EXPORT UI_MOTIF_DISPLAY : public UI_DISPLAY
{
public:
    struct MOTIFFONT
    {
        XFontStruct *fontStruct;
        XmFontList fontList;
    };

    static MOTIFFONT fontTable[MAX_LOGICAL_FONTS];
```

```
    UI_MOTIF_DISPLAY(int *argc = NULL, char **argv = NULL,
        char *appClass = "ZincApp");
    virtual ~UI_MOTIF_DISPLAY(void);
    virtual void Bitmap(SCREENID screenID, int column, int line,
        int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette = NULL,
        const UI_REGION *clipRegion = NULL, HBITMAP *colorBitmap = NULL,
        HBITMAP *monoBitmap = NULL);
    virtual void BitmapArrayToHandle(SCREENID screenID, int bitmapWidth,
        int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette, HBITMAP *colorBitmap,
        HBITMAP *monoBitmap);
    virtual void BitmapHandleToArray(SCREENID screenID, HBITMAP colorBitmap,
        HBITMAP monoBitmap, int *bitmapWidth, int *bitmapHeight,
        UCHAR **bitmapArray);
    virtual void Ellipse(SCREENID screenID, int column, int line,
        int startAngle, int endAngle, int xRadius, int yRadius,
        const UI_PALETTE *palette, int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void IconArrayToHandle(SCREENID screenID, int iconWidth,
        int iconHeight, const UCHAR *iconArray, const UI_PALETTE *palette,
        HICON *icon);
    virtual void IconHandleToArray(SCREENID screenID, HICON icon,
        int *iconWidth, int *iconHeight, UCHAR **iconArray);
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground);
    virtual void Polygon(SCREENID screenID, int numPoints,
        const int *polygonPoints, const UI_PALETTE *palette,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion, SCREENID screenID = ID_SCREEN);
    virtual void RegionDefine(SCREENID screenID, int left, int top,
        int right, int bottom);
    virtual void RegionMove(const UI_REGION &oldRegion, int newColumn,
        int newLine, SCREENID oldScreenID = ID_SCREEN,
        SCREENID newScreenID = ID_SCREEN);
    virtual void Text(SCREENID screenID, int left, int top,
        const char *text, const UI_PALETTE *palette, int length = -1,
        int fill = TRUE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN, LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextWidth(const char *string, SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
        int bottom);
    virtual int VirtualPut(SCREENID screenID);

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

};
```

MOTIFFONT is a structure that contains the following font information:

fontStruct is a pointer to the X font structure, **XFontStruct**.

fontList is a list of Motif fonts created from *fontStruct*.

- *fontTable* is an array of font handles for Motif. The following entries are pre-defined by Zinc:

  FNT_SMALL_FONT—is a font that is used to display an icon's text string.

  FNT_DIALOG_FONT—is a font that is used when text is displayed on window objects (e.g., UIW_BUTTON, UIW_STRING, UIW_TEXT, etc.)

  FNT_SYSTEM_FONT—is a sans-serif style font that is used to display a window's title.

UI_MOTIF_DISPLAY::UI_MOTIF_DISPLAY

Syntax

```
#include <ui_dsp.hpp>

UI_MOTIF_DISPLAY(int *argc = NULL, char **argv = NULL,
    char *appClass = "ZincApp");
```

Portability

This function is available on the following environments:

☐ DOS ☐ MS Windows ☐ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UI_MOTIF_DISPLAY class object.

- $argc_{in}$ is a pointer to an integer containing the number of arguments passed to function **main( )**. *argc* points to a 1 if only the program was invoked with no command-line arguments. If the program were invoked with 1 command-line argument, *argc* would point to a 2, etc. *argc* is used to determine the number of parameters contained in *argv*. This parameter is passed to **XtAppInitialize( )**.

- *argv*$_{in}$ is a pointer to an array of character strings that contain the actual command-line parameters. For example, if the program TEST.EXE were invoked with a /C switch, *argv[0]* would point to "TEST.EXE" and *argv[1]* would point to "/C". This parameter is passed to **XtAppInitialize( )**.

- *appClass*$_{in}$ is a pointer to a character string denoting the class name of application being executed. This parameter is passed to **XtAppInitialize( )**.

Example

```
#include <ui_win.hpp>

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

    // Initialize the event manager.
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);

    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;

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

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

CHAPTER 21 – UI_MSC_DISPLAY

The UI_MSC_DISPLAY class implements a graphics display that uses the Microsoft C/C++ graphics library to display information to the screen. Since the UI_MSC_-DISPLAY class is derived from UI_DISPLAY, only details specific to the UI_MSC_-DISPLAY class are given in this chapter. For descriptions and examples regarding virtual or inherited display members, see "Chapter 6—UI_DISPLAY."

The UI_MSC_DISPLAY class is declared in **UI_DSP.HPP**. Its public and protected members are:

```
class EXPORT UI_MSC_DISPLAY : public UI_DISPLAY, public UI_REGION_LIST
{
public:
    struct EXPORT MSCFONT
    {
        char *typeFace;
        char *options;
    };
    typedef unsigned char MSCPATTERN[8];

    static UI_PATH *searchPath;
    static MSCFONT fontTable[MAX_LOGICAL_FONTS];
    static MSCPATTERN patternTable[MAX_LOGICAL_PATTERNS];

    UI_MSC_DISPLAY(int mode = 0);
    virtual ~UI_MSC_DISPLAY(void);
    virtual void Bitmap(SCREENID screenID, int column, int line,
        int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette = NULL,
        const UI_REGION *clipRegion = NULL, HBITMAP *colorBitmap = NULL,
        HBITMAP *monoBitmap = NULL);
    virtual void BitmapArrayToHandle(SCREENID screenID, int bitmapWidth,
        int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette, HBITMAP *colorBitmap,
        HBITMAP *monoBitmap);
    virtual void BitmapHandleToArray(SCREENID screenID, HBITMAP colorBitmap,
        HBITMAP monoBitmap, int *bitmapWidth, int *bitmapHeight,
        UCHAR **bitmapArray);
    virtual void Ellipse(SCREENID screenID, int column, int line,
        int startAngle, int endAngle, int xRadius, int yRadius,
        const UI_PALETTE *palette, int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void IconArrayToHandle(SCREENID screenID, int iconWidth,
        int iconHeight, const UCHAR *iconArray, const UI_PALETTE *palette,
        HICON *icon);
    virtual void IconHandleToArray(SCREENID screenID, HICON icon,
        int *iconWidth, int *iconHeight, UCHAR **iconArray);
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground);
    virtual void Polygon(SCREENID screenID, int numPoints,
        const int *polygonPoints, const UI_PALETTE *palette,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
```

```
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion, SCREENID screenID = ID_SCREEN);
    virtual void RegionDefine(SCREENID screenID, int left, int top,
        int right, int bottom);
    virtual void RegionMove(const UI_REGION &oldRegion, int newColumn,
        int newLine, SCREENID oldScreenID = ID_SCREEN,
        SCREENID newScreenID = ID_SCREEN);
    virtual void Text(SCREENID screenID, int left, int top,
        const char *text, const UI_PALETTE *palette, int length = -1,
        int fill = TRUE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN, LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextWidth(const char *string, SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
        int bottom);
    virtual int VirtualPut(SCREENID screenID);

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

protected:
    int maxColors;
    char *_virtualCount;
    UI_REGION _virtualRegion;
    char _stopDevice;
    int _fillPattern;
    int _backgroundColor;
    int _foregroundColor;

    void SetFont(LOGICAL_FONT logicalFont);
    void SetPattern(const UI_PALETTE *palette, int xor);
};
```

- *MSC_FONT* is a structure that contains the following font information:

  typeFace contains the font name (e.g., in string format). FNT_SMALL_FONT, FNT_DIALOG_FONT and FNT_SYSTEM_FONT are pre-defined by Zinc.

  options contains the font characteristics. For more information see **_setfont( )** in the *Microsoft C/C++ Library Reference*.

- *MSCPATTERN* is an array of 8 bytes that make-up the 8x8 bitmap pattern. Each byte (8 bits) corresponds to 8 pixels in the pattern. The patterns defined by Zinc are: PTN_SOLID_FILL, PTN_INTERLEAVE_FILL and PTN_BACKGROUND_FILL. For more information see **setfillpattern( )** in the *Microsoft C++ Library Reference*.

- *searchPath* contains the path to be searched for the graphics and font files.

- *fontTable* is an array of *MSC_FONT*. The default array contains space for 10 *MSC_FONT* entries. The following entries are pre-defined by Zinc:

FNT_SMALL_FONT—is a font that is used to display an icon's text string.

FNT_DIALOG_FONT—is a font that is used when text is displayed on window objects (e.g., UIW_BUTTON, UIW_STRING, UIW_TEXT, etc.)

FNT_SYSTEM_FONT—is a sans-serif style font that is used to display a window's title.

NOTE: To use these fonts, or if other "stroked" fonts are added to this table, the proper font files must be in the current path at run-time.

- *patternTable* is an array of *MSCPATTERN*. The default array contains space for 15 *MSCPATTERN* entries. The following entries are pre-defined by Zinc:

  PTN_SOLID_FILL—Solid fill.

  PTN_INTERLEAVE_FILL—Interleaving line fill.

  PTN_BACKGROUND_FILL—Background fill style.

- *maxColors* tells the maximum number of colors supported by the display. For example, an EGA display supports sixteen colors.

- *_virtualCount* is a count of the number of virtual screen operations that have taken place. For example, when the **VirtualGet( )** function is called, *_virtualCount* is decremented. Additionally, when the **VirtualPut( )** function is called, *_virtualCount* is incremented.

- *_virtualRegion* is the region affected by either **VirtualGet( )** or **VirtualPut( )**.

- *_stopDevice* is a variable used to disable the update of the display. If *_stopDevice* is TRUE, no drawing will be done to the screen. Otherwise, drawing will be made directly to the screen display.

- *_fillPattern* is an index into the *patternTable* specifying the current fill pattern.

- *_backgroundColor* is the current background drawing color.

- *_foregroundColor* is the current foreground drawing color.

UI_MSC_DISPLAY::UI_MSC_DISPLAY

Syntax

#include <ui_dsp.hpp>

UI_MSC_DISPLAY(int *mode* = 0);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This constructor returns a pointer to a new UI_MSC_DISPLAY object. When a new UI_MSC_DISPLAY class is constructed, the screen display is set to the background color and pattern specified by the inherited variable *backgroundPalette*.

- $mode_{in}$ is an argument passed to the Microsoft **_setvideomode( )** function. The argument *mode* specifies the initial graphics mode. For more information on these arguments see **_setvideomode( )** in the *Microsoft C Run-Time Library Reference*.

Example

```
#include <ui_win.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_MSC_DISPLAY;
    .
    .
    .

    // Restore the system.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

CHAPTER 22 – UI_MSWINDOWS_DISPLAY

The UI_MSWINDOWS_DISPLAY class implements a graphics display that uses the Microsoft Windows graphics package to display information to the screen. Since the UI_MSWINDOWS_DISPLAY class is derived from UI_DISPLAY, only details specific to the UI_MSWINDOWS_DISPLAY class are given in this chapter. For descriptions and examples regarding virtual or inherited display members, see "Chapter 6—UI_-DISPLAY."

The UI_MSWINDOWS_DISPLAY class is declared in **UI_DSP.HPP**. Its public members are:

```
class EXPORT UI_MSWINDOWS_DISPLAY : public UI_DISPLAY
{
public:
    static HDC hDC;
    static HFONT fontTable[MAX_LOGICAL_FONTS];
    static WORD patternTable[MAX_LOGICAL_PATTERNS][8];

    UI_MSWINDOWS_DISPLAY(HANDLE hInstance, HANDLE hPrevInstance,
        int nCmdShow);
    virtual ~UI_MSWINDOWS_DISPLAY(void);
    virtual void Bitmap(SCREENID screenID, int column, int line,
        int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette = NULL,
        const UI_REGION *clipRegion = NULL, HBITMAP *colorBitmap = NULL,
        HBITMAP *monoBitmap = NULL);
    virtual void BitmapArrayToHandle(SCREENID screenID, int bitmapWidth,
        int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette, HBITMAP *colorBitmap,
        HBITMAP *monoBitmap);
    virtual void BitmapHandleToArray(SCREENID screenID, HBITMAP colorBitmap,
        HBITMAP monoBitmap, int *bitmapWidth, int *bitmapHeight,
        UCHAR **bitmapArray);
    virtual void Ellipse(SCREENID screenID, int column, int line,
        int startAngle, int endAngle, int xRadius, int yRadius,
        const UI_PALETTE *palette, int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void IconArrayToHandle(SCREENID screenID, int iconWidth,
        int iconHeight, const UCHAR *iconArray, const UI_PALETTE *palette,
        HICON *icon);
    virtual void IconHandleToArray(SCREENID screenID, HICON icon,
        int *iconWidth, int *iconHeight, UCHAR **iconArray);
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground);
    virtual void Polygon(SCREENID screenID, int numPoints,
        const int *polygonPoints, const UI_PALETTE *palette,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion, SCREENID screenID = ID_SCREEN);
    virtual void RegionDefine(SCREENID screenID, int left, int top,
        int right, int bottom);
```

```
    virtual void RegionMove(const UI_REGION &oldRegion, int newColumn,
        int newLine, SCREENID oldScreenID = ID_SCREEN,
        SCREENID newScreenID = ID_SCREEN);
    virtual void Text(SCREENID screenID, int left, int top,
        const char *text, const UI_PALETTE *palette, int length = -1,
        int fill = TRUE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN, LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextWidth(const char *string, SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
        int bottom);
    virtual int VirtualPut(SCREENID screenID);

protected:
    int maxColors;
};
```

- *hDC* is a handle to the current display context. Programmers should not use this member, it is for internal use only.

- *fontTable* is an array of font handles for Microsoft Windows. The following entries are pre-defined by Zinc:

  FNT_SMALL_FONT—is a font that is used to display an icon's text string.

  FNT_DIALOG_FONT—is a font that is used when text is displayed on window objects (e.g., UIW_BUTTON, UIW_STRING, UIW_TEXT, etc.)

  FNT_SYSTEM_FONT—is a sans-serif style font that is used to display a window's title.

  NOTE: When these three fonts are used, no font files are needed since they are linked into Zinc Application Framework. However, if other "stroked" fonts are added to this table, the proper font files must either be in the current path or be linked into the application.

- *patternTable* is an array containing space for 15 pattern entries. The following entries are pre-defined by Zinc:

  PTN_SOLID_FILL—Solid fill.

  PTN_INTERLEAVE_FILL—Interleaving line fill.

  PTN_BACKGROUND_FILL—Background fill style.

- *maxColors* tells the maximum number of colors supported by the display. For example, an EGA display supports sixteen colors.

UI_MSWINDOWS_DISPLAY::UI_MSWINDOWS_DISPLAY

Syntax

#include <ui_dsp.hpp>

UI_MSWINDOWS_DISPLAY(HANDLE *hInstance*, HANDLE *hPrevInstance*,
int *nCmdShow*);

Portability

This function is available on the following environments:

☐ DOS ■ MS Windows ☐ OS/2 ☐ Motif

Remarks

This constructor returns a pointer to a new UI_MSWINDOWS_DISPLAY class object.

- $hInstance_{in}$ is the particular instance under which the application is running. For example, if an application is run twice, there are two instances of that application. This value is passed in automatically by **WinMain( )**.

- $hPrevInstance_{in}$ is the previous instance of the application. If a program is run for the first time, *hPrevInstance* is 0. This value is passed in automatically by **WinMain( )**.

- $nCmdShow_{in}$ is a string containing the command line parameters.

Example

```
#include <ui_win.hpp>

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

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

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

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

CHAPTER 23 – UI_OS2_DISPLAY

The UI_OS2_DISPLAY class implements a graphics display that uses the OS/2 graphics package to display information to the screen. Since the UI_OS2_DISPLAY class is derived from UI_DISPLAY, only details specific to the UI_OS2_DISPLAY class are given in this chapter. For descriptions and examples regarding virtual or inherited display members, see "Chapter 6—UI_DISPLAY."

The UI_OS2_DISPLAY class is declared in **UI_DSP.HPP**. Its public members are:

```
class EXPORT UI_OS2_DISPLAY : public UI_DISPLAY
{
public:
    static HPS hps;
    static FONTMETRICS fontTable[MAX_LOGICAL_FONTS];

    UI_OS2_DISPLAY(void);
    virtual ~UI_OS2_DISPLAY(void);
    virtual void Bitmap(SCREENID screenID, int column, int line,
        int bitmapWidth, int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette = NULL,
        const UI_REGION *clipRegion = NULL, HBITMAP *colorBitmap = NULL,
        HBITMAP *monoBitmap = NULL);
    virtual void BitmapArrayToHandle(SCREENID screenID, int bitmapWidth,
        int bitmapHeight, const UCHAR *bitmapArray,
        const UI_PALETTE *palette, HBITMAP *colorBitmap,
        HBITMAP *monoBitmap);
    virtual void BitmapHandleToArray(SCREENID screenID, HBITMAP colorBitmap,
        HBITMAP monoBitmap, int *bitmapWidth, int *bitmapHeight,
        UCHAR **bitmapArray);
    virtual void Ellipse(SCREENID screenID, int column, int line,
        int startAngle, int endAngle, int xRadius, int yRadius,
        const UI_PALETTE *palette, int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void IconArrayToHandle(SCREENID screenID, int iconWidth,
        int iconHeight, const UCHAR *iconArray, const UI_PALETTE *palette,
        HICON *icon);
    virtual void IconHandleToArray(SCREENID screenID, HICON icon,
        int *iconWidth, int *iconHeight, UCHAR **iconArray);
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground);
    virtual void Polygon(SCREENID screenID, int numPoints,
        const int *polygonPoints, const UI_PALETTE *palette,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion, SCREENID screenID = ID_SCREEN);
    virtual void RegionDefine(SCREENID screenID, int left, int top,
        int right, int bottom);
    virtual void RegionMove(const UI_REGION &oldRegion, int newColumn,
        int newLine, SCREENID oldScreenID = ID_SCREEN,
        SCREENID newScreenID = ID_SCREEN);
```

```
    virtual void Text(SCREENID screenID, int left, int top,
        const char *text, const UI_PALETTE *palette, int length = -1,
        int fill = TRUE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN, LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextWidth(const char *string, SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
        int bottom);
    virtual int VirtualPut(SCREENID screenID);

protected:
    int maxColors;
};
```

- *hps* is a handle to the current display context. Programmers should not use this member, it is for internal use only.

- *fontTable* is an array of character strings that serve as font handles for OS/2. The following entries are pre-defined by Zinc:

  FNT_SMALL_FONT—is a font that is used to display an icon's text string.

  FNT_DIALOG_FONT—is a font that is used when text is displayed on window objects (e.g., UIW_BUTTON, UIW_STRING, UIW_TEXT, etc.)

  FNT_SYSTEM_FONT—is a sans-serif style font that is used to display a window's title.

- *maxColors* tells the maximum number of colors supported by the display. For example, an EGA display supports sixteen colors.

NOTE: All member functions use the standard Zinc screen pixel coordinates with (0,0) being the top-left corner of the display.

UI_OS2_DISPLAY::UI_OS2_DISPLAY

Syntax

```
#include <ui_dsp.hpp>

UI_OS2_DISPLAY(void);
```

Portability

This function is available on the following environments:

☐ DOS ☐ MS Windows ■ OS/2 ☐ Motif

Remarks

This constructor returns a pointer to a new UI_OS2_DISPLAY class object.

Example

```
#include <ui_win.hpp>

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

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

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

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

CHAPTER 24 – UI_PALETTE

The UI_PALETTE structure is used by Zinc Application Framework class objects for color information.

The UI_PALETTE structure is the same for all environments. Since Windows has its own definitions for color and fill patterns, changing the UI_PALETTE structure will not change the Windows palette information. The definition is contained in **UI_DSP.HPP**. Its fields are:

```
struct EXPORT UI_PALETTE
{
    // Members described in UI_PALETTE reference chapter.
    // --- Text mode ---
    UCHAR fillCharacter;              // Fill character.
    COLOR colorAttribute;             // Color attribute.
    COLOR monoAttribute;              // Mono attribute.

    // --- Graphics mode ---
    LOGICAL_PATTERN fillPattern;      // Fill pattern.
    COLOR colorForeground;            // EGA/VGA colors.
    COLOR colorBackground;
    COLOR bwForeground;               // Black & White colors (2 color).
    COLOR bwBackground;
    COLOR grayScaleForeground;        // Monochrome colors (3+ color).
    COLOR grayScaleBackground;
};
```

- *fillCharacter* 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.

- *colorAttribute* is the attribute of the foreground and background colors respectively for color text display mode.

- *monoAttribute* is the attribute of the foreground and background colors respectively for monochrome text display mode.

- *fillPattern* 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.

```
PTN_SOLID_FILL          0x0000      Solid Fill
PTN_INTERLEAVE_FILL     0x0001      Interleaving line fill
PTN_BACKGROUND_FILL     0x0002      Fill with background color
PTN_SYSTEM_COLOR        0x00F0      Fill with the system color (Motif,
                                    OS/2 and Windows only)
PTN_RGB_COLOR           0x00F1      Use red, green, blue values (Motif,
                                    OS/2 and Windows only)
```

- *colorForeground* and *colorBackground* are the attributes of the foreground and background colors respectively for VGA, VGA monochrome and EGA graphics display modes.
- *bwForeground* and *bwBackground* are the attributes of the foreground and background colors respectively for CGA and Hercules graphics display modes.
- *grayScaleForeground* and *grayScaleBackground* are the attributes of the foreground and background colors respectively for EGA monochrome graphics display mode.

NOTE: The **attrib( )** macro is used to combine text color and monochrome attribute pairs (the *colorAttribute* and *monoAttribute* variables described above.

Portability

This structure is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Example

```
#include <ui_dsp.hpp>

static UI_PALETTE backgroundPalette = {
    '\260', attrib(BLUE, BLACK), attrib(MONO_DIM, MONO_BLACK),
    PTN_INTERLEAVE_FILL, BLUE, BLUE, BW_WHITE, BW_WHITE, GS_GRAY, GS_GRAY };

main( )
{
    .
    .
    .
    display->backgroundPalette = &backgroundPalette;

    .
    .
    .

}
```

CHAPTER 25 – UI_PALETTE_MAP

The UI_PALETTE_MAP structure is used by Zinc Application Framework class objects for color map information.

The UI_PALETTE_MAP structure is declared in **UI_WIN.HPP**. Its fields are:

```
struct EXPORT UI_PALETTE_MAP
{
    // Members described in UI_PALETTE_MAP reference chapter.
    OBJECTID objectID;
    LOGICAL_PALETTE logicalPalette;
    UI_PALETTE palette;

    static UI_PALETTE *MapPalette(UI_PALETTE_MAP *mapTable,
        LOGICAL_PALETTE logicalPalette, OBJECTID id1 = ID_WINDOW_OBJECT,
        OBJECTID id2 = ID_WINDOW_OBJECT, OBJECTID id3 = ID_WINDOW_OBJECT,
        OBJECTID id4 = ID_WINDOW_OBJECT, OBJECTID id5 = ID_WINDOW_OBJECT);
};
```

- *objectID* is the object identification for which the match applies. (A full list of window identifications is given in **UI_WIN.HPP**.) Each window identification has an "ID_" prefix. Some example window object identifications are:

  ID_WINDOW_OBJECT—Associated with all class objects derived from the UI_WINDOW_OBJECT base class.

  ID_BORDER—Associated with the UIW_BORDER class object.

  ID_STRING—Associated with the UIW_STRING object or with any class object derived from the UIW_STRING base class (e.g., UIW_DATE, UIW_TIME).

- *logicalPalette* is the logical palette to map. The following logical palette identifications (defined in **UI_WIN.HPP**) are recognized: PM_ACTIVE, PM_-CURRENT, PM_HOTKEY, PM_INACTIVE, PM_NON_SELECTABLE, PM_-SELECTED, PM_SPECIAL and PM_ANY (this map is used if no other color map can be found). In addition, the PM_HOT_KEY is a special map identification used when a hot key is being drawn on the screen.

  The following algorithm is used to determine which palette map identification is used:

  1—If the object is not selectable (i.e., the object's *woFlags* variable has the WOF_NON_SELECTABLE flag set), then the **PM_NON_SELECTABLE** color map is used.

2—If the object is in an inactive state (i.e., it is not the current object on the screen or its parent window is not the front window on the screen), then the **PM_INACTIVE** color map is used.

3—If the object has been selected (i.e., the object's *woStatus* variable has the WOS_SELECTED status set), the **PM_SELECTED** color map is used.

4—If the object is not current but its parent window is at the front of the screen, then the **PM_ACTIVE** color map is used.

5—If the object is current and its parent window is the front window on the screen, then the **PM_CURRENT** color map is used.

6—If all previous cases are not true, then the color map default is **PM_ANY**.

- *palette* is the palette associated with the logical value.

UI_PALETTE_MAP::MapPalette

Syntax

```
#include <ui_map.hpp>

static UI_PALETTE *MapPalette(UI_PALETTE_MAP *mapTable,
    LOGICAL_PALETTE logicalPalette, OBJECTID id1 = ID_WINDOW_OBJECT,
    OBJECTID id2 = ID_WINDOW_OBJECT,
    OBJECTID id3 = ID_WINDOW_OBJECT,
    OBJECTID id4 = ID_WINDOW_OBJECT,
    OBJECTID id5 = ID_WINDOW_OBJECT);
```

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function provides the logical mapping (if any) of a palette.

- *returnValue*$_{out}$ is the logical palette that matches the logical value and identification parameters.

- *mapTable*$_{in}$ is a pointer to the palette map table to be used by the palette mapping function.

- *logicalPalette*$_{in}$ is the logical palette to be mapped. The following logical palette identifications (defined in **UI_WIN.HPP**) are recognized: PM_ACTIVE, PM_-CURRENT, PM_INACTIVE, PM_INVALID, PM_NON_SELECTABLE, PM_-SELECTED and PM_ANY (this map is used if no other color map can be found). In addition, the PM_HOT_KEY is a special map identification used when a hot key is being drawn on the screen.

- *id1*$_{in}$, *id2*$_{in}$, *id3*$_{in}$, *id4*$_{in}$ and *id5*$_{in}$ are hierarchal identification values to use while interpreting the palette. For example, the UIW_TEXT class object uses the following identification values when it looks for a logical mapping:

  id1—ID_TEXT
  id2—ID_WINDOW
  id3—ID_WINDOW_OBJECT
  id4—unused
  id5—unused

Example

```
#include <ui_win.hpp>
EVENT_TYPE UIW_BORDER::DrawItem(const UI_EVENT &event, EVENT_TYPE ccode)
{
    .
    .
    .

    // Draw the border.
    int size = display->cellHeight;
    UI_REGION region = parent->true;
    UI_PALETTE *outline = LogicalPalette(ccode, ID_OUTLINE);
    display->Rectangle(screenID, region, outline, 1, FALSE, FALSE, &clip);
    display->Rectangle(screenID, true, outline, 1, FALSE, FALSE, &clip);
    --region;
    if (ccode == S_DISPLAY_ACTIVE && FlagSet(parent->woAdvancedFlags,
        WOAF_DIALOG_OBJECT))
         lastPalette = UI_PALETTE_MAP::MapPalette(paletteMapTable, PM_SELECTED,
            ID_BORDER);
    .
    .
    .

    return (ccode);
}
```

CHAPTER 26 – UI_PATH

The UI_PATH class is used to store doubly-linked list elements derived from the UI_PATH_ELEMENT class. The UI_PATH class is used to access information about the location of files, how to access files and how to access the originating point of a program. UI_PATH is particularly useful in conjunction with opening and closing files used within Zinc Application Framework. It ensures that all of the needed files are found and included, even if they are located in a path other than the current path.

The UI_PATH class is declared in **UI_GEN.HPP**. Its public members are:

```
class EXPORT UI_PATH : public UI_LIST
{
public:
    // Members described in UI_PATH reference chapter.
    UI_PATH(char *programPath = NULL, int rememberCWD = TRUE);
    ~UI_PATH(void);
    const char *FirstPathName(void);
    const char *NextPathName(void);

    // Members described in UI_LIST reference chapter.
    UI_PATH_ELEMENT *Current(void);
    UI_PATH_ELEMENT *First(void);
    UI_PATH_ELEMENT *Last(void);
};
```

UI_PATH::UI_PATH

Syntax

#include <ui_gen.hpp>

UI_PATH(char **programPath* = NULL, int *rememberCWD* = TRUE);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The constructor not only uses the environment variable "PATH" when searching for a specific file, but also searches the program path and the current working directory.

- *programPath*$_{in}$ contains a pointer to the place where the program originated.
- *rememberCWD*$_{in}$ ensures that the current working directory is included in the search of paths, if the argument is set to TRUE.

Example

```
#include <ui_gen.hpp>

main(int argc, char *argv[])
{
    // Initialize the path.
    UI_PATH *path = new UI_PATH(argv[0], TRUE);
    UI_DISPLAY *display = new UI_MSC_DISPLAY(0, 0, path);
    .
    .
    .

    delete display;
    delete path;
}
```

UI_PATH::~UI_PATH

Syntax

#include <ui_gen.hpp>

~UI_PATH(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This destructor destroys the class information associated with the UI_PATH object.

Example

```
#include <ui_gen.hpp>

main(int argc, char *argv[])
{
```

```
    // Initialize the path.
    UI_PATH *path = new UI_PATH(argv[0], TRUE);
    .
    .
    .

    delete path;
}
```

UI_PATH::FirstPathName

Syntax

#include <ui_gen.hpp>

const char *FirstPathName(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function gets the first path or directory to be searched for a file. If there is no first directory, it passes back NULL.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <ui_gen.hpp>

int ExampleFunction(const char *file, unsigned int mode)
{
    UI_PATH *path = new UI_PATH;
    .
    .
    .

    char *pathName;
    pathName = path->FirstPathName();
    if (pathName == NULL)
        return (-1);

    else
    {
        while ((pathName = path->NextPathName()) != 0)
```

```
            if ((handle = open(pathName, mode)) >= 0)
                return (handle);
        }
        return (-1);
    }
```

UI_PATH::NextPathName

Syntax

#include <ui_gen.hpp>

const char *NextPathName(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function gets the next path or directory to be searched for a file. If there is no next directory, it passes back NULL.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <ui_gen.hpp>

int ExampleFunction(const char *file, unsigned int mode)
{
    int handle;
    UI_PATH *path = new UI_PATH;
    .
    .
    .
    SetFileName(file);
    char *pathName;
    pathName = path->FirstPathName()
    while ((pathName = path->NextPathName()) != 0)
        if ((handle = open(pathName, mode)) >= 0)
            return (handle);
    return (-1);
}
```

CHAPTER 27 – UI_PATH_ELEMENT

The UI_PATH_ELEMENT class is used to store information about the location of files, how to access files and how to access the origination point of a program. UI_PATH_-ELEMENT is particularly useful in conjunction with opening and closing files used within Zinc Application Framework. It allows access to a particular path item such as a file or a directory entry.

The UI_PATH_ELEMENT class is declared in **UI_GEN.HPP**. Its public members are:

```
class EXPORT UI_PATH_ELEMENT : public UI_ELEMENT
{
public:
    // Members described in UI_PATH_ELEMENT reference chapter.
    char *pathName;

    UI_PATH_ELEMENT(char *pathName, int length = -1);
    ~UI_PATH_ELEMENT(void);

    // Members described in UI_ELEMENT reference chapter.
    UI_PATH_ELEMENT *Next(void);
    UI_PATH_ELEMENT *Previous(void);
};
```

- *pathName* is a string containing the directory to be searched. *pathName* may also contain drive specifiers.

UI_PATH_ELEMENT::UI_PATH_ELEMENT

Syntax

#include <ui_gen.hpp>

UI_PATH_ELEMENT(char *pathName, int length = -1);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a UI_PATH_ELEMENT object. The UI_PATH_-

ELEMENT is used to store a path name and is used in conjunction with the UI_PATH class.

- *pathName* contains the path or directory name to be searched.
- *length* is the length of *pathName*. If no value is entered for *length*, then the length of *pathName* is automatically computed. If a value is entered, the programmer must ensure that it is at least as long as *pathName*.

Example

```
#include <ui_gen.hpp>

UI_PATH::UI_PATH(char *programPath, int rememberCWD)
{
    // Get the path names.
    for (int i = 0; i < 3; i++)
    {
        // Determine which path to look for.
        char path[256];
        if (i == 0 && rememberCWD)
            getcwd(path, 256);
        else if (i == 1 && programPath)
            strcpy(path, programPath);
        else if (i == 2 && getenv("PATH"))
            strcpy(path, getenv("PATH"));
        else
            strcpy(path, "");

        // Parse the directory tree.
        for (int j = 0; path[j]; )
        {
            for (int k = j; path[k] && path[k] != ';'; )
                k++;
            Add(NULL, new UI_PATH_ELEMENT(&path[j], k - j));
            j = path[k] ? k + 1 : k;
        }
    }
}
```

UI_PATH_ELEMENT::~UI_PATH_ELEMENT

Syntax

#include <ui_gen.hpp>

~UI_PATH_ELEMENT(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This destructor destroys the class information associated with the UI_PATH_ELEMENT object.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    .
    .
    .

    path - pathElement;
    delete pathElement;
}
```

CHAPTER 28 – UI_POSITION

The UI_POSITION structure is used to store positional information (e.g., mouse screen positions) passed through the library in the UI_EVENT structure.

The UI_POSITION structure is declared in **UI_DSP.HPP**. Its fields are shown below:

```
struct EXPORT UI_POSITION
{
    // Members described in UI_POSITION reference chapter.
    int column, line;

#if defined (ZIL_MSWINDOWS)
    void Assign(const POINT &point);
#elif defined(ZIL_OS2)
    void Assign(const POINTL &point);
#endif

    int operator==(const UI_POSITION &position);
    int operator!=(const UI_POSITION &position);
    int operator<(const UI_POSITION &position);
    int operator>(const UI_POSITION &position);
    int operator>=(const UI_POSITION &position);
    int operator<=(const UI_POSITION &position);
    UI_POSITION &operator++(void);
    UI_POSITION &operator--(void);
    UI_POSITION &operator+=(int offset);
    UI_POSITION &operator-=(int offset);
};
```

- *column* is a horizontal position indicator.
- *line* is a vertical position indicator.

Example

```
#include <ui_evt.hpp>

EVENT_TYPE UID_CURSOR::Event(const UI_EVENT &event)
{
    .
    .
    .

    switch (event.rawCode)
    {
    .
    .
    .

    case D_SHOW:
        if (state != D_OFF)
        {
            UI_REGION region;
            region.left = position.column;
            region.top = position.line;
            region.right = region.left + display->cellWidth - 1;
```

```
            region.bottom = region.top + display->cellHeight - 1;
            if (region.Overlap(event.region))
                tState = (event.rawCode == D_HIDE) ? D_HIDE : D_ON;
        }
        break;
    .
    .
    .
    }
}
```

UI_POSITION::Assign

Syntax

#include <ui_dsp.hpp>

void Assign(const POINT &*point*);
or
void Assign(const POINTL &*point*);

Portability

This function is available on the following environments:

☐ DOS ■ MS Windows ■ OS/2 ☐ Motif

Remarks

This function copies the position maintained by a Windows POINT (or OS/2 POINTL) structure into the UI_POSITION object.

- $point_{in}$ is the POINT (or POINTL) structure whose value is to be copied into the UI_POSITION object.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    position.Assign(tPosition);
```

```
        .
        .
        .
    }
```

UI_POSITION::operator ==

Syntax

#include <ui_dsp.hpp>

int operator == (const UI_POSITION &*position*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator compares the UI_POSITION object with another UI_POSITION object specified by *position*.

- $returnValue_{out}$ is TRUE if the two UI_POSITION structures are equivalent. Otherwise, *returnValue* is FALSE.
- $position_{in}$ is the UI_POSITION structure to be compared.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    if (position == tPosition)
    {
        .
        .
        .

    }
}
```

UI_POSITION::operator !=

Syntax

#include <ui_dsp.hpp>

int operator != (const UI_POSITION *&position*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator compares the UI_POSITION object with another UI_POSITION object specified by *position*.

- $returnValue_{out}$ is TRUE if the two UI_POSITION structures are not equivalent. Otherwise, *returnValue* is FALSE.
- $position_{in}$ is the UI_POSITION structure to be compared.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    if (position != tPosition)
    {
        .
        .
        .

    }
}
```

UI_POSITION::operator <

Syntax

#include <ui_dsp.hpp>

int operator < (const UI_POSITION *&position*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator compares the UI_POSITION object with another UI_POSITION object specified by *position*.

- *returnValue*$_{out}$ is TRUE if the UI_POSITION object is less than *position*. Otherwise, *returnValue* is FALSE.

- *position*$_{in}$ is the UI_POSITION structure to be compared.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    if (position < tPosition)
    {
        .
        .
        .

    }
}
```

UI_POSITION::operator >

Syntax

#include <ui_dsp.hpp>

int operator > (const UI_POSITION &*position*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator compares the UI_POSITION object with another UI_POSITION object specified by *position*.

- $returnValue_{out}$ is TRUE if the UI_POSITION object is greater than *position*. Otherwise, *returnValue* is FALSE.
- $position_{in}$ is the UI_POSITION structure to be compared.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    if (position > tPosition)
    {
        .
        .
        .

    }
}
```

UI_POSITION::operator >=

Syntax

#include <ui_dsp.hpp>

int operator >= (const UI_POSITION &*position*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator compares the UI_POSITION object with another UI_POSITION object specified by *position*.

- *returnValue*$_{out}$ is TRUE if the UI_POSITION object is greater than or equal to *position*. Otherwise, *returnValue* is FALSE.

- *position*$_{in}$ is the UI_POSITION structure to be compared.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    if (position >= tPosition)
    {
        .
        .
        .

    }
}
```

UI_POSITION::operator <=

Syntax

#include <ui_dsp.hpp>

int operator <= (const UI_POSITION &*position*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator compares the UI_POSITION object with another UI_POSITION object specified by *position*.

- $returnValue_{out}$ is TRUE if the UI_POSITION object is less than or equal to *position*. Otherwise, *returnValue* is FALSE.
- $position_{in}$ is the UI_POSITION structure to be compared.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    if (position <= tPosition)
    {
        .
        .
        .

    }
}
```

UI_POSITION::operator ++

Syntax

#include <ui_dsp.hpp>

UI_POSITION &operator ++ (void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator increments the *line* and *column* values of the UI_POSITION structure.

- *returnValue*$_{out}$ is the address of the UI_POSITION object. This value is only used by this operation.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    .
    .
    .
    position++;
}
```

UI_POSITION::operator --

Syntax

#include <ui_dsp.hpp>

UI_POSITION &operator -- (void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator decrements the *line* and *column* values of the UI_POSITION structure.

- *returnValue*$_{out}$ is the address of the UI_POSITION object. This value is only used by this operation.

Example

```
#include <ui_dsp.hpp>
ExampleFunction ( )
{
    .
    .
    .
    position--;
}
```

UI_POSITION::operator +=

Syntax

#include <ui_dsp.hpp>

UI_POSITION &operator += (int offset);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator increments the *line* and *column* values of the UI_POSITION

structure by *offset*.

- *returnValue*$_{out}$ is the address of the UI_POSITION object. This value is only used by this operation.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    .
    .
    .
    position += 5;
}
```

UI_POSITION::operator -=

Syntax

#include <ui_dsp.hpp>

UI_POSITION &operator -= (int offset);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator decrements the *line* and *column* values of the UI_POSITION structure by *offset*.

- *returnValue*$_{out}$ is the address of the UI_POSITION object. This value is only used by this operation.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    .
    .
    .

    position -= 5;
}
```

CHAPTER 29 – UI_QUEUE_BLOCK

The UI_QUEUE_BLOCK is an advanced class that is only used by the Event Manager. In general, programmers should not be concerned with it. It allows the program to allocate a large piece of memory at the start of the application and use it repeatedly instead of continually creating and destroying memory space throughout the application. The UI_QUEUE_BLOCK class is designed to behave like a list and thus has access to all of the list functions within Zinc Application Framework. It is always used with the UI_QUEUE_ELEMENT class.

The UI_QUEUE_BLOCK class is declared in **UI_EVT.HPP**. Its public and protected members are:

```
class EXPORT UI_QUEUE_BLOCK : public UI_LIST_BLOCK
{
public:
    // Members described in UI_QUEUE_BLOCK reference chapter.
    UI_QUEUE_BLOCK(int noOfElements);
    ~UI_QUEUE_BLOCK(void);

    // Members described in UI_LIST reference chapter.
    UI_QUEUE_ELEMENT *Current(void);
    UI_QUEUE_ELEMENT *First(void);
    UI_QUEUE_ELEMENT *Last(void);
};
```

UI_QUEUE_BLOCK::UI_QUEUE_BLOCK

Syntax

#include <ui_evt.hpp>

UI_QUEUE_BLOCK(int *noOfElements*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor allocates an array of UI_QUEUE_ELEMENT classes and initializes them

to behave like a list. This is done so that the programmer can have access to all of the list functions without having to create new array functions.

- *noOfElements*$_{in}$ designates the number of elements to be assigned space in memory.

Example

```
#include <ui_evt.hpp>

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

UI_EVENT_MANAGER::UI_EVENT_MANAGER(UI_DISPLAY *_display, int _noOfElements) :
    UI_LIST(UI_DEVICE::CompareDevices), queueBlock(_noOfElements), level(1)
{
    display = _display;
    UI_DEVICE::display = display;
    UI_DEVICE::eventManager = this;
}
```

UI_QUEUE_BLOCK::~UI_QUEUE_BLOCK

Syntax

#include <ui_evt.hpp>

~UI_QUEUE_BLOCK(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This destructor destroys the class information associated with the UI_QUEUE_BLOCK class.

Example

```
#include <ui_evt.hpp>

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

CHAPTER 30 – UI_QUEUE_ELEMENT

The UI_QUEUE_ELEMENT class is an advanced class that is only used by the UI_-QUEUE_BLOCK class within the Event Manager. It contains the UI_EVENT structure, which includes the information that devices use to operate within the system. In general, programmers should not be concerned with it.

The UI_QUEUE_ELEMENT class is declared in **UI_EVT.HPP**. Its public and protected members are:

```
class EXPORT UI_QUEUE_ELEMENT : public UI_ELEMENT
{
public:
    // Members described in UI_QUEUE_ELEMENT reference chapter.
    UI_QUEUE_ELEMENT(void);
    UI_EVENT event;

    // Members described in UI_ELEMENT reference chapter.
    UI_QUEUE_ELEMENT *Next(void);
    UI_QUEUE_ELEMENT *Previous(void);
};
```

- *event* is the UI_EVENT information associated with the queue element.

UI_QUEUE_ELEMENT::UI_QUEUE_ELEMENT

Syntax

#include <ui_evt.hpp>

UI_QUEUE_ELEMENT(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor creates a UI_QUEUE_ELEMENT object.

Example

```
#include <ui_evt.hpp>

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

CHAPTER 31 – UI_REGION

The UI_REGION structure is used to store information about the rectangular regions (e.g., window object regions) used with all window objects.

The UI_REGION structure is declared in **UI_DSP.HPP**. Its public members are:

```
struct EXPORT UI_REGION
{
public:
    // Members described in UI_REGION reference chapter.
    int left, top, right, bottom;

#if defined (ZIL_MSWINDOWS)
    void Assign(const RECT &rect);
#elif defined (ZIL_OS2)
    void Assign(const RECTL &rect);
#endif

    int Encompassed(const UI_REGION &region);
    int Height(void);
    int Overlap(const UI_REGION &region);
    int Overlap(const UI_POSITION &position);
    int Overlap(const UI_REGION &region, UI_REGION &result);
    int Touching(const UI_POSITION &position);
    int Width(void);
    int operator==(const UI_REGION &region);
    int operator!=(const UI_REGION &region);
    UI_REGION &operator++(void);
    UI_REGION &operator--(void);
    UI_REGION &operator+=(int offset);
    UI_REGION &operator-=(int offset);
};
```

- *left* and *top* is the starting position of the region.
- *right* and *bottom* is the ending position of the region.

UI_REGION::Assign

Syntax

#include <ui_dsp.hpp>

void Assign(const RECT &*rect*);
or
void Assign(const RECTL &*rect*);

Portability

This function is available on the following environments:

☐ DOS ■ MS Windows ■ OS/2 ☐ Motif

Remarks

This function copies the region maintained by a Windows RECT (or OS/2 RECTL) structure into the UI_REGION object.

- $rect_{in}$ is the RECT (or RECTL) structure whose value is to be copied into the UI_REGION object.

Example

```
#include <ui_dsp.hpp>

ExampleFunction()
{
    region.Assign(tRegion);
    .
    .
    .

}
```

UI_REGION::Encompassed

Syntax

#include <ui_dsp.hpp>

int Encompassed(const UI_REGION &*region*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function tests if the UI_REGION object is completely encompassed by the UI_REGION structure specified by *region*.

- *returnValue*$_{out}$ is TRUE if the UI_REGION object is encompassed by *region*. Otherwise, *returnValue* is FALSE.

- *region*$_{in}$ is the UI_REGION structure to be compared.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    if (region1.Encompassed(region2))
    {
        .
        .
        .

    }
}
```

UI_REGION::Height

Syntax

#include <ui_dsp.hpp>

int Height(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns the height of the region.

- *returnValue*$_{out}$ is an integer containing the height of the region.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    if (region1.Encompassed(region2))
    {
        .
        .
        .

    }
}
```

UI_REGION::Overlap

Syntax

#include <ui_dsp.hpp>

int Overlap(const UI_REGION *&region*);
or
int Overlap(const UI_POSITION *&position*);
or
int Overlap(const UI_REGION *&region*, UI_REGION *&result*);

Portability

These functions are available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded functions test if the UI_REGION object overlaps another region.

The first overloaded function tests to see if the UI_REGION object is overlapped by another UI_REGION structure specified by *region*.

- $returnValue_{out}$ is TRUE if the UI_REGION object is overlapped by *region*. Otherwise, *returnValue* is FALSE.

- $region_{in}$ is the UI_REGION structure to be compared.

The second overloaded function tests to see if the UI_REGION object is overlapped by a UI_POSITION structure specified by *position*.

- $returnValue_{out}$ is TRUE if the UI_REGION object is overlapped by *position*. Otherwise, *returnValue* is FALSE.

- $position_{in}$ is the UI_POSITION structure to be compared.

The third overloaded function tests to see if the UI_REGION object is overlapped by the UI_REGION structure specified by *region*. The result of the overlapping regions is copied into *result*.

- $returnValue_{out}$ is TRUE if the UI_REGION object is overlapped by *region*. Otherwise, *returnValue* is FALSE.

- $region_{in}$ is the UI_REGION structure to be compared.

- $result_{in/out}$ is the region overlapped by the UI_REGION object and *region*.

Example 1

```
#include <ui_dsp.hpp>

EVENT_TYPE UID_CURSOR::Event(const UI_EVENT &event)
{
    .
    .
    .

    switch (event.rawCode)
    {
    .
    .
    .
    .

    case D_SHOW:
        if (state != D_OFF)
        {
            UI_REGION region;
            region.left = position.column;
            region.top = position.line;
            region.right = region.left + display->cellWidth - 1;
            region.bottom = region.top + display->cellHeight - 1;
            if (region.Overlap(event.region))
                tState = (event.rawCode == D_HIDE) ? D_HIDE : D_ON;
        }
        break;

    }
}
```

- *returnValue*$_{out}$ is TRUE if the two UI_REGION structures are equivalent. Otherwise, *returnValue* is FALSE.

- *region*$_{in}$ is the UI_REGION structure to be compared.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    if (region1 == region2)
    {
        .
        .
        .

    }
}
```

UI_REGION::operator !=

Syntax

#include <ui_dsp.hpp>

int operator != (const UI_REGION &*region*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator compares the UI_REGION object with another UI_REGION object specified by *region*.

- *returnValue*$_{out}$ is TRUE if the two UI_REGION structures are not equivalent. Otherwise, *returnValue* is FALSE.

- *region*$_{in}$ is the UI_REGION structure to be compared.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    if (region1 != region2)
    {
        .
        .
        .

    }
}
```

UI_REGION::operator ++

Syntax

#include <ui_dsp.hpp>

UI_REGION &operator ++ (void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator decrements *left* and *top* and increments the *right* and *bottom* values of the UI_REGION structure by one.

- *returnValue*$_{out}$ is the address of the UI_REGION object. This value is only used by this operation.

Example

```
#include <ui_dsp.hpp>
ExampleFunction ( )
{
    .
    .
    .

    region++;
}
```

UI_REGION::operator --

Syntax

#include <ui_dsp.hpp>

UI_REGION &operator -- (void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator increments *left* and *top* and decrements the *right* and *bottom* values of the UI_REGION structure by one.

- *returnValue*$_{out}$ is the address of the UI_REGION object. This value is only used by this operation.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    .
    .
    .

    region--;
}
```

UI_REGION::operator +=

Syntax

#include <ui_dsp.hpp>

UI_REGION &operator += (int *offset*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator subtracts *offset* from *left* and *top* and adds *offset* to *right* and *bottom*. The values *left*, *top*, *right* and *bottom* are members of the UI_REGION structure.

- $returnValue_{out}$ is the address of the UI_REGION object. This value is only used by this operation.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    .
    .
    .
    region += 5;
}
```

UI_REGION::operator -=

Syntax

#include <ui_dsp.hpp>

UI_REGION &operator -= (int offset);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded operator adds *offset* to *left* and *top* and subtracts *offset* from *right* and *bottom*. The values *left*, *top*, *right* and *bottom* are members of the UI_REGION structure.

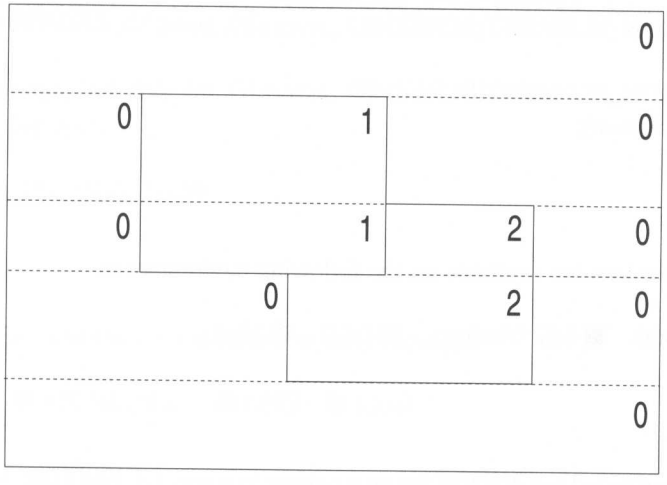
- *returnValue*$_{out}$ is the address of the UI_REGION object. This value is only used by this operation.

Example

```
#include <ui_dsp.hpp>

ExampleFunction ( )
{
    .
    .
    .
    region -= 5;
}
```

CHAPTER 32 – UI_REGION_ELEMENT

The UI_REGION_ELEMENT class is used to store a rectangular region with a specific identification. For example, the screen uses UI_REGION_ELEMENT class objects to store screen regions with the appropriate window identification. The picture below shows how a screen may be split up (where 0 is the screen background and 1 and 2 are overlapping windows):



The UI_REGION_ELEMENT class is declared in **UI_DSP.HPP**. Its public and protected members are:

```
class EXPORT UI_REGION_ELEMENT : public UI_ELEMENT
{
public:
    // Members described in UI_REGION_ELEMENT reference chapter.
    SCREENID screenID;
    UI_REGION region;

    UI_REGION_ELEMENT(SCREENID _screenID, const UI_REGION &_region);
    UI_REGION_ELEMENT(SCREENID _screenID, int _left, int _top, int _right,
        int _bottom);

    // Members described in UI_ELEMENT reference chapter.
    UI_REGION_ELEMENT *Next(void);
    UI_REGION_ELEMENT *Previous(void);
};
```

- *screenID* is the identification to associate with the region. When a screen region is reserved using **display->RegionDefine( )**, *screenID* represents an identification associated with the window object that is to be displayed on the screen.

- *region* is the rectangular region that is to be reserved.

UI_REGION_ELEMENT::UI_REGION_ELEMENT

Syntax

#include <ui_dsp.hpp>

UI_REGION_ELEMENT(SCREENID *_screenID*, const UI_REGION &*_region*);
or
UI_REGION_ELEMENT(SCREENID *_screenID*, int *_left*, int *_top*, int *_right*, int *_bottom*);

Portability

These functions are available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors return a pointer to a new UI_REGION_ELEMENT object.

- $\_screenID_{in}$ is the identification to associate with the region.
- $\_region_{in}$ is the region to define.
- $\_left_{in}$, $\_top_{in}$, $\_right_{in}$ and $\_bottom_{in}$ are a second way of defining the region.

Example

```
#include <ui_dsp.hpp>

void UI_DISPLAY::RegionDefine(SCREENID screenID, int left, int top, int right,
    int bottom)
{
    UI_REGION region;
    region.left = left;
    region.top = top;
    region.right = right;
    region.bottom = bottom;
    // See if it is a full screen definition.
    if (region.left <= 0 && region.top <= 0 &&
        region.right >= columns - 1 && region.bottom >= lines - 1)
```

```
    {
        UI_REGION_LIST::Destroy();
        Add(0, new UI_REGION_ELEMENT(screenID, 0, 0, columns - 1, lines - 1));
        return;
    }
    .
    .
    .
}
```

UI_REGION_ELEMENT::~UI_REGION_ELEMENT

Syntax

#include <ui_dsp.hpp>

~UI_REGION_ELEMENT(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This destructor destroys the class information associated with the UI_REGION_-ELEMENT object.

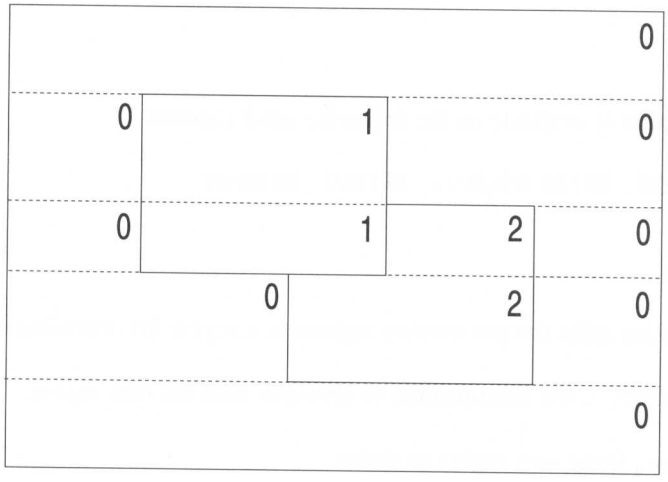
Example

```
#include <ui_dsp.hpp>

void UI_REGION_LIST::Split(int screenID, const UI_REGION &region)
{
    UI_REGION tRegion, sRegion;
    UI_REGION_ELEMENT *dRegion, *t_dRegion;
    // Split any overlapping regions.
    for (dRegion = First(); dRegion; dRegion = t_dRegion)
    {
        .
        .
        .


        // Region 3 is the object's region.
```

```
        UI_LIST::Subtract(dRegion);
        delete dRegion;
    }
}
```

CHAPTER 33 – UI_REGION_LIST

The UI_REGION_LIST class is used to store region elements. For example, it is used when a screen is split up into regions according to the windows that are displayed. The following picture shows how a sample screen may be split up (where 0 is the screen background and 1 and 2 are overlapping windows):



The region list is used to contain each element and to determine how a group of regions should be re-defined if a new region is declared.

The UI_REGION_LIST class is declared in **UI_DSP.HPP**. Its public and protected members are:

```
class EXPORT UI_REGION_LIST : public UI_LIST
{
public:
    void Split(SCREENID screenID, const UI_REGION &region,
        int allocateBelow = FALSE);

    // Members described in UI_LIST reference chapter.
    UI_REGION_ELEMENT *Current(void);
    UI_REGION_ELEMENT *First(void);
    UI_REGION_ELEMENT *Last(void);
};
```

UI_REGION_LIST::Split

Syntax

#include <ui_dsp.hpp>

void Split(SCREENID *screenID*, const UI_REGION &*region*,
int *allocateBelow* = FALSE);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function splits the pre-existing regions in a region list according to the new region.

- $screenID_{in}$ is the identification to associate with the new region.
- $region_{in}$ is the new region to define.
- $allocateBelow_{in}$ specifies whether the new region should be created.

Example

```
#include <ui_dsp.hpp>

void UI_DISPLAY::RegionDefine(SCREENID screenID, int left, int top, int right,
    int bottom)
{
    UI_REGION region;
    region.left = left;
    region.top = top;
    region.right = right;
    region.bottom = bottom;
    .
    .
    .

    // Clip regions partially off the screen to fit on the screen.
    if (region.left < 0)
        region.left = 0;
    if (region.right >= columns)
        region.right = columns - 1;
    if (region.top < 0)
        region.top = 0;
```

```
    if (region.bottom >= lines)
        region.bottom = lines - 1;
    // Split any overlapping regions.
    Split(screenID, region);

    // Define the new display region.
    Add(0, new UI_REGION_ELEMENT(screenID, &region));
}
```

CHAPTER 34 – UI_SCROLL_INFORMATION

The UI_SCROLL_INFORMATION structure is an advanced structure used to relay messages from objects to a scroll bar through the UI_EVENT structure. The object classes that use scroll bars are: UIW_COMBO_BOX (inherent part of the class), UIW_-HZ_LIST, UIW_VT_LIST, UIW_TEXT and UIW_WINDOW.

The UI_SCROLL_INFORMATION structure is declared in **UI_EVT.HPP**. Its fields are shown below:

```
struct EXPORT UI_SCROLL_INFORMATION
{
    // Members described in UI_SCROLL_INFORMATION reference chapter.
    int current, minimum, maximum, showing, delta;
};
```

- *current*, *maximum*, *minimum* and *showing* are values that indicate the current status of the scroll bar. *current* is the current position within the screen. *maximum* is the maximum amount of the scroll region, or the maximum size of the object to which the scroll bar is attached. *minimum* is the minimum amount of the scroll region. Together these variables are used in ratios, based upon the method of measurement of the object to which the scroll bar is attached. For example, if a scroll bar were attached to a text object of 10 lines, where 5 lines were showing and the cursor were on the third line, *current*, *maximum* and *showing* would be 3, 10 and 5 respectively.

- *delta* indicates to the scroll bar how many lines up or down it needs to move the middle. For example, a *delta* value of -2 means that the scrollbar associated with a UIW_TEXT object must be scrolled up two lines.

Portability

This structure is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Example

```
#include <ui_win.hpp>

EVENT_TYPE UIW_WINDOW::Event(const UI_EVENT &event)
{
    UI_WINDOW_OBJECT *object;
    .
    .
    .
```

```
    // Switch on the event type.
    switch (ccode)
    {
    case S_HSCROLL_CHECK:
    case S_VSCROLL_CHECK:
    case S_HSCROLL:
    case S_VSCROLL:
        {
        object = Current();
        int hDelta = 0, vDelta = 0;
        .
        .
        .
        else if (ccode == S_HSCROLL && hScroll)
        {
            hScroll->Event(event);
            hDelta = -event.scroll.delta * (object->true.right -
                object->true.left + 1);
        }
        else if (ccode == S_VSCROLL && vScroll)
        {
            vScroll->Event(event);
            vDelta = -event.scroll.delta * (object->true.bottom -
                object->true.top + 1);
        }
        .
        .
        .
        break;
    }

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

CHAPTER 35 – UI_STORAGE

The UI_STORAGE class is used to write or read Zinc Application Framework data files. It is created as a class so that the file can be treated as an object, which does the reading and writing.

Although the UI_STORAGE is stored in a file, it should be thought of as a file system and not just a file. The UI_STORAGE class is similar, in design, to the Unix file system. This means that within a UI_STORAGE object, there can be many files (i.e., objects derived from UI_STORAGE_OBJECT) and levels of sub-directories. The programmer has the ability to copy, delete and move files across directories. The maximum length of a UI_STORAGE object is about 16 megabytes with the maximum size of an individual object being 4 megabytes. A single UI_STORAGE object may contain a maximum of 16,000 objects (i.e., UI_STORAGE_OBJECT.)

The main use of UI_STORAGE is to allow persistent objects (e.g., objects created using Zinc Designer) to be stored and retrieved. In addition, programmer-defined objects may also be used with a UI_STORAGE.

The UI_STORAGE class is declared in **UI_GEN.HPP**. Its public and protected members are:

```
struct EXPORT UI_STATS_INFO
{
public:
    long size;
    long createTime;
    long modifyTime;
    USHORT useCount;
    USHORT revision;
    USHORT countryID;
    inum_t inum;
};

class EXPORT UI_STORAGE
{
    friend class EXPORT UI_STORAGE_OBJECT;
    friend class EXPORT UI_STORAGE_SUPPORT;
public:
    static int cacheSize;
    static UI_PATH *searchPath;
    int storageError;

    UI_STORAGE(void);
    UI_STORAGE(const char *name, UIS_FLAGS pFlags = UIS_READWRITE);
    ~UI_STORAGE(void);
    static void AppendFullPath(char *fullPath, const char *pathName = NULL,
        const char *fileName = NULL, const char *extension = NULL);
    static void ChangeExtension(char *name, const char *newExtension);
    int ChDir(const char *newName);
    int RenameObject(const char *oldObject, const char *newName);
    int DestroyObject(const char *name);
    char *FindFirstObject(const char *pattern);
```

```
    char *FindNextObject(void);
    char *FindFirstID(OBJECTID nObjectID);
    char *FindNextID(void);
    int Flush(void);
    int MkDir(const char *newName);
    int Save(int revisions = 0);
    int SaveAs(const char *newName, int revisions = 0);
    UI_STATS_INFO *Stats(void);
    static void StripFullPath(const char *fullPath, char *pathName = NULL,
        char *fileName = NULL, char *objectName = NULL,
        char *objectPathName = NULL);
    static int ValidName(const char *name, int createStorage = FALSE);
    int Version(void);
    int RmDir(const char *name);
    void StorageName(char *buff);
    int Link(const char *path1, const char *path2);
};
```

- *UI_STATS_INFO* contains the status of the file access operations.

  size is the size, in bytes, of the object or file.

  createTime contains the time when the object or file was created. *createTime* uses the C language type *time_t*.

  modifyTime contains the time when the object or file was last modified. *modifyTime* uses the C language type *time_t*.

  useCount is the number of times the object is used. MS-DOS files are used only once.

  revision is the revision number of the file or object (i.e., the number of times that a file or object has been modified.)

  countryID denotes the ID of the country for which the object was created. A value of 0 is used to denote the current country.

  inum is the inode number of the object. (No further documentation of this member is provided.)

- *cacheSize* indicates how much memory is to be used as a read and write cache. The default *cacheSize* is 8 Kbytes.

- *searchPath* contains the search path for the UI_STORAGE file when it is opened.

- *storageError* is the result of the last file access. This value is set to one of the values defined by *errno*. For more information see the global variable *errno* in the compiler language reference manual.

UI_STORAGE::UI_STORAGE

Syntax

#include <ui_gen.hpp>

UI_STORAGE(void);
or
UI_STORAGE(const char **name*, UIS_FLAGS *pFlags* = UIS_READWRITE);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors both return a pointer to a new UI_STORAGE class object.

The first overloaded constructor creates a UI_STORAGE with no associated disk file.

The second overloaded constructor creates a UI_STORAGE and binds it to a disk file.

- $name_{in}$ is either the name or the path name of the file to be opened. If the storage file cannot be opened, this function will use UI_PATH functions to search other paths for the file. For example, "~zinc~examples~*.dat" will return the first **.DAT** file found with that path. Notice that the '~' character is used as the directory separator. The '*' character is considered to be a wildcard character.

- $pFlags_{in}$ indicates how a file is to be opened. The following UIS_FLAGS are supported:

  UIS_READ—Opens the file for read access only.

  UIS_READWRITE—Opens the file for read and write access. This flag allows modifications to be made to the file.

  UIS_CREATE—Creates and opens a file for write access. Any previous file will be deleted.

UIS_OPENCREATE—Creates a file for write access. If a previous file exists it will be opened for read access only and will not be overwritten. If writing to the file is desired, the UIS_READWRITE flag should also be set.

UIS_TEMPORARY—This file is created as a temporary file. When UI_STORAGE is destroyed, this file will be deleted.

Example

```
#include <ui_win.hpp>

UI_HELP_SYSTEM::UI_HELP_SYSTEM(char *pathName,
    UI_WINDOW_MANAGER *windowManager, UI_HELP_CONTEXT helpContext) :
    defaultHelpContext(helpContext)
{
    // Open the help storage unit.
    storage = new UI_STORAGE(pathName, UIS_READ);
    if (storage->storageError)
    {
        delete storage;
        storage = NULL;
    }
    .       .
    .
    .

}
```

UI_STORAGE::~UI_STORAGE

Syntax

#include <ui_gen.hpp>

~UI_STORAGE(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This destructor destroys the class information associated with the UI_STORAGE object

and closes any files opened by the UI_STORAGE constructor. If a file was opened with the UIS_TEMPORARY flag, it will be deleted when this destructor is called.

Example

```
#include <ui_win.hpp>

UI_HELP_SYSTEM::~UI_HELP_SYSTEM(void)
{
    if (storage)
        delete storage;
    delete helpWindow;
}
```

UI_STORAGE::AppendFullPath

Syntax

#include <ui_gen.hpp>

static void AppendFullPath(char **fullPath*, const char **pathName* = NULL, const char **fileName* = NULL, const char **extension* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function combines fragments of the path, file name and extension in order to construct a complete path name.

- $fullPath_{in/out}$ is the complete path name that is passed back.
- $pathName_{in}$ is the name of the path.
- $fileName_{in}$ is the name of the storage file.
- $extension_{in}$ is the extension to the file name (e.g., ".dat").

Example

```
#include <ui_win.hpp>

UI_WINDOW_OBJECT *UI_WINDOW_OBJECT::New(const char *name, UI_STORAGE *directory,
    UI_STORAGE_OBJECT *file)
{
    // Check for a valid directory and file.
    int tempDirectory = FALSE;
    if (name && !file)
    {
        char pathName[128], fileName[32], objectName[32];
        UI_STORAGE::StripFullPath(name, pathName, fileName, objectName);
        if (!directory)
        {
            UI_STORAGE::AppendFullPath(pathName, pathName, fileName);
            UI_STORAGE::ChangeExtension(pathName, ".dat");
            directory = new UI_STORAGE(pathName, UIS_READ);
            tempDirectory = TRUE;
        }
        if (!file)
            return (new UIW_WINDOW(name, directory, NULL));
    }
    .
    .
    .
}
```

UI_STORAGE::ChangeExtension

Syntax

#include <ui_gen.hpp>

static void ChangeExtension(char **name*, const char **newExtension*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function changes the extension (e.g., ".dat") associated with the filename.

- $name_{in/out}$ is the full name of the path.
- $newExtension_{in}$ is the new extension that will replace the previous extension (if any).

Example

```
#include <ui_win.hpp>

UI_WINDOW_OBJECT *UI_WINDOW_OBJECT::New(const char *name, UI_STORAGE *directory,
    UI_STORAGE_OBJECT *file)
{
    // Check for a valid directory and file.
    int tempDirectory = FALSE;
    if (name && !file)
    {
        char pathName[128], fileName[32], objectName[32];
        UI_STORAGE::StripFullPath(name, pathName, fileName, objectName);
        if (!directory)
        {
            UI_STORAGE::AppendFullPath(pathName, pathName, fileName);
            UI_STORAGE::ChangeExtension(pathName, ".DAT");
            directory = new UI_STORAGE(pathName, UIS_READ);
            tempDirectory = TRUE;
        }
        if (!file)
            return (new UIW_WINDOW(name, directory, NULL));
    }
    .
    .
    .
}
```

UI_STORAGE::ChDir

Syntax

#include <ui_gen.hpp>

int ChDir(const char *_newName_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function changes the current working directory within a UI_STORAGE file according to the following parameters:

- *returnValue*$_{out}$ is 0 on success and -1 on failure.

- *newName*$_{in}$ is the name of the directory that will become the new current working directory. *newName* specifies a single sub-directory and not an entire path. The character "." is used to refer to the current working directory and ".." is used to refer to the parent directory. ".." refers to the root directory if the current working directory is the root directory. The separator '~' is similar to the '\' in the DOS directory system.

Example

```
#include <ui_win.hpp>

main(int argc, char *argv[])
{
    .
    .
    .
    // Create the help directory.
    UI_STORAGE *helpFile = new UI_STORAGE(fileName, UIS_READWRITE);
    if (helpFile->storageError)
    {
        delete helpFile;
        helpFile = new UI_STORAGE(fileName, UIS_CREATE | UIS_READWRITE);
    }
    helpFile->MkDir("UI_HPP");
    if (newHelpDirectory)
        helpFile->RmDir("UI_HELP_CONTEXT");
    helpFile->MkDir("UI_HELP_CONTEXT");

    // Print genhelp status.
    PrintStatus("PROCESSING %s:\n", fileName);
    OBJECTID helpID = 0;

    // Generate the HPP directory.
    helpFile->ChDir("~UI_HPP");
    UI_STORAGE_OBJECT *hppElement = new UI_STORAGE_OBJECT(*helpFile,
        "HELP_CONTEXTS", ID_HELP_CONTEXT, UIS_CREATE | UIS_READWRITE);

    // Generate the help contexts.
    helpFile->ChDir("~UI_HELP_CONTEXT");
    .
    .
    .
}
```

UI_STORAGE::DestroyObject

Syntax

#include <ui_gen.hpp>

int DestroyObject(const char *_name_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This removes an object from the UI_STORAGE file. After it is removed, the object is destroyed.

- *returnValue*$_{out}$ is 0 on success. If a failure occurred, -1 is returned.
- *name*$_{in}$ is the name of the object to be destroyed.

Example

```
#include <ui_win.hpp>

int RemoveDirectory(UI_STORAGE *storage, const char *name,
    const char *directoryName)
{
    // Clean the directory.
    for (name = FindFirstObject(name); name; name = FindFirstObject(name))
        DestroyObject(name);

    storage->ChDir("..");
    storage->RmDir(directoryName);
}
```

UI_STORAGE::FindFirstID

Syntax

#include <ui_gen.hpp>

char *FindFirstID(OBJECTID *nObjectID*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function finds the first object in the current directory of the storage object whose *objectID* matches *nObjectID*.

- *returnValue*$_{out}$ is a pointer to a string containing the *stringID* of the object whose *objectID* matches *nObjectID*. If no match is found, *returnValue* is NULL.

- *nObjectID*$_{in}$ is used as the search specification to find an object.

UI_STORAGE::FindFirstObject

Syntax

```
#include <ui_gen.hpp>

char *FindFirstObject(const char *pattern);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function finds the first object whose *stringID* matches *pattern* inside the storage object in its current directory.

- *returnValue*$_{out}$ is a pointer to a string containing the *stringID* of the object that matches *pattern*. If no match is found, *returnValue* is NULL.

- *pattern*$_{in}$ is used as the search specification to find an object. *pattern* may contain wildcards. For example, if *pattern* were "i*", then **FindFirstObject( )** would return the *stringID* of the first object that has an "i" as the first letter of *stringID*. Additionally, the '?' wildcard may be used to specify a single unknown character such as "*.BK?".

UI_STORAGE::FindNextID

Syntax

```
#include <ui_gen.hpp>

char *FindNextID(void);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function finds the next object in the current directory in the storage whose *objectID* matches the *objectID* that was used for the most recent call to **FindFirstID( )**. This function call must be preceded by a call to **FindFirstID( )**.

- *returnValue*$_{out}$ is a pointer to a string containing the *stringID* of the object whose *objectID* matches the *objectID* that was used for the most recent call to **FindFirstID( )**. If no match is found, *returnValue* is NULL.

UI_STORAGE::FindNextObject

Syntax

```
#include <ui_gen.hpp>

char *FindNextObject(void);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function finds the next object in current directory in the storage whose *stringID* matches the *pattern* of the last call to **FindFirstObject( )**. This function call must be preceded by a call to **FindFirstObject( )**.

- *returnValue*$_{out}$ is a pointer to a string containing the *stringID* of the object that matches the *pattern* of the last call to **FindFirstObject( )**. If no match is found, *returnValue* is NULL.

UI_STORAGE::Flush

Syntax

```
#include <ui_gen.hpp>

int Flush(void);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function writes the internal cache buffer to a temporary file. One use of this function would be to save backup copies of a file within a timer function. **Flush( )** does not update the actual storage file. To commit the changes to the storage file, use **Save( )**.

- *returnValue*$_{out}$ is 0 on success and -1 if an error occurs.

Example

```
#include <ui_gen.hpp>

int TimerBackup(UI_STORAGE *storage)
{
    .
    .
    .
```

```
    storage->Flush();
}
```

UI_STORAGE::Link

Syntax

#include <ui_gen.hpp>

int Link(const char *_path1_, const char *_path2_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function gives an existing object a second name. This allows a second (or greater) instance of an object within a **.DAT** file to reference, or point to, the data stored with the original instance of the object. Thus, many objects can use the same data without duplicating the data for each instance. After this function is called, both names refer to the same object. Deleting one name will not delete the other name or the object. This is different than a rename.

- $returnValue_{out}$ is 0 on success and -1 if an error occurs.
- $path1_{in}$ is the original name (including its path) of the object .
- $path2_{in}$ is the new name (including its path) by which the object can also be referenced.

Example

```
#include <ui_gen.hpp>

int EmployeeSetup(void)
{
    .
    .
    .
    storage->Link("~employees~Blake", "~development~Blake");
}
```

UI_STORAGE::MkDir

Syntax

#include <ui_gen.hpp>

int MkDir(const char **newName*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function makes a new directory within a UI_STORAGE file according to the following parameters:

- *returnValue*$_{out}$ is 0 on success and -1 on failure.
- *newName*$_{in}$ is the name of the directory to be created. For example, "~UI_HPP~MYDIR" can be specified to make the directory "MYDIR" in the UI_HPP subdirectory.

Example

```
#include <ui_gen.hpp>

main(int argc, char *argv[])
{
    .
    .
```

```
    .

    // Create the help directory.
    UI_STORAGE *helpFile = new UI_STORAGE(fileName, UIS_READWRITE);
    if (helpFile->storageError)
    {
        delete helpFile;
        helpFile = new UI_STORAGE(fileName, UIS_CREATE | UIS_READWRITE);
    }
    helpFile->MkDir("UI_HPP");
    if (newHelpDirectory)
        helpFile->RmDir("UI_HELP_CONTEXT");
    helpFile->MkDir("UI_HELP_CONTEXT");

    // Print genhelp status.
    PrintStatus("PROCESSING %s:\n", fileName);
    OBJECTID helpID = 0;

    // Generate the HPP directory.
    helpFile->ChDir("~UI_HPP");
    UI_STORAGE_OBJECT *hppElement = new UI_STORAGE_OBJECT(*helpFile,
        "HELP_CONTEXTS", ID_HELP_CONTEXT, UIS_CREATE | UIS_READWRITE);

    // Generate the help contexts.
    helpFile->ChDir("~UI_HELP_CONTEXT");
    .
    .
    .

}
```

UI_STORAGE::RenameObject

Syntax

#include <ui_gen.hpp>

int RenameObject(const char *_oldObject_, const char *_newName_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function changes the name (i.e., *stringID*) of an object inside the storage object.

- *returnValue*$_{out}$ is 0 on success and -1 on failure.

- *oldObject*$_{in}$ is the name of the object whose name is to be changed.
- *newName*$_{in}$ is the new name of the object.

Example

```
#include <ui_gen.hpp>

ExampleFunction(UI_STORAGE *storage)
{
    .
    .
    .
    storage->RenameObject("Item1", "FirstItem");
}
```

UI_STORAGE::RmDir

Syntax

#include <ui_gen.hpp>

int RmDir(const char *_name_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function removes a directory within a UI_STORAGE file according to the following parameters:

- *returnValue*$_{out}$ is 0 on success and -1 on failure.
- *name*$_{in}$ is the name of the directory to be removed.

NOTE: A directory must be empty in order for it to be deleted.

Example

```
#include <ui_win.hpp>

main(int argc, char *argv[])
{
    .
    .
    .

    // Create the help directory.
    UI_STORAGE *helpFile = new UI_STORAGE(fileName, UIS_READWRITE);
    if (helpFile->storageError)
    {
        delete helpFile;
        helpFile = new UI_STORAGE(fileName, UIS_CREATE | UIS_READWRITE);
    }
    helpFile->MkDir("UI_HPP");
    if (newHelpDirectory)
        helpFile->RmDir("UI_HELP_CONTEXT");
    helpFile->MkDir("UI_HELP_CONTEXT");

    // Print genhelp status.
    PrintStatus("PROCESSING %s:\n", fileName);
    OBJECTID helpID = 0;

    // Generate the HPP directory.
    helpFile->ChDir("~UI_HPP");
    UI_STORAGE_OBJECT *hppElement = new UI_STORAGE_OBJECT(*helpFile,
        "HELP_CONTEXTS", ID_HELP_CONTEXT, UIS_CREATE | UIS_READWRITE);

    // Generate the help contexts.
    helpFile->ChDir("~UI_HELP_CONTEXT");
    .
    .
    .

}
```

UI_STORAGE::Save

Syntax

#include <ui_gen.hpp>

int Save(int *revisions* = 0);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function saves the storage file and all of the information contained in it. This operation (or a call to **SaveAs( )**) must be performed even if each individual object is stored. Since some objects are saved in temporary files, this is the only way to ensure that the information is saved to the main storage file.

- *returnValue*$_{out}$ is 0 on success, or -1 on failure.

- *revisions*$_{in}$ is the number of backup files to be kept. Backup files are files with the .BK? extension where the "?" denotes which backup number the file is. For example, **TEST.BK1** would be the most recent backup of the file **TEST.DAT**, **TEST.BK2** would be the previous backup of the file **TEST.DAT**, etc.

NOTE: A backup file is only created the first time **Save( )** is called. The different backup file revisions are created from previous times the file was opened. To create another backup file, you must close and then re-open the storage.

Example

```
#include <ui_gen.hpp>

int UI_STORAGE::SaveAs(const char *newName, int revisions)
{
    if (!FlagSet(flags, UIS_READWRITE))
    {
        storageError = EACCES;
        return -1;
    }
    if (modified) (void) time(&sb->modifytime);
    Flush();
    StripFullPath(newName, pname, fname);
    firstTime = 1;
    Save(revisions);
    return 0;
}
```

UI_STORAGE::SaveAs

Syntax

#include <ui_gen.hpp>

int SaveAs(char **newName*, int *revisions* = 0);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function saves the storage file and all of the information contained in it. This operation (or a call to **Save( )**) must be performed even if each individual object is stored. Since some objects are saved in temporary files, this is the only way to ensure that the information is saved to the main storage file.

- *returnValue*$_{out}$ is 0 on success, or -1 on failure.
- *newName*$_{in}$ is the new name of the storage file.
- *revisions*$_{in}$ is the number of backup files to be kept. Backup files are files with the .BK? extension where the "?" denotes which backup number the file is. For example, **TEST.BK1** would be the most recent backup of the file **TEST.DAT**, **TEST.BK2** would be the previous backup of the file **TEST.DAT**, etc.

NOTE: A backup file is only created the first time **SaveAs( )** is called. The different backup file revisions are created from previous times the file was opened. To create another backup file, you must close and then re-open the storage.

Example

```
#include <ui_gen.hpp>

int CloseAll(UI_STORAGE *storage, const char *newName, int revisions)
{
    char currentName[129];

    Flush();
    StorageName(currentName);
    if (!strcmp(currentName, newName));
        Save(revisions);
    else
        SaveAs(newName, revisions);
    return 0;
}
```

UI_STORAGE::Stats

Syntax

```
#include <ui_gen.hpp>

UI_STATS_INFO *Stats(void);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns statistics regarding the UI_STORAGE object.

- *returnValue*$_{out}$ is a pointer to a UI_STATS_INFO structure. For more information regarding UI_STATS_INFO, see the beginning of this chapter. If an error occurs, NULL is returned.

UI_STORAGE::StorageName

Syntax

```
#include <ui_gen.hpp>

void StorageName(char *buff);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns the name of the file associated with the UI_STORAGE object.

- *buff*$_{in/out}$ a character pointer used to pass back the name of the UI_STORAGE file. *buff* should be large enough to hold the largest path plus the NULL terminator (i.e., 129 bytes.)

Example

```
#include <ui_gen.hpp>

char *ExampleFunction(UI_STORAGE *storage)
{
    static char name[129];
    storage->StorageName(&name);
    .
    .
    .

    return (&name);
}
```

UI_STORAGE::StripFullPath

Syntax

#include <ui_gen.hpp>

static void StripFullPath(const char **fullPath*, char **pathName* = NULL,
    char **fileName* = NULL, char **objectName* = NULL,
    char *objectPathName = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function takes a full path name and divides it into its path and file name components. The arguments *fullPath*, *pathName*, *fileName*, *objectName* and *objectPathName* all have the NULL argument specified so that the information will not be saved if no other argument is provided.

- *fullPath*$_{in}$ is the complete path name that is passed down.

- *pathName*$_{in/out}$ is the name of the path.
- *fileName*$_{in/out}$ is the name of the storage file (including the extension).
- *objectName*$_{in/out}$ is the name of the specific object.
- *objectPathName*$_{in/out}$ is the path name for the specific object.

Example

```
#include <ui_win.hpp>

UI_WINDOW_OBJECT *UI_WINDOW_OBJECT::New(const char *name, UI_STORAGE *directory,
    UI_STORAGE_OBJECT *file)
{
    // Check for a valid directory and file.
    int tempDirectory = FALSE;
    if (name && !file)
    {
        char pathName[128], fileName[32], objectName[32], objectPathName[128];
        UI_STORAGE::StripFullPath(name, pathName, fileName, objectName,
            objectPathName);
        if (!directory)
        {
            UI_STORAGE::AppendFullPath(pathName, pathName, fileName);
            UI_STORAGE::ChangeExtension(pathName, ".dat");
            directory = new UI_STORAGE(pathName, UIS_READ);
            tempDirectory = TRUE;
        }
        if (!file)
            return (new UIW_WINDOW(name, directory, NULL));
    }
    .
    .
    .
}
```

UI_STORAGE::ValidName

Syntax

#include <ui_gen.hpp>

static int ValidName(const char **name*, int *createStorage* = FALSE);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function indicates whether the specified MS-DOS file exists on the disk.

- *returnValue*$_{out}$ is TRUE if the file exists or if it can be created. Otherwise, *returnValue* is FALSE.
- *name*$_{in}$ is the name of the file to be checked.
- *createStorage*$_{in}$, when TRUE, allows the name to not actually exist as long as the directory and the file name are valid. When *createStorage* is FALSE, the file must exist in the specified path and directory.

Example

```
#include <ui_gen.hpp>

main(int argc, char *argv[])
{
    // Make sure there is a specified text file.
    if (argc != 2)
    {
        printf("Usage: genhelp <text file name>\n");
        return (1);
    }

    // Open the text file.
    char fileName[128];
    strcpy(fileName, argv[1]);
    UI_STORAGE::ChangeExtension(fileName, ".txt");
    FILE *textFile = fopen(fileName, "rb");
    if (!textFile)
        printf("Could not open the text file: %s.\n", fileName);
    // Open the data file.
    UI_STORAGE::ChangeExtension(fileName, ".dat");


    if (!UI_STORAGE::ValidName(fileName, TRUE))
    {
        printf("Could not create the help data file: %s.\n", fileName);
        return (0);
    }
    .
    .
    .

}
```

UI_STORAGE::Version

Syntax

#include <ui_gen.hpp>

int Version(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns the version of UI_STORAGE used to create the file.

- *returnValue*$_{out}$ is an integer containing the UI_STORAGE version number. The version number is "301" for version 3.5 of Zinc Application Framework.

Example

```
#include <ui_gen.hpp>

ExampleFunction(UI_STORAGE *storage)
{
    float version = (storage->Version())/100;
    printf("The version of this data file is %f.\n", version);
}
```

CHAPTER 36 – UI_STORAGE_OBJECT

The UI_STORAGE_OBJECT class is used to access (i.e., load or store) an object's data using Zinc Application Framework data files. Although the UI_STORAGE_OBJECT is stored in a file, it should be thought of as a file and not a record in a file. (See "Chapter 35—UI_STORAGE" in this manual for more information regarding data files.)

The main use of UI_STORAGE_OBJECT is to provide a method to load or store persistent objects (e.g., objects created using Zinc Designer) using Zinc data files. It is used in conjunction with the UI_STORAGE class.

The UI_STORAGE_OBJECT class is declared in **UI_GEN.HPP**. Its public members are:

```
class EXPORT UI_STORAGE_OBJECT
{
    friend class EXPORT UI_STORAGE;
    friend class EXPORT UI_STORAGE_SUPPORT;
public:
    int objectError;
    OBJECTID objectID;
    char stringID[32];

    UI_STORAGE_OBJECT(void);
    UI_STORAGE_OBJECT(UI_STORAGE &file, const char *name,
        OBJECTID nObjectID, UIS_FLAGS pFlags = UIS_READWRITE);
    ~UI_STORAGE_OBJECT(void);
    int Load(void *buff, int size, int length);
    int Load(char *string, int length);
    int Load(char **string);
    int Load(char *value);
    int Load(UCHAR *value);
    int Load(short *value);
    int Load(USHORT *value);
    int Load(long *value);
    int Load(ULONG *value);
    void Touch(void);
    UI_STATS_INFO *Stats(void);
    UI_STORAGE *Storage(void);
    int Store(void *buff, int size, int length);
    int Store(const char *string);
    int Store(char value);
    int Store(UCHAR value);
    int Store(short value);
    int Store(USHORT value);
    int Store(long value);
    int Store(ULONG value);
};
```

- *objectError* is the result of the last attempt to load (or store) this object from a file. This value is set to one of the values defined by *errno*. For more information see the global variable *errno* in the compiler language reference manual.

- *objectID* is an identification code for the type of object being loaded or stored.

- *stringID* is the name of the object (in string format).

UI_STORAGE_OBJECT::UI_STORAGE_OBJECT

Syntax

```
#include <ui_gen.hpp>

UI_STORAGE_OBJECT(void);
    or
UI_STORAGE_OBJECT(UI_STORAGE &file, const char *name, OBJECTID nObjectID,
    UIS_FLAGS pFlags = UIS_READWRITE);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors each return a pointer to a new UI_STORAGE_OBJECT.

The first overloaded constructor creates a UI_STORAGE_OBJECT with no associated object. This advanced constructor is reserved for internal use only.

The second overloaded constructor creates a UI_STORAGE_OBJECT with the following parameters:

- $file_{in}$ is the file containing the object. If the object is not found, it will be created if UIS_CREATE or UIS_OPENCREATE and UIS_READWRITE are specified.
- $name_{in}$ is the *stringID* of the target object.
- $nObjectID_{in}$ is the *objectID* of the target object.
- $pFlags_{in}$ indicates how the storage object is to be opened. The following UIS_FLAGS are supported:

  UIS_READ—Allows read only access to the object.

UIS_READWRITE—Allows read and write access to the object. This flag allows modifications to be made to the object.

UIS_CREATE—Creates an object and allows write access to it. Any previous object will be deleted.

UIS_OPENCREATE—Creates an object and allows write access to it. If a previous object exists it will be opened and not be overwritten.

Example

```
#include <ui_win.hpp>

void UIW_WINDOW::Load(const char *name, UI_STORAGE *directory,
    UI_STORAGE_OBJECT *file)
{
    // Check for a valid directory and file.
    int tempDirectory = FALSE, tempFile = FALSE;
    if (name && !file)
    {
        char pathName[128], fileName[32], objectName[32];
        UI_STORAGE::StripFullPath(name, pathName, fileName, objectName);
        if (!directory)
        {
            UI_STORAGE::AppendFullPath(pathName, pathName, fileName);
            UI_STORAGE::ChangeExtension(pathName, ".dat");
            directory = new UI_STORAGE(pathName, UIS_READ);
            tempDirectory = TRUE;
        }
        if (!file)
        {
            if (objectName[0] == '\0')
                strcpy(objectName, fileName);
            directory->ChDir("~UIW_WINDOW");
            file = new UI_STORAGE_OBJECT(*directory, objectName, ID_WINDOW,
                UIS_READ);
            if (file->objectError)
            .
            .
            .
        }
    }
    .
    .
    .

}
```

UI_STORAGE_OBJECT::~UI_STORAGE_OBJECT

Syntax

#include <ui_gen.hpp>

~UI_STORAGE_OBJECT(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This destructor destroys the class information associated with the UI_STORAGE_-OBJECT.

Example

```
#include <ui_win.hpp>

void UIW_WINDOW::Load(const char *name, UI_STORAGE *directory,
    UI_STORAGE_OBJECT *file)
{
    .
    .
    .

    // Clean up the file and storage.
    if (tempFile)
        delete file;
    if (tempDirectory)
        delete directory;
}
```

UI_STORAGE_OBJECT::Load

Syntax

#include <ui_gen.hpp>

int Load(void **buff*, int *size*, int *length*);

or

int Load(char *_string_, int _length_);
or
int Load(char **_string_);
or
int Load(char *_value_);
or
int Load(UCHAR *_value_);
or
int Load(short *_value_);
or
int Load(USHORT *_value_);
or
int Load(long *_value_);
or
int Load(ULONG *_value_);

Portability

These functions are available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first overloaded function allows the programmer to have low-level access to the file. In general, programmers are discouraged from using this function, because the integrity of the type of value being loaded cannot be guaranteed across environments. For example, the storage size of a value in DOS might be different than that in Motif. All of the other **Load( )** functions, however, are the same for DOS and Motif.

The first overloaded function reads information from the storage file according to the following values:

- $buff_{in/out}$ is a pointer to the buffer that will receive the information. This buffer must be large enough to contain the information read.
- $size_{in}$ is the size of each item to be read.
- $length_{in}$ is the number of items to be read.

The second overloaded function reads information from the storage file according to the following values:

- *string*$_{in/out}$ is a pointer to the buffer that will receive the information. This buffer must be large enough to contain the information read.

- *length*$_{in}$ is the number of bytes that may be read.

The third overloaded function reads information from the storage file according to the following values:

- *string*$_{in/out}$ is a pointer to a string pointer where the information will be written. This string is allocated by the library.

The rest of the overloaded functions read information from the storage file according to the type of value given.

- *returnValue*$_{out}$ is the number of bytes read.

- *value*$_{in/out}$ is the numeric value to be read. A pointer to this value must be set. The following values are supported:

  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.)

  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.)

UI_STORAGE_OBJECT::Stats

Syntax

#include <ui_gen.hpp>

UI_STATS_INFO *Stats(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns some statistics regarding the UI_STORAGE_OBJECT.

- *returnValue*$_{out}$ is a pointer to a UI_STATS_INFO structure. For more information regarding UI_STATS_INFO, see the beginning of "Chapter 35—UI_STORAGE." If an error occurs, NULL is returned.

UI_STORAGE_OBJECT::Storage

Syntax

#include <ui_gen.hpp>

UI_STORAGE *Storage(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a pointer to the UI_STORAGE that contains the UI_STORAGE_-OBJECT.

- *returnValue*$_{out}$ is a pointer to a UI_STORAGE file.

UI_STORAGE_OBJECT::Store

Syntax

```
#include <ui_gen.hpp>

int Store(void *buff, int size, int length);
    or
int Store(const char *string);
    or
int Store(char value);
    or
int Store(UCHAR value);
    or
int Store(short value);
    or
int Store(USHORT value);
    or
int Store(long value);
    or
int Store(ULONG value);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first overloaded function allows the programmer to have low-level access to the file. In general, programmers are discouraged from using this function, because the integrity of the type of value being loaded cannot be guaranteed across environments. For

example, the storage size of a value (e.g., int) in DOS might be different than that in Motif. All of the other **Store( )** functions, however, are the same for DOS and Motif.

The first overloaded function writes information into the storage file according to the following values:

- $buff_{in/out}$ is a pointer to the buffer that contains the information to be written.
- $size_{in}$ is the size of each item to be written.
- $length_{in}$ is the number of items to be written.

The second overloaded function writes information into the storage file according to the following value:

- $string_{in/out}$ is a pointer to the buffer that contains the information.

The rest of the overloaded functions write information to the storage file according to the type of value given.

- $returnValue_{out}$ is the number of bytes written.
- $value_{in/out}$ is the numeric value to be write. The following values are supported:

  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.)

  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.)

UI_STORAGE_OBJECT::Touch

Syntax

#include <ui_gen.hpp>

void Touch(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function sets *UI_STORAGE::inode.modifyTime* (the time of the last modification) of the storage object to the current time.

Example

```
#include <ui_gen.hpp>

ExampleFunction(UI_STORAGE_OBJECT *object)
{
    .
    .
    .
    object->Touch();
}
```

CHAPTER 37 – UI_TEXT_DISPLAY

The UI_TEXT_DISPLAY class implements a text display that writes to screen memory. Since the UI_TEXT_DISPLAY class is derived from the display class UI_DISPLAY, only details specific to the UI_TEXT_DISPLAY class are given in this chapter. For descriptions and examples regarding virtual or inherited display members, see "Chapter 6—UI_DISPLAY."

The UI_TEXT_DISPLAY class is declared in **UI_DSP.HPP**. Its public and protected members are:

```
class EXPORT UI_TEXT_DISPLAY : public UI_DISPLAY, public UI_REGION_LIST
{
public:
    TDM_MODE mode;

    UI_TEXT_DISPLAY(TDM_MODE _mode = TDM_AUTO);
    virtual ~UI_TEXT_DISPLAY(void);
    virtual void Line(SCREENID screenID, int column1, int line1,
        int column2, int line2, const UI_PALETTE *palette, int width = 1,
        int xor = FALSE, const UI_REGION *clipRegion = NULL);
    virtual COLOR MapColor(const UI_PALETTE *palette, int isForeground);
    virtual void Rectangle(SCREENID screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1,
        int fill = FALSE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL);
    virtual void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion, SCREENID screenID = ID_SCREEN);
    virtual void RegionDefine(SCREENID screenID, int left, int top,
        int right, int bottom);
    virtual void RegionMove(const UI_REGION &oldRegion, int newColumn,
        int newLine, SCREENID oldScreenID = ID_SCREEN,
        SCREENID newScreenID = ID_SCREEN);
    virtual void Text(SCREENID screenID, int left, int top,
        const char *text, const UI_PALETTE *palette, int length = -1,
        int fill = TRUE, int xor = FALSE,
        const UI_REGION *clipRegion = NULL,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextHeight(const char *string,
        SCREENID screenID = ID_SCREEN, LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int TextWidth(const char *string, SCREENID screenID = ID_SCREEN,
        LOGICAL_FONT font = FNT_DIALOG_FONT);
    virtual int VirtualGet(SCREENID screenID, int left, int top, int right,
        int bottom);
    virtual int VirtualPut(SCREENID screenID);

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

protected:
    USHORT *_screen;
    USHORT *_moveBuffer;
    int _virtualCount;
    UI_REGION _virtualRegion;
    char _stopDevice;

    void CloseScreen(void);
```

```
    void GetText(int left, int top, int right, int bottom, void *buffer);
    void MoveText(int left, int top, int right, int bottom, int destX,
        int destY);
    void OpenScreen(int _mode);
    void PutText(int left, int top, int right, int bottom, void *buffer);
};
```

- *_mode* is the text mode that is initialized.
- *_screen* is a pointer to the BIOS screen buffer.
- *_moveBuffer* is a pointer to screen memory. This extra memory facilitates faster screen moves.
- *_virtualCount* is a count of the number of virtual screen operations that have taken place. For example, when the **VirtualGet( )** function is called, *_virtualCount* is decremented. Additionally, when the **VirtualPut( )** function is called, *_virtualCount* is incremented.
- *_virtualRegion* is the region affected by either **VirtualGet( )** or **VirtualPut( )**.
- *_stopDevice* is a variable used to prevent recursive updates of device images on the display. If *_stopDevice* is TRUE, no drawing will be done to the screen. Otherwise, drawing will be made directly to the screen display.

UI_TEXT_DISPLAY::UI_TEXT_DISPLAY

Syntax

#include <ui_dsp.hpp>

UI_TEXT_DISPLAY(TDM_MODE *_mode* = TDM_AUTO);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This constructor returns a pointer to a new UI_TEXT_DISPLAY object. When a new

UI_TEXT_DISPLAY class is constructed, the system clears the screen to the background color and pattern specified by the inherited palette variable *backgroundPalette*. (See "Chapter 24—UI_PALETTE" of this manual.)

- *_mode*$_{in}$ specifies the type of text display to create. The available display modes (defined in **UI_DSP.HPP**) are:

  TDM_AUTO—Constructs a text display using the current text mode.

  TDM_25x40 and **TDM_BW_25x40**—Split the screen display into 25 line by 40 column character cells.

  TDM_25x80, **TDM_BW_25x80** and **TDM_MONO_25x80**—Split the screen display into 25 line by 80 column character cells.

  TDM_43x80—Splits the screen display into 43 line x 80 column character cells on an EGA display or 50 line x 80 column on a VGA display.

The default configuration of the UI_TEXT_DISPLAY class is given below:

mode—The default mode is determined by the *mode* argument discussed above.

background—The background color is determined by the text background value contained in the *backgroundPalette* variable.

cell size—The default cell size (i.e., *cellWidth* and *cellHeight*) is 1 cell wide by 1 cell high. Thus when objects are constructed, they are placed on normal text cell boundaries.

colors—The run-time colors are determined by the colors defined in the UI_-PALETTE_MAP variables, or by those you define within your application. The text class turns off the blink-bit so that high intensity colors can be used.

Example

```
#include <ui_win.hpp>

main()
{
    // Initialize Zinc Application Framework.
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
    if (!display->installed)
    {
        delete display;
        display = new UI_TEXT_DISPLAY;
    }
```

```
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;

    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .

    // Restore the system.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

CHAPTER 38 – UI_TIME

The UI_TIME class is a lower-level class used to store hour, minute, second and hundredths of second time information. It is not a window object. (See "Chapter 67—UIW_TIME" of this manual for information about the time window object.)

The UI_TIME class is declared on **UI_GEN.HPP**. Its public and protected members are shown below:

```
class EXPORT UI_TIME : public UI_INTERNATIONAL
{
public:
    // Members described in UI_TIME reference chapter.
    static char *amPtr;
    static char *pmPtr;

    UI_TIME(void);
    UI_TIME(const UI_TIME &time);
    UI_TIME(int hour, int minute, int second = 0, int hundredth = 0);
    UI_TIME(const char *string, TMF_FLAGS tmFlags = TMF_NO_FLAGS);
    UI_TIME(int packedTime);
    virtual ~UI_TIME(void);
    void Export(int *hour, int *minute, int *second = NULL,
        int *hundredth = NULL);
    void Export(char *string, TMF_FLAGS tmFlags);
    void Export(int *packedTime);
    TMI_RESULT Import(void);
    TMI_RESULT Import(const UI_TIME &time);
    TMI_RESULT Import(int hour, int minute, int second = 0,
        int hundredth = 0);
    TMI_RESULT Import(const char *string, TMF_FLAGS tmFlags);
    TMI_RESULT Import(int packedTime);
    long operator=(long hundredths);
    long operator=(const UI_TIME &time);
    long operator+(long hundredths);
    long operator+(const UI_TIME &time);
    long operator-(long hundredths);
    long operator-(const UI_TIME &time);
    int operator>(UI_TIME &time);
    int operator>=(UI_TIME &time);
    int operator<(UI_TIME &time);
    int operator<=(UI_TIME &time);
    long operator++(void);
    long operator--(void);
    int operator==(UI_TIME &time);
    int operator!=(UI_TIME &time);
    void operator+=(long hundredths);
    void operator-=(long hundredths);
};
```

- *amPtr* is a pointer to the symbols that are used to denote A.M. time. The default *amPtr* points to " a.m."

- *pmPtr* is a pointer to the symbols that are used to denote P.M. time. The default *pmPtr* points to " p.m."

UI_TIME::UI_TIME

Syntax

```
#include <ui_gen.hpp>

UI_TIME(void);
    or
UI_TIME(const UI_TIME &time);
    or
UI_TIME(int hour, int minute, int second = 0, int hundredth = 0);
    or
UI_TIME(const char *string, TMF_FLAGS tmFlags = TMF_NO_FLAGS);
    or
UI_TIME(int packedTime);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors return a pointer to a new UI_TIME class object.

The first overloaded constructor takes no arguments. It sets the time information according to the system's time.

The second overloaded constructor is a copy constructor that takes a previously constructed UI_TIME class object to specify the default time.

- $time_{in}$ is a reference pointer to a previously constructed time.

The third overloaded constructor uses integer arguments to specify the default time.

- $hour_{in}$ is the hour. This argument must be in a range from 0 to 23.
- $minute_{in}$ is the minute. This argument must be in a range from 0 to 59.
- $second_{in}$ is the second. This argument must be in a range from 0 to 59.

- *hundredth*$_{in}$ is the hundredths of second. This argument must be in a range from 0 to 99.

The fourth overloaded constructor uses an ASCII string argument to specify the default time.

- *string*$_{in}$ is an ASCII string that contains the time information.

- *tmFlags*$_{in}$ gives information on how to interpret the time string. The following flags (declared in **UI_GEN.HPP**) override the country dependent information (supplied by all DOS based systems):

  TMF_COLON_SEPARATOR—Separates each time variable with a colon. For example, 12 p.m. would be displayed as 12:00pm.

  TMF_HUNDREDTHS—Interprets a hundredths value for the UI_TIME object. For example, if time were "12:15:10:09pm" and the TMF_HUNDREDTHS were set, the value "12" would be interpreted as hours, the value "15" would be interpreted as minutes, "10" would be interpreted as seconds, and the "09" would be interpreted as hundredths of seconds.

  TMF_LOWER_CASE—Converts the time to lower-case.

  TMF_NO_FLAGS—Does not associate any special flags with the UI_TIME class object. In this case, the ASCII time will be interpreted using the default country information. This flag should not be used in conjunction with any other TMF flag. This is the default argument if no other argument is specified.

  TMF_NO_HOURS—Does not interpret an hour value for the UI_TIME object. For example, if time were "12:15" and the TMF_NO_HOURS were set, the value "12" would be interpreted as minutes and "15" would be interpreted as seconds.

  TMF_NO_MINUTES—Does not interpret a minute value for the UI_TIME object. For example, if time were "12:15pm" and the TMF_NO_MINUTES were set, the value "12" would be interpreted as seconds and the value "15" would be interpreted as hundredths of seconds.

  TMF_NO_SEPARATOR—No separator characters are used to delimit the time values.

TMF_SECONDS—Interprets a second value for the UI_TIME object. For example, if time were "12:15:10pm" and the TMF_SECONDS were set, the value "12" would be interpreted as hours, the value "15" would be interpreted as minutes and "10" would be interpreted as seconds.

TMF_SYSTEM—Fills a blank time with the system time. For example, if a blank ASCII time value were entered by the end-user and the TMF_SYSTEM flag were set, then the time would be set to the current system time.

TMF_TWELVE_HOUR—Forces the time to be displayed and interpreted using a 12 hour clock, regardless of the default country information.

TMF_TWENTY_FOUR_HOUR—Forces the time to be displayed and interpreted using a 24 hour clock, regardless of the default country information.

TMF_UPPER_CASE—Converts the time to upper-case.

TMF_ZERO_FILL—Forces the hour, minute and second values to be zero filled when their values are less than 10.

The fifth overloaded constructor uses a packed integer argument to specify the default time.

- *packedTime*$_{in}$ is a packed representation of the time (whose format is the same as the MS-DOS file times). This argument is packed according to the following bit pattern:

  bits 0-4 specify the seconds divided by 2 (e.g., a value of 5 means 10 seconds),
  bits 5-10 specify the minutes(0 through 59) and
  bits 11-15 specify the hours (0 through 59).

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_TIME time1;                              // System time initialization.
    UI_TIME time2(12, 0, 0);                    // Integer initialization.
    UI_TIME *time3 = new UI_TIME("12:00:00pm"); // String initialization.
    .
    .
    .

}
```

UI_TIME::~UI_TIME

Syntax

#include <ui_gen.hpp>

virtual ~UI_TIME(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_TIME object.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_TIME time1;                                  // System time initialization.
    UI_TIME time2(12, 0, 0);                        // Integer initialization.
    UI_TIME *time3 = new UI_TIME("12:00:00pm"); // String initialization.
    .
    .
    .

    delete time3;
    // The destructor for time1 and time2 is automatically called
    // when the scope of this function ends.
}
```

UI_TIME::Export

Syntax

#include <ui_gen.hpp>

void Export(int **hour*, int **minute*, int **second* = NULL, int **hundredth* = NULL);
or

void Export(char **string*, TMF_FLAGS *tmFlags*);

or

void Export(int **packedTime*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first overloaded function returns time information through the four integer arguments.

- $hour_{in/out}$ is a pointer to the hour. If this argument is NULL, no hour information is returned. Otherwise, this argument will always be set within a range from 0 to 23.

- $minute_{in/out}$ is a pointer to the minutes. If this argument is NULL, no minute information is returned. Otherwise, this argument will always be set within a range from 0 to 59.

- $second_{in/out}$ is a pointer to the seconds. If this argument is NULL, no second information is returned. Otherwise, this argument will always be set within a range from 0 to 59.

- $hundredth_{in/out}$ is a pointer to the hundredths. If this argument is NULL, no hundredths information is returned. Otherwise, this argument will always be set within a range from 0 to 99.

The second overloaded function returns the time information through the *string* argument.

- $string_{in/out}$ is a pointer to a string that gets the ASCII formatted time. This string must be long enough to hold the string (including the trailing NULL byte).

- $tmFlags_{in}$ gives formatting information about the return ASCII time. The following flags (declared in **UI_GEN.HPP**) override the country dependent information (supplied by all DOS based systems):

  TMF_COLON_SEPARATOR—Separates each time variable with a colon.

```
12:00
13:00:00
12:00 a.m.
```

TMF_HUNDREDTHS—Includes the hundredths value in the time. (By default the hundredths value is not included.)	`1:05:00:00`
	`23:15:05:99`
	`7:45:59:00 a.m.`

TMF_LOWER_CASE—Converts the time to lower-case.	`12:00 p.m.` `1:00 a.m.`

TMF_NO_FLAGS—Does not associate any special flags with the format function. In this case, the ASCII time will be formatted using the default country information. This flag should not be used in conjunction with any other TMF flag.	`12:00`
	`13:00:00`
	`12:00 a.m.`

TMF_NO_SEPARATOR—Does not use any separator characters to delimit the time values.	`1200` `130000`

TMF_SECONDS—Includes the seconds value in the time. (By default the seconds value is not included.)	`12:00:05 a.m.`
	`1:13:25`
	`16:00:00`

TMF_TWELVE_HOUR—Forces the time to be formatted using a 12 hour clock, regardless of the default country information.	`12:00 a.m.`
	`1:00 p.m.`
	`5:00 p.m.`

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

TMF_UPPER_CASE—Converts the time to upper-case.	`12:00 P.M.` `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.	`01:10 a.m`
	`13:05:03`
	`01:01 p.m.`

The third overloaded function returns time information through a packed integer argument.

- *packedTime*$_{in/out}$ is a packed representation of the time (whose format is the same as the MS-DOS file times). This argument is packed according to the following bit pattern:

  bits 0-4 specify the seconds divided by 2 (e.g., a value of 5 means 10 seconds),

bits 5-10 specify the minutes(0 through 59) and
bits 11-15 specify the hours (0 through 59).

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_TIME time;        // Use a system time.

    int hour, minute, second;
    time.Export(&hour, &minute, &second);
    printf("Integer time value: hour-%d, minute-%d, second-%d\n",
        hour, minute, second);

    char asciiTime[128];
    time.Export(asciiTime, TMF_NO_FLAGS);
    printf("ASCII time value: %s", asciiTime);

    // The destructor for time is automatically called when the
    // scope of this function ends.
}
```

UI_TIME::Import

Syntax

#include <ui_gen.hpp>

TMI_RESULT Import(void);
or
TMI_RESULT Import(const UI_TIME &*time*);
or
TMI_RESULT Import(int *hour*, int *minute*, int *second* = 0, int *hundredth* = 0);
or
TMI_RESULT Import(const char **string*, TMF_FLAGS *tmFlags*);
or
TMI_RESULT Import(int *packedTime*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first overloaded function sets the time information according to the system time.

- *returnValue*$_{out}$ is the result of the import operation. The following values are accepted:

  TMI_OK—The time was successfully imported.

  TMI_INVALID—The time was invalid or was in an invalid format.

  TMI_OUT_OF_RANGE—The time was out of the valid range for times.

  TMI_VALUE_MISSING—All of the required values were not present.

The second overloaded function copies the time information from the *time* reference argument.

- *returnValue*$_{out}$ is the result of the import operation. See the first function for possible values.
- *time*$_{in}$ is a reference pointer to a previously constructed time.

The third overloaded function sets the time information according to specified integer arguments.

- *returnValue*$_{out}$ is the result of the import operation. See the first function for possible values.
- *hour*$_{in}$ is the hour. This argument must be in a range from 0 to 23.
- *minute*$_{in}$ is the minute. This argument must be in a range from 0 to 59.
- *second*$_{in}$ is the second. This argument must be in a range from 0 to 59.
- *hundredth*$_{in}$ is the hundredths of second. This argument must be in a range from 0 to 99.

The fourth overloaded function sets the time information according to an ASCII string.

- *returnValue*$_{out}$ is the result of the import operation. See the first function for possible values.

- *string*$_{in}$ is a pointer to the ASCII time.

- *tmFlags*$_{in}$ gives information on how to interpret the time string. The following flags (declared in **UI_GEN.HPP**) override the country dependent information (supplied by all DOS based systems):

  TMF_HUNDREDTHS—Interprets a hundredths value for the UI_TIME object. For example, if time were "12:15:10:09pm" and the TMF_HUNDREDTHS were set, the value "12" would be interpreted as hours, the value "15" would be interpreted as minutes, "10" would be interpreted as seconds, and the "09" would be interpreted as hundredths of seconds.

  TMF_NO_FLAGS—Does not associate any special flags with the UI_TIME class object. In this case, the ASCII time will be interpreted using the default country information. This flag should not be used in conjunction with any other TMF flag. This is the default argument if no other argument is specified.

  TMF_NO_HOURS—Does not interpret an hour value for the UI_TIME object. For example, if time were "12:15" and the TMF_NO_HOURS were set, the value "12" would be interpreted as minutes and "15" would be interpreted as seconds.

  TMF_NO_MINUTES—Does not interpret a minute value for the UI_TIME object. For example, if time were "12:15pm" and the TMF_NO_MINUTES were set, the value "12" would be interpreted as seconds and the value "15" would be interpreted as hundredths of seconds.

  TMF_SECONDS—Interprets a second value for the UI_TIME object. For example, if time were "12:15:10pm" and the TMF_SECONDS were set, the value "12" would be interpreted as hours, the value "15" would be interpreted as minutes and "10" would be interpreted as seconds.

The fifth overloaded function sets the time information through a packed integer argument.

- *returnValue*$_{out}$ is the result of the import operation. See the first function for possible values.

- *packedTime*$_{in/out}$ is a packed representation of the time (whose format is the same as the MS-DOS file times). This argument is packed according to the following bit pattern:

bits 0-4 specify the seconds divided by 2 (e.g., a value of 5 means 10 seconds),
bits 5-10 specify the minutes(0 through 59) and
bits 11-15 specify the hours (0 through 59).

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_TIME time;       // Initialize a system time.

    // Import the time in various forms, then print out
    // the results.
    char asciiTime[128];
    time.Import(20, 0);
    time.Export(asciiTime, TMF_NO_FLAGS);
    printf("ASCII time value: %s\n", asciiTime);

    time.Import("8:00 p.m.");
    time.Export(asciiTime, TMF_TWENTY_FOUR_HOUR);
    printf("ASCII time value: %s\n", asciiTime);

    // The destructor for time is automatically called when the
    // scope of this function ends.
}
```

UI_TIME::operator =

Syntax

#include <ui_gen.hpp>

long operator = (long *hundredths*);
or
long operator = (const UI_TIME &*time*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first operator overload assigns the value specified by *hundredths* to the UI_TIME object.

- *returnValue*$_{out}$ is the number of hundredths of seconds in the resulting time. This raw value is only used to transfer the current time to another UI_TIME object.

- *hundredths*$_{in}$ is the time, given in the number of hundredths of seconds, to be assigned to the UI_TIME object. This value does not necessarily correspond to any twenty-four hour time, but could be used to denote a period of time such as 1000 hours.

The second operator overload assigns the value specified by *time* to the UI_TIME object.

- *returnValue*$_{out}$ is the number of hundredths of seconds in the resulting time. This raw value is only used to transfer the current time to another UI_TIME object.

- *time*$_{in}$ is the time, specified by another UI_TIME object, to be assigned to the UI_TIME object.

Example

```
#include <ui_gen.hpp>

AddOneHour(UI_TIME currentTime, UI_TIME &nextHour, UI_TIME &hourAfterNext)
{
    long oneHour = 360000L;
    UI_TIME twoHours(2, 0);

    // Adding 1 hour to the current time gives the next hour.
    nextHour = currentTime + oneHour;

    // Adding 2 hour to the current time gives the following hour.
    hourAfterNext = currentTime + twoHours;
}
```

UI_TIME::operator +

Syntax

#include <ui_gen.hpp>

long operator + (long *hundredths*);
or
long operator + (const UI_TIME &*time*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first operator overload adds the value *hundredths* to the UI_TIME object.

- *returnValue*$_{out}$ is the number of hundredths of seconds resulting from the addition operation. This raw value is only used to transfer the current time to another UI_TIME object.

- *hundredths*$_{in}$ is the number of hundredths of seconds to be added to the UI_TIME object. This value does not necessarily correspond to any twenty-four hour time, but could be used to denote a period of time such as 1000 hours.

The second operator overload adds the value of *time* to the UI_TIME object.

- *returnValue*$_{out}$ is the value resulting from the addition operation. This raw value is only used to transfer the current time to another UI_TIME object.

- *time*$_{in}$ is another UI_TIME object to be added to the UI_TIME object.

Example

```
#include <ui_gen.hpp>

AddOneHour(UI_TIME currentTime, UI_TIME &nextHour, UI_TIME &hourAfterNext)
{
    long oneHour = 360000L;
    UI_TIME twoHours(2, 0);

    // Adding 1 hour to the current time gives the next hour.
    nextHour = currentTime + oneHour;

    // Adding 2 hours to the current time gives the following hour.
    hourAfterNext = currentTime + twoHours;
}
```

UI_TIME::operator -

Syntax

#include <ui_gen.hpp>

long operator - (long *hundredths*);
or
long operator - (const UI_TIME &*time*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The first operator overload subtracts the value *hundredths* from the UI_TIME object.

- *returnValue*$_{out}$ is the number of hundredths of seconds resulting from the subtraction operation. This raw value is only used to transfer the current time to another UI_TIME object.

- *hundredths*$_{in}$ is the number of hundredths of seconds to be subtracted from the UI_TIME object. This value does not necessarily correspond to any twenty-four hour time, but could be used to denote a period of time such as 1000 hours.

The second operator overload subtracts the value of *time* from the UI_TIME object.

- *returnValue*$_{out}$ is the value resulting from the subtraction operation. This raw value is only used to transfer the current time to another UI_TIME object.

- *time*$_{in}$ is another UI_TIME object to be subtracted from the UI_TIME object.

Example

```
#include <ui_gen.hpp>

SubtractOneHour(UI_TIME currentTime, UI_TIME &previousHour,
    UI_TIME &twoHoursBefore)
{
    long oneHour = 360000L;
```

```
    UI_TIME twoHours(2, 0);

    // Subtracting 1 hour from the current time gives the previous hour.
    previousHour = currentTime - oneHour;

    // Subtracting 2 hours from the current time gives the hour
    // two hours previous.
    twoHoursBefore = currentTime - twoHours;
}
```

UI_TIME::operator >

Syntax

#include <ui_gen.hpp>

int operator > (UI_TIME &*time*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator overload determines whether the UI_TIME object is chronologically greater than the time specified by *time*.

- $returnValue_{out}$ is TRUE if the UI_TIME object is chronologically greater than *time*. Otherwise, *returnValue* is FALSE.

- $time_{in}$ is the UI_TIME object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_TIME currentTime; // Initialize a system time.
    UI_TIME midnight("12:00am");

    // Check the time.
    if (currentTime == midnight)
        printf("It's exactly midnight.\n");
    else if (currentTime > midnight)
```

```
        printf("We're in the wee hours of the morning.\n");
    else
        printf("It's still late night.\n");
}
```

UI_TIME::operator >=

Syntax

#include <ui_gen.hpp>

int operator >= (UI_TIME &*time*);

Portability

This function is available on the following environments:

■ DOS . ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator overload determines whether the UI_TIME object is chronologically greater than or equal to the time specified by *time*.

- $returnValue_{out}$ is TRUE if the UI_TIME object is chronologically greater than or equal to *time*. Otherwise, *returnValue* is FALSE.
- $time_{in}$ is the UI_TIME object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_TIME currentTime; // Initialize a system time.
    UI_TIME midnight("12:00am");

    // Check the time.
    if (currentTime >= midnight)
        printf("Tomorrow is here.\n");
    else
        printf("It's still late night.\n");
}
```

UI_TIME::operator <

Syntax

#include <ui_gen.hpp>

int operator < (UI_TIME &*time*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator overload determines whether the UI_TIME object is chronologically less than the time specified by *time*.

- *returnValue*$_{out}$ is TRUE if the UI_TIME object is chronologically less than *time*. Otherwise, *returnValue* is FALSE.

- *time*$_{in}$ is the UI_TIME object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_TIME currentTime;    // Initialize a system time.
    UI_TIME midnight("12:00am");

    // Check the time.
    if (currentTime < midnight)
        printf("It's still late night.\n");
    else
        printf("Tomorrow is here.\n");
}
```

UI_TIME::operator <=

Syntax

#include <ui_gen.hpp>

int operator <= (UI_TIME &*time*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator overload determines whether the UI_TIME object is chronologically less than or equal to the time specified by *time*.

- *returnValue*$_{out}$ is TRUE if the UI_TIME object is chronologically less than or equal to *time*. Otherwise, *returnValue* is FALSE.

- *time*$_{in}$ is the UI_TIME object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_TIME currentTime; // Initialize a system time.
    UI_TIME midnight("12:00am");

    // Check the time.
    if (midnight <= currentTime)
        printf("It's still late night.\n");
    else
        printf("Tomorrow is here.\n");
}
```

UI_TIME::operator ++

Syntax

#include <ui_gen.hpp>

int operator ++ (void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator overload increments the UI_TIME by one hundredth of a second.

- *returnValue*$_{out}$ is the number of hundredths of seconds after the UI_TIME object has been incremented. This raw value is only used to update the current UI_TIME object.

Example

```
#include <ui_gen.hpp>
AdvanceCurrentTime(UI_TIME &currentTime)
{
    // Advance the current time.
    currentTime++;
}
```

UI_TIME::operator --

Syntax

#include <ui_gen.hpp>

int operator -- (void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator overload decrements the UI_TIME by one hundredth of a second.

- *returnValue*$_{out}$ is the number of hundredths of seconds after the UI_TIME object has been decremented. This raw value is only used to update the current UI_TIME object.

Example

```
#include <ui_gen.hpp>

DecrementCurrentTime(UI_TIME &currentTime)
{
    // Advance the current Time.
    currentTime--;
}
```

UI_TIME::operator +=

Syntax

#include <ui_gen.hpp>

void operator += (long *hundredths*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator adds *hundredths* to the UI_TIME object and copies the result back into the UI_TIME object.

- *hundredths*$_{in}$ is the number of hundredths of seconds to be added to the UI_TIME object. This value does not necessarily correspond to any 24 hour clock, but could be used to denote a period of time such as 1000 seconds.

Example

```
#include <ui_gen.hpp>

AddOneHour(UI_TIME *currentTime)
{
    long oneHour = 360000L;

    // Add 1 hour.
    *currentTime += oneHour;
}
```

UI_TIME::operator -=

Syntax

#include <ui_gen.hpp>

void operator -= (long *hundredths*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator subtracts *hundredths* from the UI_TIME object and copies the result back into the UI_TIME object.

- *hundredths*$_{in}$ is the time, given in hundredths of seconds to be subtracted from the UI_TIME object. This value does not necessarily correspond to a 24 hour clock, but could be used to denote a period of time such as 1000 seconds.

Example

```
#include <ui_gen.hpp>

SubtractHour(UI_TIME *currentTime)
{
    long oneHour = 360000L;
    // Subtract 1 hour.
    *currentTime -= oneHour;
}
```

UI_TIME::operator ==

Syntax

#include <ui_gen.hpp>

int operator == (UI_TIME &*time*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator overload determines whether the UI_TIME object is chronologically equal to the time specified by *time*.

- $returnValue_{out}$ is TRUE if the UI_TIME object is chronologically equal to *time*. Otherwise, *returnValue* is FALSE.
- $time_{in}$ is the UI_TIME object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_TIME currentTime;    // Initialize a system time.
    UI_TIME midnight("12:00am");

    // Check the time.
    if (currentTime == midnight)
        printf("It's exactly midnight.\n");
```

```
    else if (currentTime > midnight)
        printf("We're in the wee hours of the morning.\n");
    else
        printf("It's still late night.\n");
}
```

UI_TIME::operator !=

Syntax

#include <ui_gen.hpp>

int operator != (UI_TIME &*time*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This operator overload determines whether the UI_TIME object is chronologically not equal to the time specified by *time*.

- $returnValue_{out}$ is TRUE if the UI_TIME object is chronologically not equal to *time*. Otherwise, *returnValue* is FALSE.

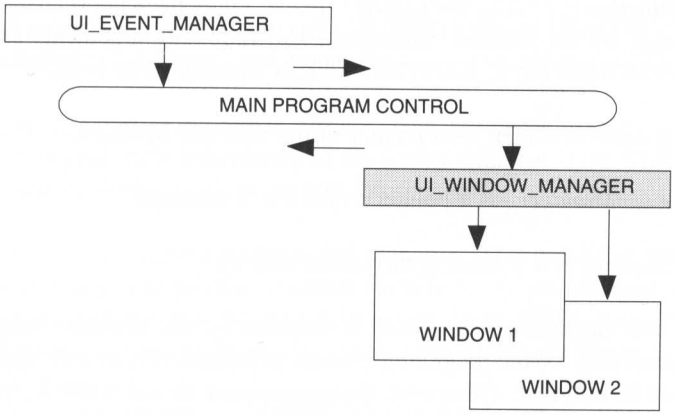
- $time_{in}$ is the UI_TIME object to be compared.

Example

```
#include <ui_gen.hpp>

ExampleFunction()
{
    UI_TIME currentTime; // Initialize a system time.
    UI_TIME startTime("12:00am");

    // Check the time.
    if ((currentTime != startTime) && (currentTime < startTime))
        printf("It's still not time yet!\n");
}
```

CHAPTER 39 – UI_WINDOW_MANAGER

The UI_WINDOW_MANAGER class serves as the control unit for windows that are attached to the screen display. The graphic illustration below shows the conceptual operation of the Window Manager within the library:



The controlling portion of the UI_WINDOW_MANAGER class contains a list of all windows attached to the screen. These windows receive message information from the Window Manager during an application program. This information is routed to each window according to its order in the Window Manager's list.

The UI_WINDOW_MANAGER class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UI_WINDOW_MANAGER : public UIW_WINDOW
{
    friend class UI_WINDOW_OBJECT;
public:
    // Members described in UI_WINDOW_MANAGER reference chapter.
    UI_WINDOW_MANAGER(UI_DISPLAY *display, UI_EVENT_MANAGER *eventManager,
        EVENT_TYPE (*exitFunction)(UI_DISPLAY *display,
        UI_EVENT_MANAGER *eventManager,
        UI_WINDOW_MANAGER *windowManager) = NULL);
    virtual ~UI_WINDOW_MANAGER(void);
    void Center(UI_WINDOW_OBJECT *object);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    EVENT_TYPE (*exitFunction)(UI_DISPLAY *display,
        UI_EVENT_MANAGER *eventManager, UI_WINDOW_MANAGER *windowManager);

    // Members described in UI_LIST reference chapter.
    UI_WINDOW_OBJECT *Add(UI_WINDOW_OBJECT *object);
```

```
    UI_WINDOW_OBJECT *Subtract(UI_WINDOW_OBJECT *object);
    UI_WINDOW_MANAGER &operator+(UI_WINDOW_OBJECT *object);
    UI_WINDOW_MANAGER &operator-(UI_WINDOW_OBJECT *object);
};
```

- *exitFunction* is a programmer defined function that is called whenever the Window Manager receives the L_EXIT_FUNCTION message. For example, if the programmer wants to confirm whether the end user really wants to exit the program, *exitFunction* would be used to display a confirmation window to the end user. If *exitFunction* is NULL, the L_EXIT_FUNCTION message is changed to an L_EXIT message by the Window Manager. The following arguments are passed to *exitFunction* when the L_EXIT_FUNCTION message is received:

  windowManager$_{in}$ is a pointer to the Window Manager.

  eventManager$_{in}$ is a pointer to the Event Manager.

  display$_{in}$ is a pointer to the screen display.

  The exit function's *returnValue* is returned by the Window Manager to the main program loop. If the programmer wants to immediately exit the application, L_EXIT should be returned. Otherwise, the programmer should return L_EXIT_FUNCTION or any other valid system or logical event. To have *exitFunction* get called when a certain window (the application's main window, for instance) is deleted, the Window Manager's *screenID* must be set equal to the window's *screenID*. The following piece of code demonstrates this:

```
*windowManager
    + window;

windowManager->screenID = window->screenID;
```

UI_WINDOW_MANAGER::UI_WINDOW_MANAGER

Syntax

#include <ui_win.hpp>

UI_WINDOW_MANAGER(UI_DISPLAY **display*,
UI_EVENT_MANAGER **eventManager*,
EVENT_TYPE (**exitFunction*) (UI_DISPLAY *display,
UI_EVENT_MANAGER *eventManager,
UI_WINDOW_MANAGER *windowManager) = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UI_WINDOW_MANAGER class object. It should be called after the display and Event Manager classes have been called.

- *display*$_{in}$ is a pointer to the screen display. This pointer is used by window objects when they display their information to the screen (e.g., the UIW_TEXT class object uses the screen display to show its text information).

- *eventManager*$_{in}$ is a pointer to the Event Manager. This pointer is used by window objects when they send private communication to the Event Manager, or when input device information needs to be changed (e.g., the UIW_TEXT class object sends the mouse device messages to change its cursor state to the edit figure ' | ').

- *exitFunction*$_{in}$ is a programmer defined function that is called whenever the Window Manager receives the L_EXIT_FUNCTION message. For example, if the programmer wants to confirm whether the end user really wants to exit the program, *exitFunction* would be used to display a confirmation window to the end user. If *exitFunction* is NULL, the L_EXIT_FUNCTION message is changed to an L_EXIT message by the Window Manager. The following arguments are passed to *exitFunction* when the L_EXIT_FUNCTION message is received:

  windowManager$_{in}$ is a pointer to the Window Manager.

  eventManager$_{in}$ is a pointer to the Event Manager.

  display$_{in}$ is a pointer to the screen display.

  The exit function's *returnValue* is returned by the Window Manager to the main program loop. If the programmer wants to immediately exit the application, L_EXIT should be returned. Otherwise, the programmer should return L_EXIT_FUNCTION or any other valid system or logical event. To have *exitFunction* get called when a certain window (the application's main window, for instance) is deleted, the Window Manager's *screenID* must be set equal to the window's *screenID*. The following piece of code demonstrates this:

```
        *windowManager
            + window;

        windowManager->screenID = window->screenID;
```

Example

```
#include <ui_win.hpp>

main()
{
    // Initialize the system.
    UI_DISPLAY *display = new UI_TEXT_DISPLAY();
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANAGER(display,
        eventManager);
    .
    .
    .

    // Restore the system.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

UI_WINDOW_MANAGER::~UI_WINDOW_MANAGER

Syntax

#include <ui_win.hpp>

virtual ~UI_WINDOW_MANAGER(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UI_WINDOW_-MANAGER object and destroys the class information of any window object that remains attached to the Window Manager.

Example

```
#include <ui_win.hpp>

main()
{
    // Initialize the system.
    UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    UI_WINDOW_MANAGER *windowManager = new UI_WINDOW_MANAGER(display,
        eventManager);
    .
    .
    .

    // Restore the system.
    delete windowManager;
    delete eventManager;
    delete display;
    return (0);
}
```

UI_WINDOW_MANAGER::Center

Syntax

#include <ui_win.hpp>

void Center(UI_WINDOW_OBJECT *_object_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function centers an object on the screen. Only objects that either already are attached to the Window Manager, or will be attached to the Window Manager, should be centered using this function.

- $object_{in}$ is a pointer to the object that is to be centered on the screen. This must only be an object that is, or will be, attached to the Window Manager.

UI_WINDOW_MANAGER::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT *&event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine transfers event information received from the Event Manager to the Window Manager. All user input and system information is put in the Event Manager's input queue. The event is received by calling the **UI_EVENT_MANAGER::Get( )** function. When an event is received, it is sent to the Window Manager for processing. The **UI_WINDOW_MANAGER::Event( )** function determines to which window the event should go and then the event is passed to that window for processing. For example, the keyboard event E_KEY is passed to the current window where it is in turn sent to the affected object on the window. If the event was not handled by the receiving window (or any of its objects), the Window Manager will then try to interpret it. If the Window Manager then cannot handle the event, a return code of S_UNKNOWN is returned. Otherwise, a return code reflecting the outcome of the event is returned.

- *returnValue*$_{out}$ is the type of action that the Window Manager used to process the event passed to it. Normally this is the same value that is provided in the *event.type* variable. On occasion, however, this return code will have a significant meaning. See "Appendix B—System Events" for a complete listing of system events and "Appendix C—Logical Events" for a complete listing of logical events. The following return codes (declared in **UI_EVT.HPP**) should be handled in a different manner:

  L_EXIT—The Window Manager received an event that either mapped to the L_EXIT command, or an action was performed that caused the Window Manager to generate the L_EXIT command. If this command is received by the programmer, program execution should be discontinued.

L_EXIT_FUNCTION—The Window Manager received a request to exit the application program. Before this message is changed to L_EXIT, the Window Manager's *exitFunction* (if any) is invoked.

S_CLOSE—The current object in the Window Manager's list is subtracted and then deleted.

S_ERROR—The Window Manager detected an error while performing an operation on the last event.

S_NO_OBJECT—There are no objects in the Window Manager's list. This message is sent back to the programmer whenever the message is object specific but no object is attached to the Window Manager.

S_UNKNOWN—The event passed to the Window Manager was not recognized by the Window Manager or by any window attached to the screen display.

- $event_{in}$ is the event to be processed by the Window Manager. This event can be generated by the programmer or may be received from the Event Manager using the **UI_EVENT_MANAGER::Get( )** routine.

Example

```
#include <ui_win.hpp>

main()
{
    .
    .
    .

    // Get events until the L_EXIT logical key is entered.
    EVENT_TYPE ccode;
    do
    {
        UI_EVENT event;
        eventManager->Get(event, Q_NORMAL);
        ccode = windowManager->Event(event);
    } while (ccode != L_EXIT);
    .
    .
    .

}
```

UI_WINDOW_MANAGER::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is NULL.

- *request*$_{in}$ is a request to get or set information associated with the object. The following requests (defined in **UI_WIN.HPP**) are recognized by the Window Manager:

  GET_NUMBERID_OBJECT—Returns a pointer to an object whose *numberID* matches the value in *data*, if one exists in the Window Manager's list. If no object has a *numberID* that matches *data*, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined USHORT.

  GET_STRINGID_OBJECT—Returns a pointer to an object whose *stringID* matches the character string in *data*, if one exists in the Window Manager's list. If no object has a *stringID* that matches *data*, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined string (i.e., char *).

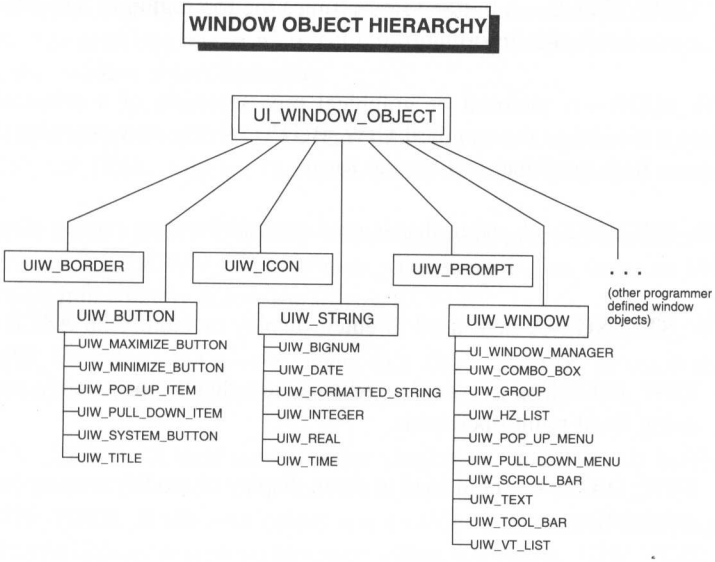
- *data*$_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

Example

```
#include <ui_win.hpp>

UI_WINDOW_OBJECT *GetObject(char *name, UI_WINDOW_MANAGER *windowManager)
{
    // Find the window object given a name and return a pointer to it.
    return (windowManager->Information(GET_STRINGID_OBJECT, name));
}
```

CHAPTER 40 – UI_WINDOW_OBJECT

The UI_WINDOW_OBJECT class is the base class to all window objects and defines basic information associated with window objects (e.g., borders, buttons, menus). It should not be used as a constructed class. Rather, derived classes, such as UIW_BORDER, UIW_BUTTON or UIW_WINDOW must be used. The figure below shows the window object hierarchy:



Classes derived from the UI_WINDOW_OBJECT base class include:

UIW_BORDER—An outlining border drawn around a window.

UIW_BUTTON—A rectangular region of the screen that, when selected, performs run-time operations specified by the programmer. This class includes check boxes, radio buttons and buttons with bitmaps.

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

UIW_MINIMIZE_BUTTON—A button that, when selected, reduces its parent window to an icon. For this to occur, a minimize icon must have been previously added to the parent window.

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

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_TITLE—A button that occupies the top region of a window and contains a window's title information.

UIW_ICON—A pictorial or graphical representation of a selectable item. This object is similar to the standard UIW_BUTTON object, except that the information contains both graphical and textual form.

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

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

UIW_BIGNUM—A field used to enter, display or modify numerical values using fixed-point operations.

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

UIW_FORMATTED_STRING—A field used to enter, display or modify programmer specified ASCII string buffers (e.g., phone numbers, social security numbers).

UIW_INTEGER—A field used to enter, display or modify integer values.

UIW_REAL—A field used to enter, display or modify real number values.

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

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

UI_WINDOW_MANAGER—A control unit for windows attached to the display.

UIW_COMBO_BOX—An object that is a combination of a list box and an entry field. The object is represented as an entry field and a button. A scrollable list box appears when the button is clicked or when the down arrow key is pressed. Once an object is selected from the list, it is copied into the initial entry field and the list box disappears.

UIW_GROUP—An object that provides a method of grouping window objects within a window.

UIW_HZ_LIST—A horizontal list of related, non-editable items. These items are organized into one or more columns and may be any of the objects described in the window object hierarchy.

UIW_POP_UP_MENU—An object that provides a method of grouping UIW_-POP_UP_ITEM objects. The items in this menu are displayed on multiple lines.

UIW_PULL_DOWN_MENU—An object that provides a method of grouping UIW_PULL_DOWN_ITEM objects. The items in this menu are displayed in a region immediately below the title.

UIW_SCROLL_BAR—An object that allows the user to scroll though information that cannot be fully represented on the screen.

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

UIW_TOOL_BAR—An object that provides a method of grouping a series of window objects in a menu structure within a window. UIW_TOOL_BAR may contain any mixture of window objects and resides in the upper portion of the window. If the parent window contains a UIW_PULL_DOWN_MENU, the UIW_TOOL_BAR will be positioned below the menu.

UIW_VT_LIST—A vertical list of related non-editable items. These items are organized into a single column and may be any of the objects described in the window object hierarchy.

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 and window objects are attached to the Window Manager, or to a parent window, at run-time by the programmer. Once a window or window object is attached, it receives event information from the Window Manager.

The public and protected members of the UI_WINDOW_OBJECT class (declared in **UI_WIN.HPP**) are:

```
class EXPORT UI_WINDOW_OBJECT : public UI_ELEMENT
{
    friend class EXPORT UIW_WINDOW;
    friend class EXPORT UIW_COMBO_BOX;
    friend class EXPORT UIF_CONTROL;
    friend class EXPORT UIF_WINDOW_OBJECT;
public:
    // Forward declaration of classes used by UI_WINDOW_OBJECT.
    friend class EXPORT UI_WINDOW_MANAGER;
    friend class EXPORT UI_ERROR_SYSTEM;
    friend class EXPORT UI_HELP_SYSTEM;

    static int repeatRate;
    static int doubleClickRate;
    static WOS_STATUS defaultStatus;
    static UI_DISPLAY *display;
    static UI_EVENT_MANAGER *eventManager;
    static UI_WINDOW_MANAGER *windowManager;
    static UI_ERROR_SYSTEM *errorSystem;
    static UI_HELP_SYSTEM *helpSystem;
    static UI_STORAGE *defaultStorage;
    static UI_ITEM *objectTable;
    static UI_ITEM *userTable;

    UI_EVENT_MAP *eventMapTable;
    UI_EVENT_MAP *hotKeyMapTable;
    UI_PALETTE_MAP *paletteMapTable;

    SCREENID screenID;
    WOF_FLAGS woFlags;
    WOAF_FLAGS woAdvancedFlags;
    WOS_STATUS woStatus;
    UI_REGION true;
    UI_REGION relative;
    UI_WINDOW_OBJECT *parent;
    UI_HELP_CONTEXT helpContext;

    UIF_FLAGS userFlags;
    UIS_STATUS userStatus;
    void *userObject;
    EVENT_TYPE (*userFunction)(UI_WINDOW_OBJECT *object, UI_EVENT &event,
        EVENT_TYPE ccode);

    EVENT_TYPE UserFunction(const UI_EVENT &event, EVENT_TYPE ccode);
    virtual ~UI_WINDOW_OBJECT(void);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    UI_WINDOW_OBJECT *Get(const char *name);
    UI_WINDOW_OBJECT *Get(NUMBERID numberID);
    unsigned HotKey(unsigned hotKey = 0);
    unsigned HotKey(char *text);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);
    int Inherited(OBJECTID matchID);
    EVENT_TYPE LogicalEvent(const UI_EVENT &event, OBJECTID currentID = 0);
    UI_PALETTE *LogicalPalette(LOGICAL_EVENT logicalEvent,
        OBJECTID currentID = 0);
    NUMBERID NumberID(NUMBERID numberID = 0);
    void RegionConvert(UI_REGION &region, int absolute);
    OBJECTID SearchID(void);
    char *StringID(const char *stringID = NULL);
    virtual int Validate(int processError = TRUE);
```

```
    // Persistence members.
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL);
    UI_WINDOW_OBJECT(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL);

    // List members documented in UI_LIST chapter.
    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);

protected:
    OBJECTID searchID;
    NUMBERID numberID;
    char stringID[32];
    OBJECTID windowID[5];

    unsigned hotKey;
    LOGICAL_FONT font;
    UI_PALETTE *lastPalette;
    char *userObjectName;              // Used for storage purposes only.
    char *userFunctionName;            // Used for storage purposes only.
    UI_REGION clip;
#if defined(ZIL_MSWINDOWS) && defined(WIN32)
    DWORD dwStyle;
    WNDPROC defaultCallback;

    void RegisterObject(char *name, char *baseName,
        WNDPROC *defProcInstance, char *title, HMENU menu = 0);
#elif defined(ZIL_MSWINDOWS)
    DWORD dwStyle;
    FARPROC defaultCallback;

    void RegisterObject(char *name, char *baseName, int *offset,
        FARPROC *procInstance, FARPROC *defProcInstance, char *title,
        HMENU menu = 0);
#elif defined(ZIL_OS2)
    ULONG flStyle;
    ULONG flFlag;
    PFNWP defaultCallback;

    void RegisterObject(char *name, char *baseName, PFNWP *baseCallback,
        char *title);
#elif defined(ZIL_MOTIF)
    static Arg args[50];
    static int nargs;
    Widget shell;

    void RegisterObject(WidgetClass widgetClass,
        MOTIF_CONVENIENCE_FUNCTION convenienceFunction, EVENT_TYPE ccode,
        int useArgs = FALSE, int manage = TRUE, SCREENID _parent = NULL);
    SCREENID TopWidget(void);
#endif

    UI_WINDOW_OBJECT(int left, int top, int width, int height,
        WOF_FLAGS woFlags, WOAF_FLAGS woAdvancedFlags);
    EVENT_TYPE DrawBorder(SCREENID screenID, UI_REGION &region,
        int fillRegion, EVENT_TYPE ccode);
    virtual EVENT_TYPE DrawItem(const UI_EVENT &event, EVENT_TYPE ccode);
    EVENT_TYPE DrawShadow(SCREENID screenID, UI_REGION &region, int depth,
        int fillRegion, EVENT_TYPE ccode);
    EVENT_TYPE DrawText(SCREENID screenID, UI_REGION &region,
        const char *text, UI_PALETTE *palette, int fillRegion,
        EVENT_TYPE ccode);
    void Modify(const UI_EVENT &event);
```

```
    int NeedsUpdate(const UI_EVENT &event, EVENT_TYPE ccode);
    virtual void RegionMax(UI_WINDOW_OBJECT *object);
};
```

- *repeatRate* is the time (in hundredths of seconds) that must elapse before an event is repeated. For example, when the down arrow on a UIW_SCROLL_BAR is depressed and held, *repeatRate* determines the frequency that the repeat events are generated.

- *doubleClickRate* is the time in hundredths of seconds (e.g., 10 = 10/100 or 1/10 of a second) that is used to determine if two consecutive mouse clicks are to be interpreted as a double click.

- *defaultStatus* is the status assigned to a window object when it is first constructed. If this value is changed to a valid status, all window objects will be created with this status (e.g., setting WOS_GRAPHICS would cause all window objects to be created with pixel boundaries and sizes).

- *display* is a pointer to the screen display. This static pointer is set by the Window Manager when it is created.

- *eventManager* is a pointer to the Event Manager. This static pointer is set by the Window Manager when it is created.

- *windowManager* is a pointer to the Window Manager. This static pointer is set by the Window Manager when it is created.

- *errorSystem* is a pointer to an error system. This pointer is initially set to NULL. This is a static pointer that should be set (with UI_ERROR_SYSTEM or a derived class) by the programmer before entering the main program loop.

- *helpSystem* is a pointer to a help system. This pointer is initially set to NULL. This is a static pointer that should be set (with UI_HELP_SYSTEM or a derived class) by the programmer before entering the main program loop.

- *defaultStorage* is a pointer to the default storage file that is used to read button and icon images. This pointer is initially set to NULL. This static pointer should be set by the programmer before entering the main program loop.

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

    // Install a text display if no graphics capability.
    if (!display->installed)
    {
```

```
        delete display;
        display = new UI_TEXT_DISPLAY;
    }

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

    // Setup the default storage.
    UI_WINDOW_OBJECT::defaultStorage = new UI_STORAGE("support.dat");
    .
    .
    .

}
```

- *objectTable* is a pointer to an array of UI_ITEM structures. The UI_ITEM structures are used to hold object identifications and their corresponding New functions. In order for an object to be read in from a **.DAT** file and then be constructed, its objectID is looked up in *objectTable* and the associated New function is called. In general, **_objectTable**, created by Zinc Designer, is assigned to *objectTable* in the Designer-generated **.CPP** module.

```
UI_ITEM _objectTable[] =
{
    { ID_BORDER, &UIW_BORDER::New, NULL, WOF_NO_FLAGS },
    { ID_BUTTON, &UIW_BUTTON::New, "BUTTON", WOF_NO_FLAGS },
    { ID_GROUP, &UIW_GROUP::New, "GROUP", WOF_NO_FLAGS },
    { ID_ICON, &UIW_ICON::New, "ICON", WOF_NO_FLAGS },
    { ID_HZ_LIST, &UIW_HZ_LIST::New, "HLIST", WOF_NO_FLAGS },
    { ID_MAXIMIZE_BUTTON, &UIW_MAXIMIZE_BUTTON::New, NULL,
      WOF_NO_FLAGS },
    { ID_MINIMIZE_BUTTON, &UIW_MINIMIZE_BUTTON::New, NULL,
      WOF_NO_FLAGS },
    { ID_PROMPT, &UIW_PROMPT::New, "PROMPT", WOF_NO_FLAGS },
    { ID_SCROLL_BAR, &UIW_SCROLL_BAR::New, "SCROLL_BAR", WOF_NO_FLAGS },
    { ID_STRING, &UIW_STRING::New, "STRING", WOF_NO_FLAGS },
    { ID_SYSTEM_BUTTON, &UIW_SYSTEM_BUTTON::New, NULL, WOF_NO_FLAGS },
    { ID_TITLE, &UIW_TITLE::New, NULL, WOF_NO_FLAGS },
    { ID_WINDOW, &UIW_WINDOW::New, "WINDOW", WOF_NO_FLAGS },
    { ID_END, NULL, NULL, 0 }
};
UI_ITEM UI_WINDOW_OBJECT::objectTable = _objectTable;
```

NOTE: Initially, *objectTable* points to a default table (contained in **G_JUMP.CPP**). The default table has all of the entries commented out so that their code is not automatically linked into the library. To use any of these entries, make a copy of the table and un-comment the desired objects and re-compile the program.

- *userTable* is a pointer to an array of UI_ITEM structures. The UI_ITEM structures are used to hold object identifications and either user functions or compare functions. Objects created using Zinc Designer that have associated user functions or compare functions will have an entry in *userTable*. In order to make these connections,

_userTable, created by Zinc Designer, must be assigned to *userTable*. Generally, this is done in the Designer-generated **.CPP** module.

```
UI_ITEM _userTable[] =
{
    { DATE_COMPARE, &DateCompare, NULL, WOF_NO_FLAGS },
    { WINDOW_CLOSE, &CloseFunction, NULL, WOF_NO_FLAGS },
    { OPTION_OK, &OkFunction, NULL, WOF_NO_FLAGS },
    { OPTION_CANCEL, &CancelFunction, NULL, WOF_NO_FLAGS },
    { OPTION_SAVE, &SaveFunction, NULL, WOF_NO_FLAGS },
    { OPTION_CLOSE, &CloseWindow, NULL, WOF_NO_FLAGS },
    { ID_END, NULL, NULL, 0 }
};
UI_ITEM UI_WINDOW_OBJECT::userTable = _userTable;
```

- *eventMapTable* is a pointer to the event map table. This table initially points to the system event map table. This table is used to interpret raw events when entered by the user.

- *hotkeyMapTable* is a pointer to a hotkey map table.

- *paletteMapTable* is a pointer to a palette map table. This table is used to determine the color combinations used to display a window object. This variable initially points to the system palette map table.

- *screenID* is a unique identification given to a window object when it is attached to the Window Manager. This value is used whenever a display function is called. In DOS, this value is generated by Zinc. In Windows, it is the handle returned by a call to **CreateWindow( )**. In Motif, it is defined to be a pointer to the type of Xt widget that the object is.

- *woFlags* are flags (common to all window objects) that determine the general operation of the window object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a window object:

  WOF_AUTO_CLEAR—Automatically clears the edit buffer if the end-user tabs to the window field (from another window field) and presses a non-movement key.

  WOF_BORDER—Draws a single line border around the window object. If the application program is running in text mode, no border is drawn.

  WOF_INVALID—Sets the initial status of the window field to be "invalid." By default, all window information is valid. A programmer may specify a field as invalid by setting this flag upon creation of the window object or by setting the WOS_INVALID status flag during the application's run-time.

WOF_JUSTIFY_CENTER—Center-justifies the window information within the field.

WOF_JUSTIFY_RIGHT—Right-justifies the window information within the field.

WOF_MINICELL—Uses mini-cell values to determine the mini-cell heights. Initially, a mini-cell is set to 1/10 of a cell coordinate. This flag allows for more precise positioning of objects and is valid in graphics modes only.

WOF_NO_ALLOCATE_DATA—Causes the object to not allocate its own buffer for data. If this flag is set, the programmer must allocate a buffer that will be used to store the object's data.

WOF_NO_FLAGS—Does not associate any special flags with the window object. This flag should not be used in conjunction with any other WOF flag.

WOF_NON_FIELD_REGION—Causes the window object to not be a form field. If this flag is set, the window object will occupy all the remaining space of its parent window.

WOF_NON_SELECTABLE—Prevents the window object from being selected. If this flag is set, the user will not be able to position on, or edit, the window object.

WOF_SUPPORT_OBJECT—If an object with this flag set is added to a window, it will be put into the support list. Objects in the support list include: UIW_BORDER, UIW_TITLE, UIW_MAXIMIZE_BUTTON, UIW_MINIMIZE_BUTTON and UIW_SYSTEM_BUTTON. The support list is not searched when list functions are called (e.g., **window->First( )**.) As a result, the first non-support object added to a window (e.g., UIW_BUTTON, UIW_PROMPT, etc.) will be returned when **window->First( )** is called.

WOF_UNANSWERED—Sets the initial status of the window field to be "unanswered." An unanswered window field is displayed as blank space on the screen.

WOF_VIEW_ONLY—Prevents the window object from being edited. If this flag is set, the end-user will not be able to edit a window object's information but will be able to browse through the information.

- *woAdvancedFlags* are flags (common to all window objects) that determine the advanced operation of the window object. The following flags (declared in **UI_-WIN.HPP**) control the advanced operation of a window object:

  WOAF_DIALOG_OBJECT—Creates the window as a dialog box. A dialog box is a temporary window used to display or receive information from the user. Using this flag will cause a dialog style border to be displayed.

  NOTE: Some operating environments (e.g, Windows) will create a border, system button and title for a dialog window. Other environments (e.g., DOS) may not, and so a border, system button and title must be added to the dialog window by the programmer.

  WOAF_LOCKED—Prevents the user from removing the window object from the screen display.

  WOAF_MDI_OBJECT—Causes the window to be an MDI window. If this flag is set on a window that is added to the Window Manager, it becomes an MDI parent (i.e., it can contain MDI child objects). An MDI parent must have a pull-down menu. In general, other than the standard support objects (i.e., system button, border, title, etc.) and the pull-down menu, MDI parent windows should only contain MDI children.

  If this flag is set on a window that is added to another MDI window, it becomes an MDI child window. MDI child windows can be moved or sized but will remain entirely within the MDI parent window.

  NOTE: MDI is not standard across environments. For example, in Windows, child windows will be clipped by their parent window, but in Motif, the child windows will not be clipped by their parent. In Motif, the child windows are still owned by the parent window, however, and so closing the parent window will cause all child windows added to the parent to close also.

  WOAF_MODAL—Prevents any other window from receiving event information from the Window Manager. A modal window receives all event information until it is removed from the screen display.

  WOAF_NO_DESTROY—Prevents the Window Manager from calling the window's destructor. If this flag is set, the window can be removed from the screen display, but the programmer must call the destructor associated with the window object.

WOAF_NO_FLAGS—Does not associate any special advanced flags with the window object. This flag should not be used in conjunction with any other WOAF flags.

WOAF_NO_MOVE—Prevents the end-user from changing the screen location of the window during an application.

WOAF_NON_CURRENT—The window object cannot be made current. If this flag is set, users will not be able to select the object from the keyboard nor with the mouse. If a UIW_BUTTON object has this flag set, it cannot be made current; however, clicking it with the mouse or pressing the hotkey will cause it to depress and have its user function called.

WOAF_NORMAL_HOT_KEYS—Allows an object in the current selection list (e.g., a pop-up item in a pop-up menu) to be selected using the object's hotkey without having to hold down the <Alt> key.

WOAF_NO_SIZE—Prevents the end-user from changing the size of the window during an application.

WOAF_OUTSIDE_REGION—The window object occupies space outside of the *true* region of the window but is still within its parent window (e.g., the UIW_BORDER class).

WOAF_TEMPORARY—Causes the window to only occupy the screen temporarily. Once another window is selected from the screen, the temporary window is removed from the Window Manager (i.e., erased from the display). Once removed, a temporary window will be destroyed if the **WOAF_NO_DESTROY** flag is not set.

- *woStatus* are status flags that specify the current state of a window object. These flags, unlike the window object flags, are set at run-time by the Window Manager and programmer. The following status flags (declared in **UI_WIN.HPP**) specify the window object's current status:

  WOS_CHANGED—The window object's data has been modified by the end-user.

  WOS_CURRENT—The window object is the current object recognized by the Window Manager or by its parent if the object is a sub-object. If this status flag is set, the associated window object will get all user input. Only one window object may have the WOS_CURRENT flag set at any given time.

WOS_DESTROY_ICON—If the object is a run-time icon, this advanced flag is used to tell how to delete the icon. This flag is only used in Windows. It should not be used by programmers.

WOS_EDIT_MODE—The object is currently being edited. This flag is used in Windows only.

WOS_GRAPHICS—Indicates that the window object regions are specified in graphics coordinates. This flag is used by the Window Manager to determine whether a window object's region coordinates need to be converted to either character cell positions or graphic pixel coordinates.

WOS_INVALID—The window object's data is in an "invalid" state.

WOS_MAXIMIZED—The window object is in a maximized state.

WOS_MINIMIZED—The window object is in a minimized state.

WOS_NO_STATUS—No status flags are associated with the window object at the current time.

WOS_OWNERDRAW—Causes the window object's **DrawItem( )** function to be called when the object needs to be drawn.

WOS_READ_ERROR—Indicates that there was an error loading the persistent object from the **.DAT** file.

WOS_REDISPLAY—The window object needs to be redisplayed.

WOS_SELECTED—Indicates that the object has been selected. The most common use for this flag is with buttons, where a button field can be in a selected or a non-selected state.

WOS_UNANSWERED—The window object's data is in an "unanswered" state.

WOS_WINDOWS_ACTION—This flag allows an object to identify when it receives an event that was sent from Windows.

- *true* is the true coordinate of the object relative to the screen's origin. In graphics mode, this is the pixel location of the object on the screen. In text mode, it is the character location of the object.

- *relative* is the coordinate passed to the object in its constructor. It contains the desired position of the window object either on the screen (for windows) or the desired position of the object within its parent window.

- *parent* is a pointer to the window object's parent. For example, if a high level window were created and contained several child objects, the parent object for the children would be the window.

- *helpContext* is the help context identifier associated with a window object. It contains a value that indicates to the help system the nature of help to display for the object. If there is no help available for the currently defined window object, it will be passed to the parent object and the help information associated with the parent object, if available, will be displayed. If there is no help available at the parent object level, the message will be relayed on to the system, and general system help will be displayed.

- *userFlags* are flags that are set and maintained by the programmer. These flags are not used by Zinc Application Framework. *userFlags* is saved when the object is stored in a data file.

- *userStatus* are status flags that are set and maintained by the programmer. These flags are not used by Zinc Application Framework. As with other status flags, *userStatus* is not saved when the object is stored in a data file.

- *userObject* is a pointer to a user defined object. Since this is a void pointer, the object must be cast by the programmer. This pointer is not used by Zinc Application Framework. However, if *userObject* has a corresponding entry in the *UI_WINDOW_-OBJECT::userTable*, then the text name is saved in the data file.

- *userFunction* is a pointer to a user function. This function returns an EVENT_TYPE and takes the following arguments:

  object is a UI_WINDOW_OBJECT pointer to the object that called this function. If specific class information is needed, this argument must be cast by the programmer.

  event is a UI_EVENT reference pointer to the event that caused this function to be called.

  ccode is an EVENT_TYPE value containing the logical interpretation of the event that caused this function to be called.

- *searchID* is an identification used while indexing an object in a UI_STORAGE file.

- *numberID* is a numerical value used to identify an object. This value is either set by the programmer or set when the object is created using Zinc Designer.

- *stringID* is a string name used to identify an object. This string is either set by the programmer or set when the object is created using Zinc Designer. It contains a string that has a maximum length of 32 characters.

- *windowID* has five identification values, from 0 to 4, that form the object's hierarchy. When an object is initially created, all five values are assigned the ID_WINDOW_-OBJECT identifier. When the constructor is called for each particular item, however, it establishes the object's actual hierarchy by replacing ID_WINDOW_OBJECT with the identifiers of the child objects. This hierarchy is used at run-time by functions such as **MapEvent( )** and **MapPalette( )**, neither of which have any knowledge of class hierarchies. For example, for a UIW_POP_UP_ITEM, windowID[0] would be ID_POP_UP_ITEM, windowID[1] would be ID_BUTTON, and the remaining entries would be ID_WINDOW_OBJECT.

- *hotKey* indicates which hot key to associate with the specified object. A value of 0 means that no hot key is associated with the object.

- *font* is the LOGICAL_FONT associated with the object. For more information regarding fonts, see the chapter for the individual screen displays.

- *lastPalette* is a pointer to the last palette used to display the object.

- *userObjectName* points to the string representation of the user object name. This variable is used for storage purposes only.

- *userFunctionName* points to the string representation of the user function name. This variable is used for storage purposes only.

- *clip* provides additional clip region information for an object. The area represented by *clip* is known as the client area of the parent window (i.e., the window area inside of the borders and the title bar).

- *dwStyle* is the default window style when running under Windows. This member is only available under Windows.

- *defaultCallback* is the base class default callback function when running under Windows or OS/2 (e.g., **DefWindowProc( )** in Windows). This member is available under Windows and OS/2 only.

- *args* is an array of Xt resources. This member is available for Motif only.

- *nargs* is a counter of how many entries have been made in the *args* array. This member is available for Motif only.

- *shell* is a pointer to the object's shell widget. It must be set before a call to **UI_WINDOW_OBJECT::RegisterObject( )** is made. This member is available for Motif only.

Other chapters in this manual contain more information about the classes derived from the UI_WINDOW_OBJECT base class as well as their construction, destruction and use within Zinc Application Framework.

NOTE: All the member functions in this chapter are advanced. In general, only derived window objects should need access to these functions.

UI_WINDOW_OBJECT::UI_WINDOW_OBJECT

Syntax

#include <ui_win.hpp>

UI_WINDOW_OBJECT(int *left*, int *top*, int *width*, int *height*,
WOF_FLAGS *woFlags*, WOAF_FLAGS *woAdvancedFlags*);
or
UI_WINDOW_OBJECT(const char **name*, UI_STORAGE **file*,
UI_STORAGE_OBJECT **object*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These advanced overloaded constructors initialize the base UI_WINDOW_OBJECT class.

The first constructor is used for the general initialization of window objects. This constructor is protected so that only derived classes can call it directly. It requires the following parameters:

- $left_{in}$ and top_{in} are the starting cell position of the object. If either of these values is a negative number, the coordinate is tied to the right or bottom of the screen. For example, a window with values *left* = -10 and *top* = -5 will cause the window to be displayed 10 cell positions from the right side of the screen and 5 lines from the bottom of the screen. Negative position coordinates can only be used for window objects attached directly to the screen, such as a window.

- $width_{in}$ and $height_{in}$ are the size (in cells) of the window object. If negative values are used, these values represent the bottom-right position of the window object on the screen. For example, a window with values *width* = -5 and *height* = -1 will cause the window's right position to be 5 cell positions from the right side of the screen and 1 line from the bottom of the screen. Negative size coordinates can only be used for window objects attached directly to the screen, such as a window.

- *woFlags* are flags (common to all window objects) that determine the general operation of the window. A full description of these flags is given at the beginning of this chapter.

- *woAdvancedFlags* are flags that determine the advanced operation of the window object. A full description of these flags is given at the beginning of this chapter.

The second constructor is used to read an object from disk.

- $name_{in}$ is the name of the object to be loaded.

- $file_{in}$ is a pointer to the UI_STORAGE that contains the persistent object. (For more information on storage files, see "Chapter 35—UI_STORAGE.")

- $object_{in}$ is a pointer to the UI_STORAGE_OBJECT where the persistent object information will be loaded. This must be allocated by the programmer. (For more information on loading persistent objects, see "Chapter 36—UI_STORAGE_-OBJECT.")

Example

```
#include <ui_win.hpp>

UIW_BUTTON::UIW_BUTTON(int left, int top, int width, char *_string,
    USHORT _btFlags, USHORT _woFlags,
    void (*_userFunction)(void *object, UI_EVENT &event), USHORT _value) :
    UI_WINDOW_OBJECT(left, top, width, 1, _woFlags | WOF_BORDER,
    WOAF_NO_FLAGS),
    btFlags(_btFlags), btStatus(BTS_NO_STATUS), userFunction(_userFunction),
    string(NULL), value(_value), getString(NULL), time(0)
{
    .
    .
    .

}
```

UI_WINDOW_OBJECT::~UI_WINDOW_OBJECT

Syntax

#include <ui_win.hpp>

virtual ~UI_WINDOW_OBJECT(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the information associated with the base UI_-WINDOW_OBJECT class. This function is declared virtual so that the destructors associated with derived classes will be called before the base class destructor is called.

UI_WINDOW_OBJECT::DrawBorder

Syntax

#include <ui_win.hpp>

EVENT_TYPE DrawBorder(SCREENID *screenID*, UI_REGION &*region*, int *fillRegion*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function draws the single-line border of a window object.

- $returnValue_{out}$ is TRUE if the border is drawn. Otherwise, *returnValue* is FALSE.
- $screenID_{in}$ is the identification associated with the object. In Windows, *screenID* is the window handle.
- $region_{in/out}$ is the region around which the border is drawn. This value is decremented if the **WOF_BORDER** flag is set and the border is drawn. (This argument should be a copy of member variable *true*.)
- $fillRegion_{in}$ tells whether the region within the border should be filled.
- $ccode_{in}$ is the type of event that required the border to be drawn.

Example

```
#include <ui_win.hpp>

EVENT_TYPE UIW_WINDOW::Event(const UI_EVENT &event)
{
    .
    .
    .

    // Switch on the event type.
    switch (ccode)
    {
```

```
        case S_CURRENT:
        case S_NON_CURRENT:
        case S_DISPLAY_ACTIVE:
        case S_DISPLAY_INACTIVE:
            {
            // Draw the border and fill the background.
            UI_WINDOW_OBJECT::Event(event);
            if (!FlagSet(woStatus, WOS_REDISPLAY))
                break;
            UI_REGION region = true;
            display->VirtualGet(screenID, region);
            if (FlagSet(woFlags, WOF_BORDER) && true.Overlap(event.region))
                DrawBorder(screenID, region, FALSE, ccode);
        .
        .
        .

        }

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

UI_WINDOW_OBJECT::DrawItem

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE DrawItem(const UI_EVENT &*event*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual advanced routine is used to draw an object on the screen. This function is called only if the object's woStatus has the WOS_OWNERDRAW status set.

- $returnValue_{out}$ is a response based on the success of the function call. If successful, the function returns a non-zero value. If the object was not drawn, 0 is returned.

- $event_{in}$ contains a run-time message for the specified object. The object is drawn according to the type of event. The following logical events are handled by the **DrawItem( )** routine:

S_CURRENT, S_NON_CURRENT, S_DISPLAY_ACTIVE and **S_DISPLAY_INACTIVE**—These messages cause the object to be redisplayed. If S_CURRENT or S_NON_CURRENT are passed, the object will always be updated. If S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE are passed the object will only be updated if *event.region* overlaps the object region.

WM_DRAWITEM—A message that causes the object to be redrawn. This message is specific to Windows and OS/2.

Expose—A message that causes the object to be redrawn. This message is specific to Motif.

- $ccode_{in}$ contains the logical interpretation of *event*.

UI_WINDOW_OBJECT::DrawShadow

Syntax

#include <ui_win.hpp>

EVENT_TYPE DrawShadow(SCREENID *screenID*, UI_REGION &*region*, int *depth*, int *fillRegion*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function draws the three-dimensional appearance of a window object.

- $returnValue_{out}$ is TRUE if the shadow is drawn. Otherwise, *returnValue* is FALSE.
- $screenID_{in}$ is the identification associated with the object. In Windows, *screenID* is the window handle.
- $region_{in/out}$ is the region to be shadowed. *region* is decremented according to the depth. This argument should be a copy of member variable *true*.

- *depth*$_{in}$ is the degree of shading. Value greater than 0 (i.e., 1, 2) causes the object to have a positive shadow, a value of 0 causes no shadow to be drawn, and values less than 0 (i.e., -1, -2) cause the object to appear depressed.

- *fillRegion*$_{in}$ tells whether to fill the region inside of the shadow.

- *ccode*$_{in}$ is the type of event that required the shadow to be drawn.

Example

```
#include <ui_win.hpp>

EVENT_TYPE UIW_BORDER::DrawItem(const UI_EVENT &, EVENT_TYPE ccode)
{
    // Check for text mode.
    if (display->isText)
    {
        UI_REGION region = parent->true;
        DrawShadow(screenID, region, 2, FALSE, ccode);
        return (ccode);
    }
    .
    .
    .

}
```

UI_WINDOW_OBJECT::DrawText

Syntax

#include <ui_win.hpp>

EVENT_TYPE DrawText(SCREENID *screenID*, UI_REGION &*region*, const char **text*, UI_PALETTE **palette*, int *fillRegion*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function determines the presentation of the object's text.

- *returnValue*$_{out}$ is TRUE if the text is drawn. Otherwise, *returnValue* is FALSE.
- *screenID*$_{in}$ is the identification associated with the object. In Windows, *screenID* is the window handle.
- *region*$_{in/out}$ is the region that the text will occupy on the screen. If the object has the **WOF_BORDER** flag set, *region* is decremented. It should be a copy of member variable *true*.
- *text*$_{in}$ is a pointer to the text to be displayed. If the text string contains a hotkey character, denoted by a preceding '&' character, then it will be underlined if the application is running in graphics mode or highlighted if the application is running in text mode.
- *palette*$_{in}$ is the color palette to be used to draw the text.
- *fillRegion*$_{in}$ tells whether to fill the background part of the region.
- *ccode*$_{in}$ is the type of event that required the text to be drawn.

UI_WINDOW_OBJECT::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used as the base level handler for all events not processed by a derived window object.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the window object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified window object. The type of operation depends on the interpreted value for the event.

  L_BEGIN_SELECT—Begins the selection process of a window or window object. (This message is normally interpreted from a raw event generated by the mouse.) For example, if the end-user presses the left mouse button, the selection of the object is initiated. When the mouse button is released (L_END_SELECT), the selection will be completed.

  L_CONTINUE_SELECT—Is a drag operation interpreted from a raw event generated by the mouse driver, indicating that L_BEGIN_SELECT was already sent and the mouse is presently being dragged as part of the selection process.

  L_END_SELECT—Indicates that the selection process, initiated with the L_BEGIN_SELECT message, is complete. For example, the end-user has pressed and released the left mouse button.

  L_SELECT—Causes the object's user function to be called, if one exists.

  L_VIEW—Is an interpreted event which indicates that no mouse buttons are currently pressed but the mouse is being moved across the screen.

  S_ADD_OBJECT—Is used to add an object (contained in *event.data*) to the receiving object. This message is interpreted only by those objects that contain a list (e.g., windows, horizontal and vertical lists, combo boxes, etc.).

  S_CREATE and **S_SIZE**—Initializes *screenID* and causes the region coordinates to be converted (i.e., text to graphics or vise-versa) and the screen position (i.e., *true*) to be determined. This position is determined by the *relative* position of the field, the status of *woStatus* flags and the amount of available space within the parent window.

  S_CURRENT—Tells the receiving window object that it is the current (or highlighted) window object on the screen. The S_CURRENT message uses the UI_REGION portion of the UI_EVENT structure to indicate the coordinates of any region that overlaps a part of the previously current window object. The specified region will need to be refreshed, or repainted, to the foreground of the display. If no region is overlapping, the object will simply be shown in a current

mode, without any repainting. Once the S_CURRENT message is sent, the window object will receive all relevant event information passed through the Window Manager, since event messages are most often intended for the current object.

S_DISPLAY_ACTIVE—Is sent through the parent window to a window object, telling the object to re-display itself according to an active state. In other words, one of the other objects within the parent window has become current, so the rest of the objects must be re-displayed according to an active state. This message is passed with the affected region (contained in the UI_REGION portion of the UI_EVENT structure). The object only needs to re-display its screen information when the region passed by the event overlaps the region of the object.

S_DISPLAY_INACTIVE—Is sent through a parent window to a window object, telling the object to re-display itself according to an inactive state. In other words, the parent window is no longer active (meaning that none of its objects are current), and all objects within it must be re-displayed as inactive. This message is passed with the affected region (contained in the UI_REGION portion of the UI_EVENT structure). The object only needs to re-display its screen information when the region passed by the event overlaps the region of the object.

S_INITIALIZE—Sets up the default *stringID*, *numberID* and position information.

S_MOVE—Changes the *true* coordinates of the window object and determines whether the object still fits within its parent's defined region. The amount of movement is determined by *event.position*.

S_NON_CURRENT—Tells the receiving window object that it is no longer the current window object.

S_REDISPLAY—This message causes the window object to be redisplayed.

S_REGION_DEFINE—Is sent to a window object when it has the WOAF_-MULTIPLE_REGIONS flag set. This allows the window object to define its own sub-regions on the screen (using the **UI_DISPLAY::RegionDefine( )** member function).

S_SUBTRACT_OBJECT—Is used to delete an object (pointed to by *event.data*) from the receiving object. This message is interpreted only by those objects that contain a list (e.g., windows, horizontal and vertical lists, combo boxes, etc.).

All other messages cause the S_UNKNOWN message to be returned.

Example

```
#include <ui_win.hpp>

EVENT_TYPE UIW_BUTTON::Event(const UI_EVENT &event)
{
    EVENT_TYPE ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_BUTTON);
    switch (ccode)
    {
    case S_CREATE:
    case S_SIZE:
        .
        .
        .

    default:
        ccode = UI_WINDOW_OBJECT::Event(event);
        break;
    }

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

UI_WINDOW_OBJECT::Get

Syntax

#include <ui_win.hpp>

UI_WINDOW_OBJECT *Get(const char **name*);
or
UI_WINDOW_OBJECT *Get(NUMBERID *_numberID*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded function searches the object's list (if the object is inherited from UI_LIST—otherwise nothing is done) for an object whose identification matches the identification passed in.

The first overloaded function searches the object's list for an object whose *stringID* matches *name*.

- *returnValue*$_{out}$ is a pointer to the object in the list whose *stringID* matches *name*. If no match is found, then *returnValue* is NULL.

- *name*$_{in}$ is a pointer to a constant character string that contains the *stringID* of the object to be found.

The second overloaded function searches the object's list for an object whose *numberID* matches *_numberID*.

- *returnValue*$_{out}$ is a pointer to the object in the list whose *numberID* matches *_numberID*. If no match is found, then *returnValue* is NULL.

- *_numberID*$_{in}$ is the *numberID* of the object to be found.

UI_WINDOW_OBJECT::HotKey

Syntax

```
#include <ui_win.hpp>

unsigned HotKey(unsigned hotKey = 0);
    or
unsigned HotKey(char *text);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This overloaded function sets the hotkey for an object. If an object added to a window contains sub-objects with hotkeys, then the object should have its hotkey set to HOT_KEY_SUB_WINDOW so that its children can process hotkeys. For example, a UIW_TOOL_BAR with buttons on it should have its hotkey set to HOT_KEY_SUB_-WINDOW.

The first overloaded function sets the hotkey for the object.

- *returnValue*$_{out}$ is the value of the hotkey after it has been changed.
- *hotKey*$_{in}$ is the new value of the hotkey. Any alphanumeric character can be used for a hotkey. If 0 is entered for *hotKey*, no change is made and its value is returned.

The second overloaded function sets the hotkey for the object by parsing the text that is passed in looking for the hotkey designator character (an '&' by default).

- *returnValue*$_{out}$ is the value of the hotkey after it has been changed.
- *text*$_{in}$ is a pointer to the text for the object. This text is searched for the hotkey designator character (an '&' by default). If the character is found, then the character immediately after it is set to be the object's hotkey.

Example

```
ExampleFunction(UI_WINDOW_OBJECT *object1, UI_WINDOW_OBJECT *object2)
{
    object1->HotKey('A');
    .
    .
    .

    unsigned value = object1->HotKey( );
    object2->HotKey(value);
}
```

UI_WINDOW_OBJECT::Information

Syntax

#include <ui_win.hpp>

virtual void *Information(INFORMATION_REQUEST *request*, void **data*,
OBJECTID *objectID* = 0);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. The following requests (defined in **UI_WIN.HPP**) are recognized by UI_WINDOW_-OBJECT:

  GET_NUMBERID_OBJECT—Returns a pointer to the first object whose *numberID* matches the value in *data*. If this message is sent, *data* must be a pointer to a programmer defined unsigned short.

  GET_STRINGID_OBJECT—Returns a pointer to the first object whose *stringID* matches the value in *data*. If this message is sent, *data* must be a pointer to a programmer defined string (i.e., char *).

  GET_FLAGS—Returns the WOF_FLAGS associated with an object if the value in *objectID* is ID_WINDOW_OBJECT. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  SET_FLAGS—Sets the WOF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  CLEAR_FLAGS—Clears the WOF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  GET_STATUS—Returns the WOS_STATUS associated with an object if the value in *objectID* is ID_WINDOW_OBJECT. If this message is sent, *data* must be a pointer to a programmer defined UIS_STATUS.

  SET_STATUS—Sets the WOS_STATUS, specified by data, associated with an object if the value in *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIS_STATUS.

CLEAR_STATUS—Clears the WOS_STATUS, specified by *data*, associated with an object if the value in *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIS_STATUS.

CHANGED_FLAGS and **CHANGED_STATUS**—Cause the object to be re-initialized. If this flag is used, *data* is ignored and should be NULL.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

- $objectID_{in}$ is the identification code of the object to receive the request.

Example

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

void *UIW_BUTTON::Information(INFO_REQUEST request, void *data,
    OBJECTID objectID)
{
    // Switch on the request.
    switch (request)
    {
    .
    .
    .

    case GET_FLAGS:
    case SET_FLAGS:
    case CLEAR_FLAGS:
        if (objectID && objectID != ID_BUTTON)
            data = UI_WINDOW_OBJECT::Information(request, data, objectID);
        else if (request == GET_FLAGS && !data)
            data = &btFlags;
        else if (request == GET_FLAGS)
            *(BTF_FLAGS *)data = btFlags;
        else if (request == SET_FLAGS)
            btFlags |= *(BTF_FLAGS *)data;
        else
            btFlags &= ~(*(BTF_FLAGS *)data);
        break;
    .
    .
    .

    }

    // Return the information.
    return (data);
}
```

UI_WINDOW_OBJECT::Inherited

Syntax

#include <ui_win.hpp>

int Inherited(OBJECTID *matchID*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function indicates whether the current object is inherited from a particular class, given the class identification value.

- $returnValue_{out}$ is TRUE if the window object is inherited from the class given by the *matchID* class identifier. Otherwise, this value will be FALSE.

- $matchID_{in}$ is the class identification value to match. For example, the type identification ID_BUTTON matches with an object derived from the button class. If **Inherited( )** is able to match this value, it passes back TRUE.

Example

```
#include <ui_win.hpp>

EVENT_TYPE UIW_ICON::Event(const UI_EVENT &event)
{
    .
    .
    .

    // Switch on the event type.
    EVENT_TYPE ccode = event.type;
    switch (ccode)
    {
    case S_CREATE:
        if (!_iconJumpInstance)
            _iconJumpInstance = (FARPROC)IconJumpProcedure;
        UI_WINDOW_OBJECT::Event(event);
        if (parent->Inherited(ID_LIST))
            parent->woStatus |= WOS_OWNERDRAW;
```

```
            .
            .
            .
    }

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

UI_WINDOW_OBJECT::Load

Syntax

#include <ui_win.hpp>

virtual void Load(const char *_name_, UI_STORAGE *_file_,
UI_STORAGE_OBJECT *_object_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function is used to load a UI_WINDOW_OBJECT from a persistent object file.

- $name_{in}$ is the name of the object to be loaded.
- $file_{in}$ is a pointer to a UI_STORAGE that contains the persistent object. (For more information on persistent object files, see "Chapter 35—UI_STORAGE.")
- $object_{in}$ is a pointer to the UI_STORAGE_OBJECT where the persistent object information will be loaded. This must be allocated by the programmer. (For more information on loading persistent objects, see "Chapter 36—UI_STORAGE_OBJECT.")

NOTE: This function must only be called by derived objects when the derived object's **Load( )** function is called.

Example

```
void UIW_WINDOW::Load(const char *name, UI_STORAGE *directory,
    UI_STORAGE_OBJECT *file)
{
    // Check for a valid directory and file.
    int tempDirectory = FALSE, tempFile = FALSE;
    .
    .
    .

    // Load the window information.
    UI_WINDOW_OBJECT::Load(name, directory, file);

    // Load the object information.
    short noOfObjects;
    file->Load(&noOfObjects);
    for (int i = 0; i < noOfObjects; i++)
        Add(UI_WINDOW_OBJECT::New(NULL, directory, file));
    file->Load(&wnFlags);
    file->Load(&compareFunctionName);

    // Clean up the file and storage.
    if (tempFile)
        delete file;
    if (tempDirectory)
        delete directory;
}
```

UI_WINDOW_OBJECT::LogicalEvent

Syntax

#include <ui_win.hpp>

EVENT_TYPE LogicalEvent(const UI_EVENT *&event*, OBJECTID *currentID* = 0);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function is used by all window objects to interpret a raw event.

- *returnValue*$_{out}$ is the logical event that is interpreted based on the type of event and object identification passed to the function.

- $event_{in}$ is a reference pointer to the raw event.
- $currentID_{in}$ is the identification for the object receiving the event. This value is used to determine the mapping of a logical event.

Example

```
#include <ui_win.hpp>

EVENT_TYPE UIW_BORDER::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    UI_REGION region;
    EVENT_TYPE ccode = LogicalEvent(event, ID_BORDER);
    switch (ccode)
    {
    .
    .
    .

    }

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

UI_WINDOW_OBJECT::LogicalPalette

Syntax

#include <ui_win.hpp>

UI_PALETTE *LogicalPalette(LOGICAL_EVENT *logicalEvent*,
    OBJECTID *currentID* = 0);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function is used by all window objects to determine the palette associated with a logical event.

- *returnValue*$_{out}$ is a pointer to the palette that should be used to display object information to the screen.
- *logicalPalette*$_{in}$ is which logical palette entry to use.
- *currentID*$_{in}$ is the identification for the object receiving the event. This value is used to determine the palette mapping given the logical event.

Example

```
#include <ui_win.hpp>

EVENT_TYPE EXAMPLE_CLASS::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    UI_REGION region;
    EVENT_TYPE ccode = UI_WINDOW_OBJECT::LogicalEvent(event, ID_WINDOW_OBJECT);
    switch (ccode)
    {
    case S_DISPLAY_INACTIVE:
    case S_DISPLAY_ACTIVE:
        // Draw the borders around the object.
        UI_WINDOW_OBJECT::Event(event);
        UI_PALETTE *palette = LogicalPalette(ccode, ID_WINDOW_OBJECT);
        .
        .
        .

        break;
    .
    .
    .

    }

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

UI_WINDOW_OBJECT::Modify

Syntax

```
#include <ui_win.hpp>

void Modify(const UI_EVENT &event);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function is used to change an object's size or position. When this function is called, an XOR outline of the object appears. The outline can be moved or sized with the arrow keys on the keyboard or by moving the mouse (if **Modify( )** was invoked as a response to a mouse click). When <Enter> is pressed or the mouse button is released, the object will take on its new size or position.

- *event*$_{in}$ contains the type of modification to be done. *event*'s members are set to the following values:

  event.type is set to either S_MOVE or S_SIZE.

  event.rawCode contains the event types that will be checked when modifying the object (i.e., M_LEFT_CHANGE, M_TOP_CHANGE, M_RIGHT_CHANGE or M_BOTTOM_CHANGE.)

  event.position contains the object's starting position.

Example

```
#include <ui_win.hpp>

EVENT_TYPE UIW_WINDOW::Event(const UI_EVENT &event)
{
    UI_WINDOW_OBJECT *object;

    .
    .
    .

    // Switch on the event type.
    switch (ccode)
    {
    .
    .
    .
```

```
        case S_MOVE:
        case S_SIZE:
            Modify(event);
            break;
        }
}
```

UI_WINDOW_OBJECT::NeedsUpdate

Syntax

#include <ui_win.hpp>

int NeedsUpdate(const UI_EVENT &*event*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function passes back TRUE if the field or object needs to be refreshed on the screen and FALSE if it does not.

- $returnValue_{out}$ is TRUE if the object needs to be refreshed. Otherwise, the object does not need to be refreshed, based on the specified event and current state of the object.
- $event_{in}$ is the raw event used to request an update.
- $ccode_{in}$ is the logical code that caused **NeedsUpdate( )** to be called.

Example

```
#include <ui_win.hpp>

EVENT_TYPE UIW_PROMPT::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    EVENT_TYPE ccode = event.type;
    switch (ccode)
    {
    case S_CURRENT:
```

```
        case S_NON_CURRENT:
        case S_DISPLAY_ACTIVE:
        case S_DISPLAY_INACTIVE:
            if (UI_WINDOW_OBJECT::NeedsUpdate(event, ccode))
                .
                .
                .

            break;
        .
        .
        .

        }

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

UI_WINDOW_OBJECT::New

Syntax

#include <ui_win.hpp>

static UI_WINDOW_OBJECT *New(const char **name*, UI_STORAGE **file* = NULL, UI_STORAGE_OBJECT **object* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to load a persistent object from disk. When persistent objects are used, it may be desirable to provide a way of jumping into an object's constructor from a function pointer in a table. For example, Zinc Designer creates an *objectTable* for all objects that are present in the **.DAT** file that it creates. This object table is used to allow a Zinc program to look up an entry based on its identification code and then to call the appropriate constructor from the address contained in the *newFunction* field. In C++, a function must be static in order to take its address, but constructors cannot be static. Therefore, the **New( )** function is used to call the load constructor for the appropriate class. For more information regarding the load constructor see the constructor section of this chapter.

NOTE: The application must first create a display if objects are to be loaded from a **.DAT** file.

- *name*$_{in}$ is the name of the object to be loaded.
- *file*$_{in}$ is a pointer to a UI_STORAGE that contains the persistent object. (For more information on persistent object files, see "Chapter 35—UI_STORAGE.")
- *object*$_{in}$ is a pointer to the UI_STORAGE_OBJECT where the persistent object information will be loaded. This must be allocated by the programmer. (For more information on loading persistent objects, see "Chapter 36—UI_STORAGE_-OBJECT.")

Example

```
#include <ui_win.hpp>
UI_ITEM _objectTable[] =
{
    { ID_BORDER, &UIW_BORDER::New, NULL, WOF_NO_FLAGS },
    { ID_BUTTON, &UIW_BUTTON::New, "BUTTON", WOF_NO_FLAGS },
    { ID_GROUP, &UIW_GROUP::New, "GROUP", WOF_NO_FLAGS },
    { ID_ICON, &UIW_ICON::New, "ICON", WOF_NO_FLAGS },
    { ID_HLIST, &UIW_HZ_LIST::New, "HLIST", WOF_NO_FLAGS },
    { ID_MAXIMIZE_BUTTON, &UIW_MAXIMIZE_BUTTON::New, NULL, WOF_NO_FLAGS },
    { ID_MINIMIZE_BUTTON, &UIW_MINIMIZE_BUTTON::New, NULL, WOF_NO_FLAGS },
    { ID_PROMPT, &UIW_PROMPT::New, "PROMPT", WOF_NO_FLAGS },
    { ID_SCROLL_BAR, &UIW_SCROLL_BAR::New, "SCROLL_BAR", WOF_NO_FLAGS },
    { ID_STRING, &UIW_STRING::New, "STRING", WOF_NO_FLAGS },
    { ID_SYSTEM_BUTTON, &UIW_SYSTEM_BUTTON::New, NULL, WOF_NO_FLAGS },
    { ID_TITLE, &UIW_TITLE::New, NULL, WOF_NO_FLAGS },
    { ID_WINDOW, &UIW_WINDOW::New, "WINDOW", WOF_NO_FLAGS },
    { ID_END, NULL, NULL, 0 }
};

UIW_WINDOW *LoadWindow(char *windowName)
{
    UIW_WINDOW *dataWindow = NULL;
    UI_STORAGE storage("test.dat", UIS_READ);
    UI_STORAGE_OBJECT sObject(storage, windowName, ID_WINDOW, UIS_READ);
    if (!sObject.objectError)
        dataWindow = UIW_WINDOW::New(windowName, &storage, &sObject);
    return dataWindow;
}
```

UI_WINDOW_OBJECT::NumberID

Syntax

#include <ui_win.hpp>

NUMBERID NumberID(NUMBERID *numberID* = 0);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function sets or retrieves the numeric identification associated with a window object.

NOTE: The variable *numberID* is used by the library. This variable is available for advanced programming and primarily for derived objects. If this variable is changed, the programmer must ensure that the new value is unique within the scope of the parent window. Also, immediately after changing *numberID*, the parent window's *numberID* must be set to a value greater than that of any of the child objects. Because of this requirement, a window's numberID may change and thus should not be used to identify a window. It is recommended that programmers use *stringID*, rather than *numberID*, to identify window objects.

- $returnValue_{out}$ is the current numeric identification for the specified object.
- $numberID_{in}$ is the numeric value associated with the object. If this value is 0, the number identification is not redefined, but the current number identification is still passed back.

UI_WINDOW_OBJECT::RegionConvert

Syntax

#include <ui_dsp.hpp>

void RegionConvert(UI_REGION &*region*, int *absolute*);

Portability

This function is available on the following environments:

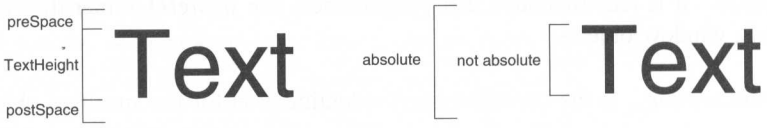
■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function converts a cell (or text) region or a minicell region to a graphics (or pixel) region. The purpose of this function is to allow portability of higher-level display code. This function checks for regions that have already been converted from text to graphics coordinates.

- *region*$_{in/out}$ is a reference pointer to the region whose cell coordinates are to be converted.

- *absolute*$_{in}$ is TRUE if the object's absolute screen coordinates are requested. *absolute* is FALSE if non-absolute coordinates are requested. With text objects, a value of TRUE will compute the absolute region (i.e., the space taken up by **TextHeight( )**, *preSpace* and *postSpace*.) Similarly, with text objects, a value of FALSE will compute the non-absolute region (i.e., the height of the text alone). The following figure shows the relationship between absolute and non-absolute conversions:



Example

```
#include <ui_win.hpp>

EVENT_TYPE UI_WINDOW_OBJECT::Event(const UI_EVENT &event)
{
    static UI_TIME lastTime;
    UI_WINDOW_OBJECT *object;

    // Switch on the event type.
    EVENT_TYPE ccode = LogicalEvent(event);
    switch (ccode)
    {
    case S_INITIALIZE:
        .
        .
        .

        dwStyle |= parent ? WS_CHILD | WS_VISIBLE : WS_OVERLAPPED;
        RegionConvert(relative, (parent && !FlagSet(woFlags,
            WOF_NON_FIELD_REGION)) ? FALSE : TRUE);
        break;
    .
    .
    .

    }
```

```
    .
    .
    .
}
```

UI_WINDOW_OBJECT::RegionMax

Syntax

#include <ui_win.hpp>

virtual void RegionMax(UI_WINDOW_OBJECT **object*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function calculates how much space a field can occupy within its parent object. Objects within the parent object that have the WOF_NON_FIELD_REGION flag set, such as the title and system button of a window, have their regions reserved and are therefore not included in the total available maximum region. The regions of any other objects, however, are included in the total available region, since these objects can overlap with others.

- $object_{in/out}$ is a pointer to the object that is requesting the maximum region of its parent.

Example

```
#include <ui_win.hpp>

#if defined(ZIL_MSWINDOWS)
EVENT_TYPE UIW_PULL_DOWN_MENU::Event(const UI_EVENT &event)
{
    UI_WINDOW_OBJECT *object;

    // Switch on the event type.
    EVENT_TYPE ccode = event.type;
    switch (ccode)
    {
    .
```

```
    .
    .

    case S_SIZE:
        // Compute the positions of the window objects.
        if (FlagSet(pdStatus, PDS_MAIN_MENU))
        {
            true.top = true.bottom = 0;
            break;
        }
        else if (ccode == S_SIZE)
            parent->RegionMax(this);
        .
        .
        .

        break;
    .
    .
    .

    }
    return (ccode);
}
```

UI_WINDOW_OBJECT::RegisterObject

Syntax

#include <ui_win.hpp>

void RegisterObject(char **name*, char **baseName*, int **offset*, FARPROC **procInstance*, FARPROC **defProcInstance*, char **title*, HMENU *menu* = 0);
or
void RegisterObject(char **name*, char **baseName*, WNDPROC **defProcInstance*, char **title*, HMENU *menu* = 0);
or
void RegisterObject(char **name*, char **baseName*, PFNWP **baseCallback*, char **title*);
or
void RegisterObject(WidgetClass *widgetClass*, MOTIF_CONVENIENCE_FUNCTION *convenienceFunction*, EVENT_TYPE *ccode*, int *useArgs* = FALSE, int *manage* = TRUE, SCREENID *_parent* = NULL);

Portability

This function is available on the following environments:

☐ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function registers the Windows, OS/2 or Motif class object if it has not already been initialized.

The first overloaded function is specific to Windows.

- *name*$_{in}$ is the name of the object being registered. These are Zinc names such as UIW_VT_LIST, UIW_BUTTON, UIW_WINDOW, etc.

- *baseName*$_{in}$ is the name of the class from which the object being registered is derived. *baseName* is the base class of the Windows object and does not necessarily follow Zinc's class hierarchy. For example, UIW_VT_LIST is derived from UIW_-WINDOW, but in the Windows environment, the Zinc object is derived from LIST_BOX and must be registered that way.

- *offset*$_{in/out}$ is the size (in bytes) of the extra area following the window instance, including the size of a UI_WINDOW_OBJECT structure. Initially, this value should be -1 (representing an unregistered object).

- *procInstance*$_{in/out}$ is the address of the callback function that Windows will call when the object gets an event. This function is provided, for each object, by Zinc Application Framework.

- *defProcInstance*$_{in/out}$ is the address of the default callback function that Windows provides for each object. This function is provided, for each object, by Windows.

- *title*$_{in}$ is a string containing the title of the window, if any.

- *menu*$_{in}$ is the pull-down menu associated with the window, if any.

The second overloaded function is specific to Windows NT.

- *name*$_{in}$ is the name of the object being registered. These are Zinc names such as UIW_VT_LIST, UIW_BUTTON, UIW_WINDOW, etc.

- *baseName*$_{in}$ is the name of the class from which the object being registered is derived. *baseName* is the base class of the Windows NT object and does not necessarily follow Zinc's class hierarchy.

- *defProcInstance*$_{in/out}$ is the address of the default callback function that Windows provides for each object. This function is provided, for each object, by Windows.

- *title*$_{in}$ is a string containing the title of the window, if any.

- *menu*$_{in}$ is the pull-down menu associated with the window, if any.

The third overloaded function is specific to OS/2.

- *name*$_{in}$ is the name of the object being registered. These are Zinc names such as UIW_VT_LIST, UIW_BUTTON, UIW_WINDOW, etc.

- *baseName*$_{in}$ is the name of the class from which the object being registered is derived. *baseName* is the base class of the OS/2 object and does not necessarily follow Zinc's class hierarchy.

- *baseCallback*$_{in/out}$ is the address of the default callback function that OS/2 provides for each object.

- *title*$_{in}$ is a string containing the title of the window, if any.

The fourth overloaded function is specific to Motif.

- *widgetClass*$_{in}$ is the type of Xt widget that is to be created. If this parameter is used, the *convenienceFunction* parameter should be NULL.

- *convenienceFunction*$_{in}$ is the convenienceFunction that is to be used to create the object. If this parameter is used, the *widgetClass* parameter should be NULL.

- *ccode*$_{in}$ distinguishes the circumstances under which **RegisterObject( )** is being called (i.e., **RegisterObject( )** could be called with an S_SIZE, an S_CREATE or some other ccode. This parameter allows the function to distinguish between the various events).

- *useArgs*$_{in}$ specifies whether the args array has been filled at all prior to the call to **RegisterObject( )**. If *useArgs* is TRUE, then the array has been partially filled already.

- *manage*$_{in}$ indicates if the widget created by the corresponding convenience function should be managed. Since most convenience functions don't manage their widget by default, setting *manage* to TRUE will cause the widget to be managed.

- *_parent*$_{in}$ specifies the Xt parent of the object.

Example

```
EVENT_TYPE UIW_VT_LIST::Event(const UI_EVENT &event)
{
    UI_WINDOW_OBJECT *object;

    // Switch on the event type.
    EVENT_TYPE ccode = LogicalEvent(event);
    switch (ccode)
    {
    .
    .
    .

    case S_CREATE:
        UI_WINDOW_OBJECT::Event(event);
        RegisterObject("UIW_VT_LIST", "LISTBOX", &_listOffset,
            &_listJumpInstance, &_listCallback, NULL);
        SendMessage(screenID, WM_SETREDRAW, FALSE, 0);
        for (object = First(); object; object = object->Next())
            object->Event(event);
        SendMessage(screenID, WM_SETREDRAW, TRUE, 0);
        break;
    .
    .
    .

    }

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

UI_WINDOW_OBJECT::SearchID

Syntax

#include <ui_win.hpp>

OBJECTID SearchID(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This member function returns the *searchID* associated with the object.

- *returnValue*$_{out}$ is the current search identification for the specified object.

NOTE: *searchID* can be any of the **ID_** values (defined in **UI_GEN.HPP**).

Example

```
#include <ui_win.hpp>

ExampleFunction1(UIW_WINDOW *window)
{
    .
    .
    .
    OBJECTID searchID = window->SearchID();
    .
    .
    .
}
```

UI_WINDOW_OBJECT::Store

Syntax

#include <ui_win.hpp>

virtual void Store(const char **name*, UI_STORAGE **file*,
UI_STORAGE_OBJECT **object*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to write an object to disk.

- $name_{in}$ is the name of the object to be stored.
- $file_{in}$ is a pointer to a UI_STORAGE that contains the persistent object. (For more information on storing persistent objects, see "Chapter 35—UI_STORAGE.")

- *object*$_{in}$ is a pointer to the UI_STORAGE_OBJECT containing the persistent object information. (For more information on persistent objects, see "Chapter 36—UI_-STORAGE_OBJECT.")

Example

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

static UIW_WINDOW *HelloWorldWindow1()
{
    // Create the standard Hello World! window.
    UIW_WINDOW *window = UIW_WINDOW::Generic(2, 2, 40, 6, "Hello World Window");
    window->StringID("HELLO_WORLD_WINDOW");

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

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

main()
{
    // Add a window to the hello.dat object file.
    UIW_WINDOW *window1 = HelloWorldWindow1();
    UI_STORAGE storage("test.dat", UIS_READWRITE);
    UI_STORAGE_OBJECT sObject(&storage, "HELLO_WORLD_WINDOW", ID_WINDOW,
        UIS_WRITE);
    window1->Store("HELLO", &storage, &sObject);
    return (0);
}
```

UI_WINDOW_OBJECT::StringID

Syntax

#include <ui_win.hpp>

char *StringID(const char *_stringID_ = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This member function sets the string identification associated with the window object.

- *returnValue*$_{out}$ is the current string identification for the specified object.
- *stringID*$_{in}$ is the string value to associate with the object. If this value is NULL, the string identification is not redefined, but the current string identification is still passed back.

Example

```
#include <ui_win.hpp>

UIW_WINDOW *ExampleFunction1(void)
{
    // Create the standard Hello World! window.
    UIW_WINDOW *window = UIW_WINDOW::Generic(2, 2, 40, 6, "Hello World Window");
    window->StringID("HELLO_WORLD_WINDOW");
    // Add the window objects to the window.
    *window
        + new UIW_TEXT(0, 0, 0, 0, "Hello, World!", 256,
            TXF_NO_FLAGS, WOF_NON_FIELD_REGION);

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

UI_WINDOW_OBJECT::UserFunction

Syntax

#include <ui_win.hpp>

EVENT_TYPE UserFunction(const UI_EVENT &event, EVENT_TYPE ccode);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to either call the object's user function, if it exists, or validate the

window object. When a window object receives the L_SELECT, S_CURRENT or S_NON_CURRENT messages, it will call **UserFunction( )**. **UserFunction( )** calls the object's **userFunction( )** if it exists. Otherwise it calls the object's **Validate( )** function.

- *returnValue*$_{out}$ is the return value from the user function, or 0 if the validation was successful or non-zero in the event of a validation error.

- *event*$_{in}$ is the event that caused **UserFunction( )** to be called. *event* is passed to the user function.

- *ccode*$_{in}$ is the object's interpretation of the event. It is used to determine the type of action that is to take place.

UI_WINDOW_OBJECT::Validate

Syntax

#include <ui_win.hpp>

virtual int Validate(int *processError* = TRUE);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to validate window objects. When a window object receives the L_SELECT, S_CURRENT or S_NON_CURRENT messages, it will call **Validate( )**. In UI_WINDOW_OBJECT, **Validate( )** always returns 0 (i.e., object is OK.) For more information, see derived objects such as UIW_BIGNUM, UIW_DATE and UIW_TIME for actual implementations of **Validate( )** routines.

- *returnValue*$_{out}$ is 0 if the validation was successful or non-zero in the event of an error.

- *processError*$_{in}$ determines whether **Validate( )** should call **UI_ERROR_SYSTEM::-ReportError( )**. If *processError* is TRUE, and an error occurs, the error system is called. Otherwise, the error is returned and no error message is generated.

CHAPTER 41 – UID_CURSOR

The UID_CURSOR class is used to display a blinking cursor on the screen. It is primarily used by objects that can be edited in order to show the end-user's position within the edit buffer. In text mode, this class uses BIOS calls to enable or disable the blinking hardware cursor. In graphics mode, this class paints a blinking cursor on the screen.

The UID_CURSOR class is declared in **UI_EVT.HPP**. Its public and protected members are:

```
class EXPORT UID_CURSOR : public UI_DEVICE
{
public:
    // Members described in UID_CURSOR reference chapter.
    static int blinkRate;

    DEVICE_IMAGE image;
    UI_POSITION position;

    UID_CURSOR(DEVICE_STATE state = D_OFF, DEVICE_IMAGE image = DC_INSERT);
    virtual ~UID_CURSOR(void);
    virtual EVENT_TYPE Event(const UI_EVENT &event);

protected:
    // Members described in UID_CURSOR reference chapter.
    UI_POSITION offset;

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

- *blinkRate* identifies the rate at which the cursor will blink. This value is in hundredths of seconds.

- *image* identifies the type of cursor being displayed on the screen. Its value may either be DC_INSERT or DC_OVERSTRIKE. If *image* is DC_INSERT, the cursor device displays a thick vertical bar cursor on the screen. If it is DC_OVERSTRIKE, the cursor device displays a thin vertical bar cursor on the screen.

- *position* gives the cursor's true screen position (based on the screen's 0,0 left-top based coordinates). The value of this structure depends on the type of display mode in which the application is running. For example, a cursor positioned in the middle of the screen may contain a *position.column* value of 40 and *position.line* value of 12, if the application is running in text mode. The same cursor position, however, may produce values of 320 and 240 if the application is running in graphics mode.

- *offset* is an offset, from *position*, where the cursor image will be displayed.

UID_CURSOR::UID_CURSOR

Syntax

```
#include <ui_evt.hpp>

UID_CURSOR(DEVICE_STATE state = D_OFF,
    DEVICE_IMAGE image = DC_INSERT);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UID_CURSOR class object. It should be called after the display and Event Manager constructors have been called.

- $state_{in}$ is the initial state of the cursor device. The cursor device may be initialized to one of the following states (defined in **UI_EVT.HPP**):

  D_OFF—Initializes the cursor to a non-blinking state. In this state, the cursor is not shown on the screen. (This is the default value if no argument is provided.)

  D_ON—Initializes the cursor to a blinking state. The initial state mask is DC_INSERT.

- $image_{in}$ identifies the initial type of cursor being displayed on the screen. Its value may be one of the following types (defined in **UI_EVT.HPP**):

  DC_INSERT—The cursor device displays a thick vertical bar cursor on the screen. (This is the default value if no argument is provided.)

  DC_OVERSTRIKE—The cursor device displays a thin vertical bar cursor on the screen.

Example

```
#include <ui_evt.hpp>

main()
{
    // Attach the keyboard, mouse and cursor devices.
    UI_TEXT_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    .
    .
    .

    // This automatically calls the keyboard, mouse and cursor destructors.
    delete eventManager;
    delete display;
    return (0);
}
```

UID_CURSOR::~UID_CURSOR

Syntax

#include <ui_evt.hpp>

virtual ~UID_CURSOR(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UID_CURSOR object. Care should be taken to only destroy a cursor device that is not already attached to the Event Manager.

Example

```
#include <ui_evt.hpp>

ExampleFunction()
{
    // Attach the keyboard, mouse and cursor devices.
    UI_TEXT_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    .
    .
    .
    // This automatically calls the keyboard, mouse and cursor destructors.
    delete eventManager;
    delete display;
    return (0);
}
```

UID_CURSOR::Event

Syntax

#include <ui_evt.hpp>

virtual EVENT_TYPE Event(const UI_EVENT *&event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to the cursor device. It is declared virtual so that any derived cursor class can override its default operation.

- $returnValue_{out}$ is the current state of the cursor device. This value will be D_OFF, DC_INSERT or DC_OVERSTRIKE.

- $event_{in}$ contains the new state information. The following messages (declared in **UI_EVT.HPP**) are recognized by the **Event( )** routine:

D_OFF—Turns off the cursor. If the cursor is "OFF" no text or graphics cursor is shown on the screen.

D_ON—Turns on the cursor. If the cursor is "ON" a text or graphics cursor is shown on the screen.

DC_INSERT—Turns the cursor on and enables the insert cursor. In this state the cursor device displays a thick vertical bar cursor on the screen.

DC_OVERSTRIKE—Turns the cursor on and enables the overstrike cursor. In this state the cursor device displays a thin vertical bar cursor on the screen.

S_POSITION—Changes the screen position of the cursor. If this message is sent, *event.position.column* and *event.position.line* must contain the run-time display position of the cursor on the screen. The values of *event.position.column* and *event.position.line* depend on the type of display mode in which the application is running. For example, if the cursor is to be positioned at the center of the screen while the application is running in text mode (i.e., a 80 column by 25 line screen) the position values should be:

```
event.position.column = 40;
event.position.line = 13;
```

If, on the other hand, the application is running in a 640 column by 480 line graphics mode, the position values should be:

```
event.position.column = 320;
event.position.line = 240;
```

If the cursor is in a D_OFF state, the position change will be reflected when the cursor is turned back on.

The state of the cursor device may also be changed using the **UI_EVENT_MANAGER::-Event( )** or **UI_EVENT_MANAGER::DeviceState( )** routines.

Example

```
#include <ui_evt.hpp>

ExampleFunction()
{
    // Attach the keyboard to the event manager.
    UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    UID_CURSOR *cursor = new UID_CURSOR;
    *eventManager + cursor;
```

```
    .
    .

    // Change the cursor to insert mode.
    UI_EVENT event;
    event.type = DC_INSERT;
    cursor->Event(event);
    .
    .
    .
    // Reposition the cursor the top-left side of the screen.
    event.type = S_POSITION;
    event.position.column = event.position.line = 0;
    cursor->Event(event);
    .
    .
    .
}
```

UID_CURSOR::Poll

Syntax

#include <ui_evt.hpp>

virtual void Poll(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine determines whether the cursor is in a blinking state. If it is, and the application is running in graphics mode, it turns the cursor on and off, generating the blinking effect. In text mode, this routine has no effect. In addition, if the cursor is in a disabled state or is turned off, this routine has no effect. This routine is declared virtual so that any derived cursor class can override its default operation.

An example of the **Poll( )** member function is given in **UI_DEVICE::Poll( )**.

CHAPTER 42 – UID_KEYBOARD

The UID_KEYBOARD class is used to get event information from the keyboard. This class implements a BIOS level keyboard interface that auto-detects for regular or enhanced keyboards. Most compiler libraries have a set of functions to get input from the keyboard (e.g., **getch( )**, **getchar( )**). However, in Zinc Application Framework, the keyboard is interfaced with other devices, such as a mouse, to provide smooth control of the user's input. (Keyboard information is placed into the event queue by the keyboard's **Poll( )** routine and retrieved by the programmer using the **UI_EVENT_MANAGER::-Get( )** routine.)

The UID_KEYBOARD class is declared in **UI_EVT.HPP**. Its public and protected members are:

```
class EXPORT UID_KEYBOARD : public UI_DEVICE
{
public:
    // Members described in UID_KEYBOARD reference chapter.
    static EVENT_TYPE breakHandlerSet;

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

protected:
    // Members described in UID_KEYBOARD reference chapter.
    virtual void Poll(void);
};
```

- *breakHandlerSet* determines whether <Ctrl+Break> can be pressed in order to exit the application. When the UID_KEYBOARD class constructor is called, the initial breakState information is saved and then replaced by *breakHandlerSet*. By default *breakHandlerSet* is set to L_EXIT, meaning that instead of exiting the program immediately, it generates an L_EXIT message which allows the programmer to terminate the application. *breakHandlerSet* may be set to a Zinc message that will be placed at the front of the Event Manager's queue when <Ctrl+Break> is pressed. When the UID_KEYBOARD class destructor is called, the *breakHandlerSet* is replaced by the initial breakState information. The example below shows how this can be done:

```
main()
{
    // Reset the break handler.
    UID_KEYBOARD::breakHandlerSet = L_EXIT_FUNCTION;

    UI_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
```

```
        .
        .
        .
    }
```

The keyboard device provides the following event information (declared in **UI_EVT.HPP**) when a key is retrieved using the **UI_EVENT_MANAGER::Get( )** function:

```
struct EXPORT UI_KEY
{
    RAW_CODE shiftState;
    RAW_CODE value;
};

struct EXPORT UI_EVENT
{
    EVENT_TYPE type;              // The type of event (E_KEY).
    RAW_CODE rawCode;             // The key's raw scan code.
    RAW_CODE modifiers;

#if defined(ZIL_MSWINDOWS)
    MSG message;              // Windows message field.
#elif defined(ZIL_OS2)
    QMSG message;             // OS/2 message field.
#elif defined(ZIL_MOTIF)
    XEvent message;           // Motif mesage field.
#endif

    union
    {
        UI_KEY key;               // The key information.
        UI_REGION region;
        UI_POSITION position;
        UI_SCROLL_INFORMATION scroll;
        void *data;
    };

    // Member functions are described in the UI_EVENT reference chapter.
    .
    .
    .

};
```

- *type* is the event type. The UID_KEYBOARD device always generates an E_KEY type event.

- *rawCode* is the key's raw scan code. The following list shows the scan code values for selected keys:

```
'a' key (lower-case)        0x1E61
'A' key (upper-case)        0x1E41
enter key                   0x1C0D
escape key                  0x011B
F1 key                      0x3B00
gray up-arrow               0x48E0
```

A set of DOS function key scan code and *#define* equivalents is provided in **UI_EVT.HPP**. For example, the function keys shown above have the following declarations:

```
#define ENTER                0x1C0D
#define ESCAPE               0x011B
#define F1                   0x3B00
#define GRAY_UP_ARROW        0x48E0
```

- *modifiers* is a bit field that indicates the shift state of the keyboard.
- *key.shiftState* is the shift state of the keyboard. The shift state may contain one or more of the following flags (declared in **UI_EVT.HPP**):

  S_ALT—The <Alt> key is pressed.

  S_CAPS_LOCK—The <Caps-Lock> key is on.

  S_CTRL—The <Ctrl> key is pressed.

  S_INSERT—The <Ins> key is on.

  S_LEFT_SHIFT—The <Left-Shift> key is pressed.

  S_NUM_LOCK—The <Num-Lock> key is on.

  S_RIGHT_SHIFT—The <Right-Shift> key is pressed.

  S_SCROLL_LOCK—The <Scroll-Lock> key is on.

- *key.value* is the low eight bits of the scan code. If a non-function key is pressed, this gives the ASCII value of the key. For example, the character 'a' produces a scan code of 0x1E61 but has an associated ASCII value of 0x61 (i.e., the character 'a' in the ASCII character set). The programmer should use *key.value* when ASCII character values are desired, not *rawCode*.

UID_KEYBOARD::UID_KEYBOARD

Syntax

#include <ui_evt.hpp>

UID_KEYBOARD(DEVICE_STATE *state* = D_ON);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UID_KEYBOARD class object. It should be called after the display and Event Manager constructors have been called.

- $state_{in}$ is the initial state of the keyboard device. The keyboard device may be initialized to one of the following states (declared in **UI_EVT.HPP**):

  D_OFF—Initializes the keyboard but disables events. If the keyboard state is set to D_OFF, events are removed from the keyboard BIOS but are not placed in the event queue (i.e., they are discarded).

  D_ON—Initializes the keyboard to feed keyboard information to the event queue. This is the default value if no argument is provided.

  The state of the UID_KEYBOARD device can be changed at run-time using the **UID_KEYBOARD::Event( )**, **UI_EVENT_MANAGER::Event( )** or **UI_EVENT_-MANAGER::DeviceState( )** function calls.

Example

```
#include <ui_evt.hpp>

main()
{
    // Attach the keyboard, mouse and cursor devices.
    UI_TEXT_DISPLAY *display = new UI_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    .
    .
    .

    // This automatically calls the keyboard, mouse and cursor destructors.
    delete eventManager;
    delete display;
    return (0);
}
```

UID_KEYBOARD::~UID_KEYBOARD

Syntax

#include <ui_evt.hpp>

virtual ~UID_KEYBOARD(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UID_-KEYBOARD object and closes the BIOS interface to the keyboard. Care should be taken to only destroy a keyboard device that is not already attached to the Event Manager.

Example

```
#include <ui_evt.hpp>

main()
{
    // Attach the keyboard, mouse and cursor devices.
    UI_DISPLAY *display = new UI_MSC_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    .
    .
    .

    // This automatically calls the keyboard, mouse and cursor destructors.
    delete eventManager;
    delete display;
}
```

UID_KEYBOARD::Event

Syntax

#include <ui_evt.hpp>

virtual EVENT_TYPE Event(const UI_EVENT *&event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function is used to send run-time information to a keyboard device. It is declared virtual so that any derived keyboard class can override its default operation.

- *returnValue*$_{out}$ is the current state of the keyboard device. This value will be D_OFF or D_ON.

- *event*$_{in}$ contains the new state information. The following messages (declared in **UI_EVT.HPP**) are recognized by the **Event( )** routine:

  D_OFF—Turns off the keyboard. If the keyboard is "OFF," information is removed from the keyboard BIOS but is not placed in the event queue (i.e., the information is discarded by the **Poll( )** routine).

  D_ON—Turns the keyboard on. When the keyboard is "ON," information is retrieved from the keyboard BIOS and placed in the Event Manager's event queue.

The state of the keyboard device may also be changed using the **UI_EVENT_-MANAGER::Event( )** or **UI_EVENT_MANAGER::DeviceState( )** routines.

Example

```
#include <ui_evt.hpp>

main()
{
    // Attach the keyboard to the event manager.
    UI_DISPLAY *display = new UI_MSC_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    UID_KEYBOARD *keyboard = new UID_KEYBOARD;
    *eventManager + keyboard;
    .
    .
    .

    // Turn the keyboard off directly.
    UI_EVENT event;
    event.type = D_OFF;
    keyboard->Event(event);
    .
    .
    .

}
```

UID_KEYBOARD::Poll

Syntax

#include <ui_evt.hpp>

virtual void Poll(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine allows the keyboard to put information into the event queue. It is declared virtual so that any derived keyboard class can override its default operation.

When the Event Manager's loop is called by **UI_EVENT_MANAGER::Get( )**, the Event Manager calls the **Poll( )** routine. This routine operates according to the following algorithm:

1—Go to the keyboard BIOS to see if a key has been pressed.

2—If a key has been pressed, get the key's information (i.e., shift state and scan code) then put it into the Event Manager's queue using the **UI_EVENT_-MANAGER::Put( )** routine.

An example of the **Poll( )** member function is given in **UI_DEVICE::Poll( )**.

CHAPTER 43 – UID_MOUSE

The UID_MOUSE class is used to get event information from a mouse device. This class implements an interrupt level mouse device that conforms to the operating protocol specified by the Microsoft mouse driver. (Mouse information is placed into the event queue by the mouse interrupt service routine and retrieved by the programmer using the **UI_EVENT_MANAGER::Get( )** function.)

The UID_MOUSE class is declared in **UI_EVT.HPP**. Its public and protected members are:

```
class EXPORT UID_MOUSE : public UI_DEVICE
{
public:
    DEVICE_IMAGE image;
    UI_POSITION position;

    UID_MOUSE(DEVICE_STATE state = D_ON, DEVICE_IMAGE image = DM_WAIT);
    virtual ~UID_MOUSE(void);
    virtual EVENT_TYPE Event(const UI_EVENT &event);

protected:
    UI_POSITION offset;

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

- *image* identifies the type of mouse cursor being displayed on the screen. Its value may be: DM_VIEW, DM_EDIT, DM_WAIT, DM_MOVE, DM_HORIZONTAL, DM_VERTICAL, DM_DIAGONAL_ULLR, DM_DIAGONAL_LLUR, DM_-POSITION. See the **Event( )** function for a complete description of these types.

- *position* gives the mouse cursor's true screen position (based on the screen's 0,0 left-top based coordinates). The value of this structure depends on the type of display mode in which the application is running. For example, a mouse cursor positioned in the middle of the screen may contain a *position.column* value of 40 and *position.line* value of 12, if the application is running in text mode. The same mouse cursor position, however, may produce values of 320 and 240 if the application is running in VGA graphics mode.

- *offset* is an offset, from *position*, where the mouse image will be displayed.

Mouse event information

The mouse device provides the following event information (declared in **UI_EVT.HPP**) when a mouse event is retrieved using the **UI_EVENT_MANAGER::Get( )** function:

```
struct UI_POSITION
{
    int column;                 // The mouse column position.
    int line;                   // The mouse line position.
};

struct EXPORT UI_EVENT
{
    EVENT_TYPE type;            // The type of event (E_KEY).
    RAW_CODE rawCode;           // The key's raw scan code.
    RAW_CODE modifiers;

#if defined(ZIL_MSWINDOWS)
    MSG message;            // Windows message field.
#elif defined(ZIL_OS2)
    QMSG message;           // OS/2 message field.
#elif defined(ZIL_MOTIF)
    XEvent message;         // Motif mesage field.
#endif

    union
    {
        UI_KEY key;             // The key information.
        UI_REGION region;
        UI_POSITION position;
        UI_SCROLL_INFORMATION scroll;
        void *data;
    };

    // Member functions are described in the UI_EVENT reference chapter.
    .
    .
    .

};
```

- *type* is the event type. The mouse device always generates an E_MOUSE type event. If an event is sent directly to **Event( )**, *type* must contain one of the values listed under *event* in the **Event( )** section.

- *rawCode* is the keyboard's shift state and the mouse's button states listed below:

  M_LEFT—The left mouse button is pressed.

  M_LEFT_CHANGE—The left mouse button state has changed. If the M_-LEFT_CHANGE and M_LEFT flags are set, the left button has just been pressed. Otherwise, the left button has just been released.

  M_MIDDLE—The middle mouse button is pressed. (This flag will only be set when a three-button mouse is in use.)

  M_MIDDLE_CHANGE—The middle mouse button state has changed. If the M_MIDDLE_CHANGE and M_MIDDLE flags are set, the middle button has just been pressed. Otherwise, the middle button has just been released. (This flag will only be set when a three-button mouse is in use.)

M_RIGHT—The right mouse button is pressed.

M_RIGHT_CHANGE—The right mouse button state has changed. If the M_RIGHT_CHANGE and M_RIGHT flags are set, the right button has just been pressed. Otherwise, the right button has just been released.

S_ALT—The <Alt> key is pressed.

S_CAPS_LOCK—The <Caps-Lock> key is on.

S_CTRL—The <Ctrl> key is pressed.

S_INITIALIZE—Initializes internal information associated with the mouse device. This message is automatically sent by the UID_MOUSE class constructor and should not be used by programmers.

S_INSERT—The <Ins> key is on.

S_LEFT_SHIFT—The <Left-Shift> key is pressed.

S_NUM_LOCK—The <Num-Lock> key is on.

S_POSITION—Changes the screen position of the mouse. If this message is sent, *event.position.column* and *event.position.line* must contain the run-time display position of the mouse on the screen. The values of *event.position.column* and *event.position.line* depend on the type of display mode in which the application is running. For example, if the mouse is to be positioned at the center of the screen while the application is running in text mode (i.e., a 80 column by 25 line screen) the position values should be:

```
event.position.column = 40;
event.position.line = 13;
```

If, on the other hand, the application is running in a 640 column by 480 line graphics mode, the position values should be:

```
event.position.column = 320;
event.position.line = 240;
```

If the mouse is in a D_OFF state, the position change will be reflected when the mouse is turned back on.

S_RIGHT_SHIFT—The <Right-Shift> key is pressed.

S_SCROLL_LOCK—The <Scroll-Lock> key is on.

NOTE: The **M_TOP_CHANGE** and **M_BOTTOM_CHANGE** values are only used when a window object is to be sized. They are not set by the UID_MOUSE class.

- *modifiers* is a bit field that indicates the shift state of the keyboard.
- *position.column* is the column (horizontal) position of the mouse on the screen. In graphics mode, this value is given in pixel coordinates. In text mode, this value is given in character coordinates.
- *position.line* is the line (vertical) position of the mouse on the screen. In graphics mode, this value is given in pixel coordinates. In text mode, this value is given in character coordinates.

UID_MOUSE::UID_MOUSE

Syntax

```
#include <ui_evt.hpp>

UID_MOUSE(DEVICE_STATE state = D_ON,
    DEVICE_IMAGE image = DM_WAIT);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UID_MOUSE class object. It should be called after the display and Event Manager constructors have been called.

- $initialState_{in}$ is the initial state of the mouse device. The mouse device may be set to one of the following states (declared in **UI_EVT.HPP**):

  D_OFF—Initializes the mouse but disables events. If the mouse is set to the D_OFF state, mouse interrupt events are not placed in the event queue.

D_ON—Initializes the mouse to feed event information to the event queue. This is the default value if no argument is provided.

- $image_{in}$ identifies the initial type of mouse image being displayed on the screen. Its value may be one of the following types (defined in **UI_EVT.HPP**): DM_VIEW, DM_EDIT, DM_WAIT, DM_MOVE, DM_HORIZONTAL, DM_VERTICAL, DM_-DIAGONAL_ULLR, DM_DIAGONAL_LLUR, DM_POSITION. See the **Event( )** function for a complete description of these types. DM_WAIT is the default image if no argument is provided.

The state of the UID_MOUSE device can be changed at run-time using the **Event( )**, **UI_EVENT_MANAGER::Event( )** or **UI_EVENT_MANAGER::DeviceState( )** function calls.

Example

```
#include <ui_evt.hpp>

main()
{
    // Attach the keyboard, mouse and cursor devices.
    UI_DISPLAY *display = new UI_MSC_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    .
    .
    .

    // This automatically calls the keyboard, mouse and cursor destructors.
    delete eventManager;
    delete display;
    return (0);
}
```

UID_MOUSE::~UID_MOUSE

Syntax

#include <ui_evt.hpp>

virtual ~UID_MOUSE(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual destructor destroys the class information associated with the UID_MOUSE object and closes the interrupt associated with the mouse. Care should be taken to only destroy a mouse device that is not already attached to the Event Manager.

Example

```
#include <ui_evt.hpp>

main()
{
    // Attach the keyboard, mouse and cursor devices.
    UI_DISPLAY *display = new UI_MSC_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_MOUSE
        + new UID_CURSOR;
    .
    .
    .

    // This automatically calls the keyboard, mouse and cursor destructors.
    delete eventManager;
    delete display;
    return (0);
}
```

UID_MOUSE::Event

Syntax

#include <ui_evt.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &event);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a mouse device. It is declared virtual so that any derived mouse class can override its default operation.

- *returnValue*$_{out}$ is the current state of the mouse device. This value will be D_OFF or D_ON.

- *event*$_{in}$ contains the new state information. All information passed to the mouse device must have the state message contained in *event.type*. The following messages (declared in **UI_EVT.HPP**) are recognized by **Event( )**:

  DM_DIAGONAL_LLUR—Changes the mouse to an upward diagonal arrow.

  DM_DIAGONAL_ULLR—Changes the mouse to a downward diagonal arrow.

  DM_EDIT—Changes the mouse to a vertical line.

  DM_HORIZONTAL—Changes the mouse to a horizontal arrow.

  DM_MOVE—Changes the mouse to a four-way arrow.

  DM_POSITION—Changes the mouse to a cross-hair.

  DM_VERTICAL—Changes the mouse to a vertical arrow.

  DM_VIEW—Changes the mouse to a left arrow.

  DM_WAIT—Changes the mouse to an hour glass.

The state of the mouse device may also be changed using the **UI_EVENT_-MANAGER::Event( )** or **UI_EVENT_MANAGER::DeviceState( )** routines.

Example

```
#include <ui_evt.hpp>

main()
{
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    UID_MOUSE *mouse = new UID_MOUSE;
    *eventManager + mouse;
    .
    .
    .

    // Turn on the mouse wait cursor.
    UI_EVENT event;
    event.type = DM_WAIT;
    mouse->Event(event);
    .
    .
    .

    // Turn on the mouse arrow cursor.
    UI_EVENT event;
    event.type = DM_VIEW;
    mouse->Event(event);
}
```

UID_MOUSE::Poll

Syntax

#include <ui_evt.hpp>

virtual void Poll(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine feeds input information from an internal mouse buffer to the event queue.

An example of the **Poll( )** member function is given in **UI_DEVICE::Poll( )**.

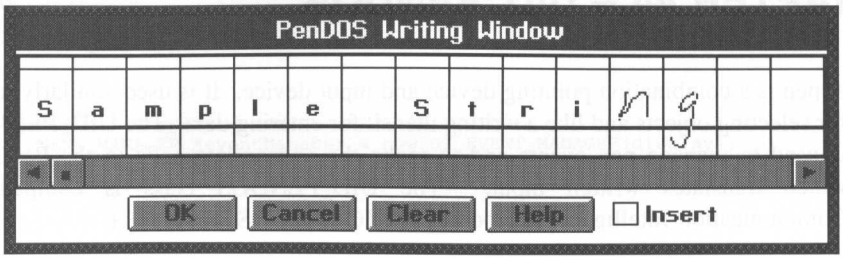
CHAPTER 44 – UID_PENDOS

A pen is a combination pointing device and input device. It is used similarly to a mouse for selecting objects and like a writing utensil for entering data. The UID_PENDOS class is used to manage pen events and to make the appropriate calls to the PenDOS API, which translates written input. The UID_PENDOS class is compatible with Communication Intelligence Corporation (CIC) PenDOS version 1.1.

Using a pen to interact with non-editable objects (e.g., buttons, scroll bars, etc.) is very much like using a mouse. Touching the pen to the screen generates the same event as clicking with the left mouse button. Moving the pen while it is touching the screen is the same as holding the left mouse button and moving the mouse. Lifting the pen up off the screen is the same as releasing the left mouse button. Pressing and releasing the side button on the pen generates the same messages as pressing and releasing the right mouse button.

Using a pen to interact with editable objects (e.g., UIW_STRING, UIW_TEXT, etc.) is like writing on a piece of paper. To enter text, the pen is used to simply write the desired characters directly on the screen. The strokes that the user draws when writing a character are translated by CIC PenDOS into JIS (Japanese Industrial Standard) which can then be translated into ASCII.

The pen device generates the same raw codes as a mouse device (even though the use of the pen device is slightly different than that of a mouse), and so events generated by the pen device generally are handled as mouse events would be. Thus, the UID_PENDOS class is derived from the UID_MOUSE class so that most raw events can be handled by the UID_MOUSE class. The only exception to how UID_PENDOS handles raw events occurs when the end user wishes to edit an editable object. If the user taps an editable field with the pen, a PenDOS window pops up to allow the user to edit the data for that field. This PenDOS window (with some already recognized characters and some pending characters) is shown below:



The grid field serves three purposes: it displays the current text (if any); it allows new text to be written using the pen; and it shows the translation of each character as new information is written. For information on drawing characters, see the CIC PenDOS User Manual.

There are several special pen gestures (a gesture being one or more strokes made while holding the side button in) and characters (a character being one or more strokes that are recognized and interpreted) that should be noted:

- To insert a space, draw a '⊥' or 'Λ' while holding the side button in.
- To delete characters, simply draw a line through the undesired character while holding the side button in. For multiple characters, draw one long, continuous line through all the characters.
- To enter a space draw a '∩'.
- To enter a hard return (in a UIW_TEXT field), first draw a forward slash ('/'), and wait for it to be recognized as a slash. Then draw another forward slash on top of the first one. The forward slash will then display as a '■' indicating that a return has been entered.

The scroll bar attached to the grid is used to scroll through the text.

Selecting the "OK" button causes the PenDOS window to be removed from the screen and the new data to be placed in the field.

Selecting the "Cancel" button causes the PenDOS window to be removed from the screen. All changes to the text will be ignored.

Selecting the "Clear" button causes the text associated with the field to be cleared from the text entry grid.

Selecting the "Help" button displays additional help.

Selecting the "Insert" button toggles between insert mode and overstrike mode. In insert mode (i.e., the button is displayed as checked), writing a character on top of another character causes all the text from that position on to be shifted right one space and the new character to be inserted. Overstrike mode (i.e., the button is displayed as not checked) causes a character to be replaced by the new one when written over.

The PenDOS window is a temporary window, so if another window is made current, the PenDOS window is removed from the screen. If this happens, the new data is written to the field. The PenDOS window updates the field by sending a series of E_KEY messages to the object being edited.

The UID_PENDOS class is declared in **UI_PEN.HPP**. Its public and protected members are:

```
class EXPORT UID_PENDOS : public UIW_WINDOW, public UID_MOUSE
{
    friend class UID_PENDOS_SCROLL;
public:

    // Members described in UID_PENDOS reference chapter.
    int insertMode;
    UID_PENDOS(void);
    virtual ~UID_PENDOS(void);
    virtual EVENT_TYPE Event(const UI_EVENT &event);

protected:
    int firstVisible;
    UIW_BUTTON *insertButton;
    char *text;
    int textLen;
    UIW_WINDOW *writeWindow;

    void DrawText(int startCell = -1, int endCell = -1);
    int ObtainRecognizedResults(int cleanup = FALSE);
    virtual void Poll(void);
    void SetTimer(void);
    void ShowWritingWindow(UI_WINDOW_OBJECT *object);
    int TimeExpired(void);
};
```

- *UID_PENDOS_SCROLL* is a support structure and should not be used.
- *insertMode* indicates if the text entry grid is in insert mode or overstrike mode.
- *firstVisible* is the offset into the string of the first character visible in the text entry grid.

- *insertButton* is a pointer to the "Insert" check box on the PenDOS window.
- *text* is a pointer to the text string for the field being edited.
- *textLen* is the maximum length of the text.
- *writeWindow* is a pointer to the window that contains the text entry grid.

NOTE: A UID_PENDOS device and a UID_MOUSE device should never be used at the same time.

Pen event information

The pen device provides the following event information (declared in **UI_EVT.HPP**) when a pen event is retrieved using the **UI_EVENT_MANAGER::Get( )** function:

```
struct UI_POSITION
{
    int column;                 // The pen column position.
    int line;                   // The pen line position.
};

struct EXPORT UI_EVENT
{
    EVENT_TYPE type;            // The type of event (E_MOUSE).
    RAW_CODE rawCode;           // The pen's raw code.
    RAW_CODE modifiers;

#if defined(ZIL_MSWINDOWS)
    MSG message;            // Windows message field.
#elif defined(ZIL_OS2)
    QMSG message;           // OS/2 message field.
#elif defined(ZIL_MOTIF)
    XEvent message;         // Motif mesage field.
#endif

    union
    {
        UI_KEY key;             // The key information.
        UI_REGION region;
        UI_POSITION position;
        UI_SCROLL_INFORMATION scroll;
        void *data;
    };

    // Member functions are described in the UI_EVENT reference chapter.
    .
    .
    .

};
```

- *type* is the event type. The pen device always generates an E_MOUSE/E_PENDOS type event. This is because the **UID_MOUSE::Poll( )** function actually handles all

pen events and places them on the queue. Note that the definitions for E_MOUSE and E_PENDOS are equivalent.

- *rawCode* is the keyboard's shift state and the pen's button state listed below:

  M_LEFT—The pen is touching the screen.

  M_LEFT_CHANGE—The pen has either just been placed on the screen or just been lifted from the screen. If the M_LEFT_CHANGE and M_LEFT flags are both set, the pen has just been placed on the screen. Otherwise, the pen has just been lifted.

  M_RIGHT—The side button is pressed.

  M_RIGHT_CHANGE—The side button state has changed. If the M_RIGHT_-CHANGE and M_RIGHT flags are both set, the side button has just been pressed. Otherwise, the side button has just been released.

  S_ALT—The <Alt> key is pressed.

  S_CAPS_LOCK—The <Caps-Lock> key is on.

  S_CTRL—The <Ctrl> key is pressed.

  S_INITIALIZE—This initializes internal information associated with the pen device. This message is automatically sent by the UID_PENDOS class constructor and should not be used by programmers.

  S_INSERT—The <Ins> key is on.

  S_LEFT_SHIFT—The <Left-Shift> key is pressed.

  S_NUM_LOCK—The <Num-Lock> key is on.

  S_POSITION—This message (which is used by a UID_MOUSE device) is suppressed. A pen device is an absolute device, so wherever it is placed on the screen is where the device's position is. It does not make sense, nor is it possible, to attempt to modify a pen's position via program control.

  S_RIGHT_SHIFT—The <Right-Shift> key is pressed.

  S_SCROLL_LOCK—The <Scroll-Lock> key is on.

- *modifiers* is a bit field that indicates the shift state of the keyboard.
- *position.column* is the column (horizontal) position of the pen on the screen. Since the pen only operates in graphics mode, this value is given in pixel coordinates.
- *position.line* is the line (vertical) position of the pen on the screen. This value is given in pixel coordinates.

UID_PENDOS::UID_PENDOS

Syntax

#include <ui_pen.hpp>

UID_PENDOS(void);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This constructor returns a pointer to a new UID_PENDOS class object. It should be called after the display and Event Manager constructors have been called.

Example

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

main()
{
    // Attach the keyboard, pen and cursor devices.
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_PENDOS
        + new UID_CURSOR;
    .
    .
    .
```

```
    // This automatically calls the keyboard, pen and cursor destructors.
    delete eventManager;
    delete display;
    return (0);
}
```

UID_PENDOS::~UID_PENDOS

Syntax

#include <ui_pen.hpp>

virtual ~UID_PENDOS(void);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This virtual destructor destroys the class information associated with the UID_PENDOS object. Because the base **UID_MOUSE::~UID_MOUSE( )** destructor is also called, the interrupt handler will be reset. Care should be taken to only destroy a pen device that is not already attached to the Event Manager.

Example

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

main()
{
    // Attach the keyboard, pen and cursor devices.
    UI_DISPLAY *display = new UI_GRAPHICS_DISPLAY;
    UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(display);
    *eventManager
        + new UID_KEYBOARD
        + new UID_PENDOS
        + new UID_CURSOR;
    .
    .
    .
```

```
    // This automatically calls the keyboard, pen and cursor destructors.
    delete eventManager;
    delete display;
    return (0);
}
```

UID_PENDOS::DrawText

Syntax

#include <ui_pen.hpp>

void DrawText(int *startCell* = -1, int *endCell* = -1);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This function redraws a portion of the grid and any characters in the affected cells.

- $startCell_{in}$ is the first grid cell that needs to be redrawn. If *startCell* is -1, the grid is redrawn starting with the first cell.
- $endCell_{in}$ is the last grid cell that needs to be redrawn. If *endCell* is -1, the grid is redrawn through the last cell.

UID_PENDOS::Event

Syntax

#include <ui_pen.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &event);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This advanced routine is used to send run-time information to a pen device. It is declared virtual so that any derived pen class can override its default operation.

- *returnValue*$_{out}$ is the **UIW_WINDOW::Event( )** function's interpretation of the event.

- *event*$_{in}$ contains the event information. The following messages (declared in **UI_PEN.HPP** and **UI_EVT.HPP**) are recognized by **Event( )**:

  E_PENDOS—Causes the *event.rawCode* to be checked. If *event.rawCode* is D_ACTIVATE, the PenDOS window is displayed to allow the user to write data for the field. This message is sent from the **UIW_STRING::Event( )** function if an L_END_MARK message is received and no text has been highlighted for marking (i.e., the user tapped an editable object intending to edit the field). *event.data* will point to the object to be edited.

  UIP_INSERT—Toggles the insert status for the text entry grid on the PenDOS window. This message is sent when the "Insert" check box on the PenDOS window is selected.

  UIP_CANCEL—Removes the PenDOS window from the display and cancels any editing that has been done. This message is sent when the "Cancel" button on the PenDOS window is selected.

  UIP_CLEAR—Clears the text from the text entry grid. This message is sent when the "Clear" button on the PenDOS window is selected.

  UIP_OK—Sets the new text for the object that was being edited and removes the PenDOS window from the display. This message is sent when the "OK" button on the PenDOS window is selected.

  UIP_DRAW_GRID—Draws the text entry grid and any initial text. This message is sent when the PenDOS window becomes current.

S_CURRENT—Determines how many cells will fit in the text entry grid and then causes the grid to be drawn.

S_NON_CURRENT—Forces any remaining strokes to be recognized and then sets the new text for the object being edited, if appropriate.

The remaining messages should be used with caution. They will be ignored normally, but if the **D_PENDOS.CPP** module is re-compiled with the pre-processor macro SHOWMOUSE defined, they will be handled by the **UID_MOUSE::Event( )** function. These messages will cause the mouse cursor to be displayed and are intended to be used for debugging purposes only.

DM_DIAGONAL_LLUR—Changes the mouse cursor to an upward diagonal arrow.

DM_DIAGONAL_ULLR—Changes the mouse cursor to a downward diagonal arrow.

DM_EDIT—Changes the mouse cursor to a vertical line.

DM_HORIZONTAL—Changes the mouse cursor to a horizontal arrow.

DM_MOVE—Changes the mouse cursor to a four-way arrow.

DM_POSITION—Changes the mouse cursor to a cross-hair.

DM_VERTICAL—Changes the mouse cursor to a vertical arrow.

DM_VIEW—Changes the mouse cursor to a left arrow.

DM_WAIT—Changes the mouse cursor to an hour glass.

All other events are handled by the **UID_MOUSE::Event( )** function or the **UIW_WINDOW::Event( )** function.

UID_PENDOS::ObtainRecognizedResults

Syntax

#include <ui_evt.hpp>

int ObtainRecognizedResults(int *cleanup* = FALSE);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This advanced routine calls the PenDOS software to attempt recognition of any outstanding strokes. Any recognized characters are stored in the text string and displayed. This function is called after every stroke.

- *returnValue*$_{out}$ is a count of the number of characters that were recognized.

- *cleanup*$_{in}$ controls whether recognition is forced or if recognition is only accepted if there is no danger of ambiguity. If *cleanup* is TRUE, recognition is forced. If *cleanup* is FALSE, a character is recognized only if the strokes that have been drawn can be uniquely identified. For example, a single vertical stroke could be recognized as either a lower case 'L' or as the initial stroke for an upper case 'E.' A value of FALSE would prevent the character from being recognized at this point, while a value of TRUE would recognize the character as a lower case 'L.'

UID_PENDOS::Poll

Syntax

#include <ui_evt.hpp>

virtual void Poll(void);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This advanced routine first checks to see if a certain length of time has passed since the last stroke was drawn by the user. If 8/10th's of a second (the time interval recommended by CIC) have passed, the PenDOS software is forced to attempt recognition of the strokes that were drawn.

The **Poll( )** function then calls the **UID_MOUSE::Poll( )** function so that it can place any outstanding pen events on the queue.

An example of the **Poll( )** member function is given in **UI_DEVICE::Poll( )**.

UID_PENDOS::SetTimer

Syntax

```
#include <ui_pen.hpp>

void SetTimer(void);
```

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This routine resets the UID_PENDOS timer. If enough time elapses after a stroke is drawn, the system will automatically attempt to recognize any outstanding strokes as a character. This function resets the timer used to keep track of the elapsed time.

UID_PENDOS::ShowWritingWindow

Syntax

#include <ui_pen.hpp>

void ShowWritingWindow(UI_WINDOW_OBJECT *_object_);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This advanced routine adds the PenDOS window to the Window Manager.

- $object_{in}$ is a pointer to the object that is to be edited. This pointer is used to initialize *text* and also to update the object with the modified text.

UID_PENDOS::TimeExpired

Syntax

#include <ui_pen.hpp>

int TimeExpired(void);

Portability

This function is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

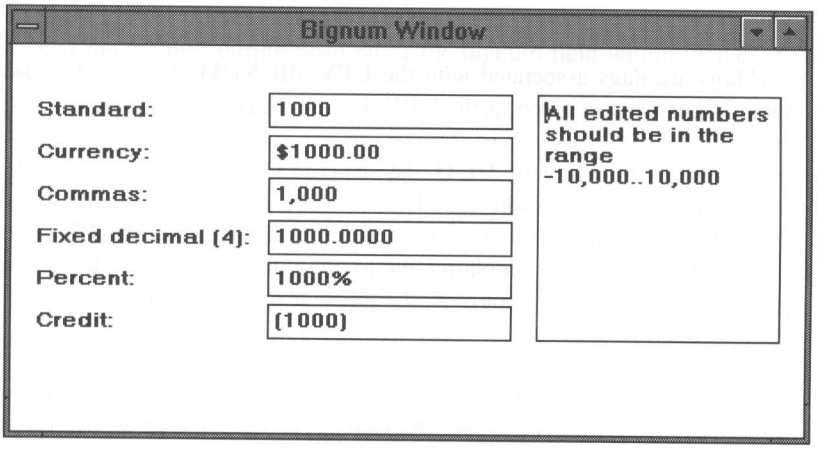
Remarks

This routine checks to see if the timer has expired since the last call to **SetTimer( )**.

- *returnValue*$_{out}$ is TRUE if the timer has expired. *returnValue* is FALSE if the timer has not expired.

CHAPTER 45 – UIW_BIGNUM

The UIW_BIGNUM class is used to display numeric information to the screen and to collect information, in numeric form, from an end user. The UIW_BIGNUM object provides special formatting features (e.g., currency, credit, percent, etc.). The figure below shows a graphic implementation of a window with several variations of the UIW_BIGNUM class object:



The UIW_BIGNUM class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class UIW_BIGNUM : public UIW_STRING
{
    friend class EXPORT UIF_BIGNUM;
public:
    // Members described in UIW_BIGNUM reference chapter.
    static NMF_FLAGS rangeFlags;
    NMF_FLAGS nmFlags;
    static UI_ITEM *errorTable;

    UIW_BIGNUM(int left, int top, int width, UI_BIGNUM *value,
        const char *range = NULL, NMF_FLAGS nmFlags = NMF_NO_FLAGS,
        WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
        USER_FUNCTION userFunction = NULL);
    UI_BIGNUM *DataGet(void);
    void DataSet(UI_BIGNUM *value);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);
    virtual int Validate(int processError = TRUE);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_BIGNUM(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_BIGNUM(void);
```

```
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);

protected:
    UI_BIGNUM *number;
    char *range;
};
```

- *rangeFlags* are flags associated with the range validation of the UIW_BIGNUM class.

- *nmFlags* are flags associated with the UIW_BIGNUM class. A full description of the number flags is given in the UIW_BIGNUM constructor.

- *errorTable* is an array of UI_ITEM structures that is used as a lookup table to determine an error message based on the type of error encountered. The array is defined and assigned to this static member in the **G_BNUM.CPP** module. These messages are placed in a lookup table to allow the programmer to easily change the text associated with the error (e.g., the message could be changed to a language other than English).

- *number* is a pointer to a UI_BIGNUM that is used to manage the low-level bignum information. If the **WOF_NO_ALLOCATE_DATA** flag is set, this member is unused.

- *range* is a copy of the string range of acceptable bignum values that is passed down to the constructor.

UIW_BIGNUM::UIW_BIGNUM

Syntax

```
#include <ui_win.hpp>

UIW_BIGNUM(int left, int top, int width, UI_BIGNUM *value,
    const char *range = NULL, NMF_FLAGS nmFlags = NMF_NO_FLAGS,
    WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
    USER_FUNCTION userFunction = NULL);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The bignum constructor returns a pointer to a new UIW_BIGNUM class object.

- *left*$_{in}$ and *top*$_{in}$ is the starting position of the bignum field within its parent window.
- *width*$_{in}$ is the width of the bignum field. (The height of the bignum field is determined automatically by the UIW_BIGNUM object.)
- *value*$_{in/out}$ is a pointer to a UI_BIGNUM object.
- *range*$_{in}$ is a string that gives all the valid numeric ranges. For example, if a range of "1000..10000" were specified, the UIW_BIGNUM class object would only accept those numeric values that fell between 1,000 and 10,000. If *range* is NULL, any number (within the absolute range) is accepted. This string is copied by the UIW_BIGNUM class object.
- *nmFlags*$_{in}$ gives information on how to display and interpret the numeric information. The following flags (declared in **UI_GEN.HPP**) control the general presentation of a UIW_BIGNUM class object:

NMF_DECIMAL(decimal)—Displays the number with a decimal point at a fixed location. *decimal* is the number of decimal places to be displayed. Valid *decimal* values range from 0 to 15.	`10,000.00`
	`43.45`
	`$149.95`

NMF_CURRENCY—Displays the number with the country-specific currency symbol.	`$10,000.00`
	`DM100`
	`£195`

NMF_CREDIT—Displays the number with the '(' and ')' credit symbols whenever the number is negative.	`(10000)` `(2350)`

NMF_COMMAS—Displays the number with commas.	`$10,000.00`
	`45,000`
	`1,195`

NMF_NO_FLAGS—Does not associate any special flags with the number object. This flag should not be used in conjunction with any other NMF flag. This is the default argument.

```
10000
4345
149950
```

NMF_PERCENT—Displays the number with a percentage symbol.

```
100%
4.5%
10%
```

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the number object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_BIGNUM class object:

  WOF_AUTO_CLEAR—Automatically clears the numeric buffer if the end-user tabs to the bignum field (from another window field) then presses a key (without first having pressed any movement or editing keys). This is a default argument.

  WOF_BORDER—Draws a single-line border around the bignum object, in graphics mode. In text mode, no border is drawn. This is a default argument.

  WOF_INVALID—Sets the initial status of the bignum field to be "invalid." Invalid numbers fit in the absolute range determined by the object type but do not fulfill all the requirements specified by the program. For example, a bignum may initially be set to 200, but the final number, edited by the end-user, must be in the range "10..100." The initial number in this example fits the absolute requirements of a UIW_BIGNUM class object but does not fit into the specified range.

  WOF_JUSTIFY_CENTER—Center-justifies the numeric information associated with the number object.

  WOF_JUSTIFY_RIGHT—Right-justifies the numeric information associated with the number object.

  WOF_NO_ALLOCATE_DATA—Prevents the bignum object from allocating a numeric value to store the numeric information. If this flag is set, the programmer must allocate the bignum (passed as the *value* parameter) that is used by the number object.

  WOF_NO_FLAGS—Does not associate any special window flags with the bignum object. Setting this flag left-justifies the numeric information. This flag should not be used in conjunction with any other WOF flag.

WOF_NON_SELECTABLE—Prevents the bignum object from being selected. If this flag is set, the user will not be able to edit the bignum information.

WOF_UNANSWERED—Sets the initial status of the bignum field to be "unanswered." An unanswered bignum field is displayed as blank space on the screen.

WOF_VIEW_ONLY—Prevents the bignum object from being edited. However, the bignum object may become current.

- *userFunction*$_{in}$ is a programmer defined function that is called whenever:

  1—the user moves onto the field (i.e., S_CURRENT message),

  2—the field is selected with the mouse or the <ENTER> key is pressed (i.e., L_SELECT message), or

  3—the user moves to a different field in the window or to a different window (i.e., S_NON_CURRENT).

  If a user function is associated with the object, then Validate must be called from within the *userFunction* function if range checking is desired. The following arguments are passed to *userFunction*:

  returnValue should be 0 if the bignum is valid. Otherwise, the programmer should call the error system with an appropriate error message and return -1.

  object$_{in}$—A pointer to the UIW_BIGNUM object or the class object derived from the UIW_BIGNUM object base class. This argument must be typecast by the programmer.

  event$_{in}$—A run-time message passed to the UIW_BIGNUM object.

  ccode$_{in}$—The logical or system code that caused the user function to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

  L_SELECT—The <ENTER> key was pressed.

  S_CURRENT—The bignum object is about to be edited. This code is sent before any editing operations are permitted.

S_NON_CURRENT—A different field or window has been selected. This code is sent after editing operations have been performed.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window and add it to the window manager.
    UI_BIGNUM value(0);
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample numbers ")
        + new UIW_PROMPT(2, 1, "Standard:")
        + new UIW_BIGNUM(12, 1, 20, &value, "0..10000")
        + new UIW_PROMPT(2, 2, "Currency:")
        + new UIW_BIGNUM(12, 2, 20, &value, "0..10000",
            NMF_CURRENCY | NMF_DECIMAL(2))
        + new UIW_PROMPT(2, 3, "Commas:")
        + new UIW_BIGNUM(12, 3, 20, &value, "0..10000", NMF_COMMAS);
    *windowManager + window;
    .
    .       .
    .

    // The number fields are automatically destroyed when the window
    // is destroyed.
}
```

UIW_BIGNUM::DataGet

Syntax

#include <ui_win.hpp>

UI_BIGNUM *DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function gets the current numeric information associated with the UIW_BIGNUM class object. This function returns a UI_BIGNUM pointer.

- *returnValue*$_{out}$ is a pointer to a UI_BIGNUM object.

Example

```
#include <ui_win.hpp>

EVENT_TYPE BignumUserFunction(UI_WINDOW_OBJECT *object, UI_EVENT &, EVENT_TYPE
ccode)
{
    if (ccode != S_NON_CURRENT)
        return (ccode);

    // Do specific validation.
    UI_BIGNUM *bignum = ((UIW_BIGNUM *)object)->DataGet();

    // Call the default Validate function to check for valid bignum.
    int valid = object->Validate(TRUE);

    // Call error system if the bignum is larger than maximum value.
    extern UI_BIGNUM _maxValue;
    if (valid == NMI_OK && *bignum < _maxValue)
    {
        valid = NMI_OUT_OF_RANGE;
        char bignumString[64];
        _maxValue.Export(bignumString, 64, NMF_NO_FLAGS);
        object->errorSystem->ReportError(object->windowManager, WOS_NO_STATUS,
            "The bignum must be less than %s.", bignumString);
    }

    // Return error status.
    if (valid == NMI_OK)
        return (0);
    else
        return (-1);
}

void ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    UI_BIGNUM value1, value2;
    UIW_WINDOW *window = UIW_WINDOW::Generic(0, 0, 45, 8, "National Debt");
    *window
        + new UIW_PROMPT(2, 1, "Initial value:")
        + new UIW_BIGNUM(12, 1, 20, &value1, NULL, NMF_NO_FLAGS,
            WOF_BORDER | WOF_AUTO_CLEAR, BignumUserFunction)
        + new UIW_PROMPT(2, 3, "Ending value:")
        + new UIW_BIGNUM(12, 3, 20, &value2, NULL, NMF_NO_FLAGS,
            WOF_BORDER | WOF_AUTO_CLEAR, BignumUserFunction);
    *windowManager + window;
}
```

UIW_BIGNUM::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(UI_BIGNUM **value*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function resets the current numeric information associated with the UIW_BIGNUM object or tells the class object that key flags, associated with the bignum object, have been changed.

- *value*$_{in/out}$ is a pointer to the new value. If the WOF_NO_ALLOCATE_DATA flag is set, this argument must be space, allocated by the programmer, that is not destroyed until the UIW_BIGNUM class object is destroyed. Otherwise, the information associated with this argument is copied by the UIW_BIGNUM class object. Care should be taken to only reset a value that is the same type as the original value. If this argument is NULL, no numeric information is changed, but the number field is re-displayed.

Example

```
#include <ui_win.hpp>

ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    // Manually add a number field to the window.
    UI_BIGNUM value(0);
    UIW_BIGNUM *numberField;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample numbers ")
        + new UIW_PROMPT(2, 1, "Standard: ")
        + (numberField = new UIW_BIGNUM(22, 1, 20, &value, "0..10000"));
    *windowManager + window;
    .
    .
    .

    // Reset the numeric information for the field.
    value->Import(100);
    numberField->DataSet(&value);
    .
    .
    .

}
```

UIW_BIGNUM::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine is used to send run-time information to a bignum object. It is declared virtual so that any derived bignum class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the bignum object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified bignum object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  L_SELECT—This message is received when the Enter is pressed while the bignum object is current. When a bignum field receives this message, it calls the programmer-specified user function if one exists. The programmer will get a pointer to the UIW_BIGNUM class, and the value for the user function *ccode* will be L_SELECT.

  S_CURRENT—This message is sent when a bignum field becomes current. When a bignum field receives this message, it calls the programmer-specified user function if one exists. The programmer will get a pointer to the UIW_BIGNUM class, and the value for the user function *ccode* will be S_CURRENT.

  S_NON_CURRENT—This message is received when the user moves from a bignum field to another field. It calls the programmer-specified user function,

which is described in the UIW_BIGNUM constructor. The value for *ccode* will be S_NON_CURRENT.

All other events are passed by **Event( )** to **UIW_STRING::Event( )** for processing.

UIW_BIGNUM::Information

Syntax

#include <ui_win.hpp>

void *Information(INFO_REQUEST *request*, void **data*, OBJECTID *objectID*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object. **Information( )** does not process any messages; it simply passes them to **UIW_STRING::Information( )**.

UIW_BIGNUM::Validate

Syntax

#include <ui_win.hpp>

virtual int Validate(int *processError* = TRUE);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to validate bignum objects. When a bignum object receives the S_CURRENT or S_NON_CURRENT messages, it will call **Validate( )** (only if *userFunction* is NULL) to check to see if the bignum value entered was valid.

- *returnValue*$_{out}$ is 0 if the validation was positive or non-zero in the event of an error.

  NMI_OUT_OF_RANGE—The bignum was not within the valid range for bignum values or was not within the specified *range*.

  NMI_OK—The bignum was entered in a correct format and within the valid range.

- *processError*$_{in}$ determines whether **Validate( )** should call **UI_ERROR_SYSTEM::-ReportError( )** if an error occurs. If *processError* is TRUE, **ReportError( )** is called.

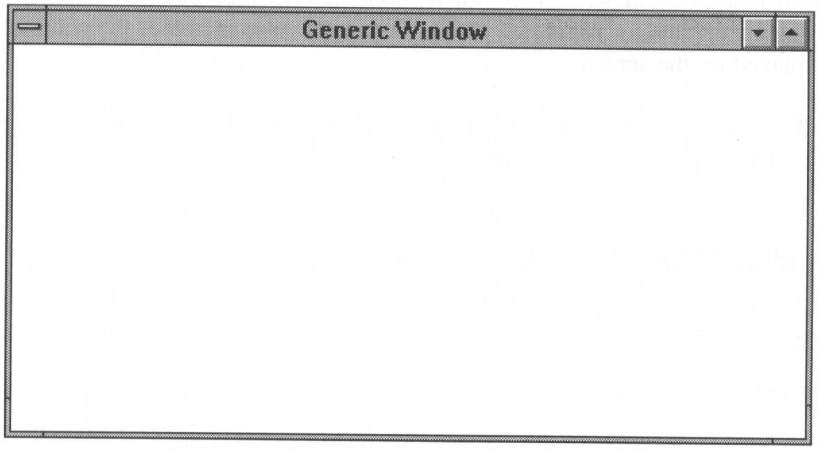
Example

```
#include <ui_win.hpp>

EVENT_TYPE BignumUserFunction(UI_WINDOW_OBJECT *object, UI_EVENT &,
    EVENT_TYPE ccode)
{
    if (ccode != S_NON_CURRENT)
        return (ccode);

    // Do specific validation.
    UI_BIGNUM *bignum = ((UIW_BIGNUM *)object)->DataGet();

    // Call the default Validate function to check for valid bignum.
    int valid = object->Validate(TRUE);

    // Call error system if the bignum is larger than maximum value.
    extern UI_BIGNUM _maxValue;
    if (valid == NMI_OK && *bignum < _maxValue)
    {
        valid = NMI_OUT_OF_RANGE;
        char bignumString[64];
        _maxValue.Export(bignumString, 64, NMF_NO_FLAGS);
        object->errorSystem->ReportError(object->windowManager, WOS_NO_STATUS,
            "The bignum must be less than %s.", bignumString);
    }
    // Return error status.
    if (valid == NMI_OK)
        return (0);
    else
        return (-1);
}

void ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    UIW_WINDOW *window = UIW_WINDOW::Generic(0, 0, 45, 8, "National Debt");
    *window
        + new UIW_PROMPT(2, 1, "Initial value:")
```

```
        + new UIW_BIGNUM(12, 1, 20, &UI_BIGNUM(), NULL, NMF_NO_FLAGS,
            WOF_BORDER | WOF_AUTO_CLEAR, BignumUserFunction)
        + new UIW_PROMPT(2, 3, "Ending value:")
        + new UIW_BIGNUM(12, 3, 20, &UI_BIGNUM(), NULL, NMF_NO_FLAGS,
            WOF_BORDER | WOF_AUTO_CLEAR, BignumUserFunction);
    *windowManager + window;
}
```

CHAPTER 46 – UIW_BORDER

The UIW_BORDER class is used to draw a border around a window in graphics modes only. The figure below shows a graphic implementation of a window which contains a UIW_BORDER class object (the outer-most region of the window):



The UIW_BORDER class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_BORDER : public UI_WINDOW_OBJECT
{
    friend class EXPORT UIF_BORDER;
public:
    // Members described in UIW_BORDER reference chapter.
    static int width;
    BDF_FLAGS bdFlags;

    UIW_BORDER(BDF_FLAGS bdFlags = BDF_NO_FLAGS);
    int DataGet(void);
    void DataSet(int width);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    virtual ~UIW_BORDER(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    UIW_BORDER(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
protected:
    // Members described in UIW_BORDER reference chapter.
    virtual EVENT_TYPE DrawItem(const UI_EVENT &event, EVENT_TYPE ccode);
};
```

- *UIF_BORDER* is used with Zinc Designer only. Programmers should not use this part of the class.

- *width* is the default width used when an application is running in graphics mode. The pre-defined value for *width* is 4 pixels. A greater value increases the width of the border displayed on the screen.

- *bdFlags* are flags associated with the UIW_BORDER class. A full description of the border flags is given in the UIW_BORDER constructor.

UIW_BORDER::UIW_BORDER

Syntax

#include <ui_win.hpp>

UIW_BORDER(BDF_FLAGS *bdFlags* = BDF_NO_FLAGS);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_BORDER object. The border object always occupies the outer-most space available in the parent window. To ensure that the border is drawn around the whole window, it must be created as the window's first object. The following example shows the correct and incorrect order of border creation:

```
1)  // CORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1")
        .
```

```
        .
        .
2)  // INCORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1")
        + new UIW_BORDER
        .
        .
        .
```

The UIW_BORDER class should not be confused with the WOF_BORDER flag. The UIW_BORDER class is attached as an object to a parent window and draws a shaded region around the parent window. The WOF_BORDER flag is not an object and only draws a 1 pixel region around the object.

NOTE: When a UIW_BORDER is added to a window, it is put into the window's support list. For more information regarding the *support* list, see "Chapter 71—UIW_WINDOW" of this manual.

- $bdFlags_{in}$ gives information on how to display the border. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of a UIW_BORDER object:

  BDF_NO_FLAGS—Does not associate any special flags with the UIW_BORDER class object. This is the default flag.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a new window and attach it to the window manager.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1");
    *windowManager + window;

    // The border will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_BORDER::DataGet

Syntax

#include <ui_win.hpp>

int DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to return the border width associated with the border object.

- *returnValue*$_{out}$ is the width of the border.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_BORDER *border)
{
    int width = border->DataGet();
    .
    .
    .

}
```

UIW_BORDER::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(int *width*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to set the border width associated with the border object. Since *UIW_BORDER::width* is static, border objects created after a call to **DataSet( )** will all be made with the new border size.

- $width_{in}$ is the new width of the border.

Example

```
#include <ui_win.hpp>

ExampleFunction1(UIW_BORDER *border)
{
    .
    .
    .

    int newWidth = 6;
    border->DataSet(newWidth);
}
```

UIW_BORDER::DrawItem

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE DrawItem(const UI_EVENT &*event*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual advanced routine is used to draw a border object on the screen. This function is called only if the object's woStatus has the WOS_OWNERDRAW status set.

- *returnValue*$_{out}$ is a response based on the success of the function call. If successful, the border object returns a non-zero value. If the object was not drawn, 0 is returned.

- *event*$_{in}$ contains a run-time message for the specified border object. The border is drawn according to the type of event. The following logical events are handled by the **DrawItem( )** routine:

  S_CURRENT, **S_NON_CURRENT**, **S_DISPLAY_ACTIVE** and **S_DISPLAY_INACTIVE**—These messages cause the border to be redisplayed. If S_CURRENT or S_NON_CURRENT are passed, the border will always be updated. If S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE are passed the border will only be updated if *event.region* overlaps the border region.

  WM_DRAWITEM—A message that causes the border to be redrawn. This message is specific to Windows and OS/2.

  Expose—A message that causes the border to be redrawn. This message is specific to Motif.

- *ccode*$_{in}$ contains the logical interpretation of *event*.

UIW_BORDER::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a border object. It is declared virtual so that any derived border class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the border object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified border object. The type of operation depends on the interpreted value for the event. The following logical events are handled by **Event( )**:

  L_BEGIN_SELECT—If the parent window can be sized, this message causes the S_SIZE message to be sent to the Window Manager via the Event Manager. The type of size operation is determined by the overlap of *event.position* within the border.

  L_VIEW—Changes the mouse pointer (if any) shown on the screen to reflect the way in which the window may be sized. For instance, if the user positions the mouse on the top part of a window's border the mouse pointer is changed to show '↕'.

  S_CREATE—Causes the border to change its size according to the size of its parent object. The border always occupies the outermost region of its parent object. The border is not shown until a display message (S_DISPLAY_-INACTIVE or S_DISPLAY_ACTIVE) is sent to the border object.

  S_DISPLAY_ACTIVE and S_DISPLAY_INACTIVE—Cause the border to be re-drawn if *event.region* overlaps any part of the border. In graphics mode, the border object is displayed as a shaded rectangle. In text mode, a shadow border is drawn.

NOTE: The S_CURRENT message is not handled by UIW_BORDER since it cannot be made current (i.e., the border class automatically sets the WOAF_NON_CUR-RENT advanced window flag).

All other events are passed by **Event( )** to **UI_WINDOW_OBJECT::Event( )** for processing.

UIW_BORDER::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

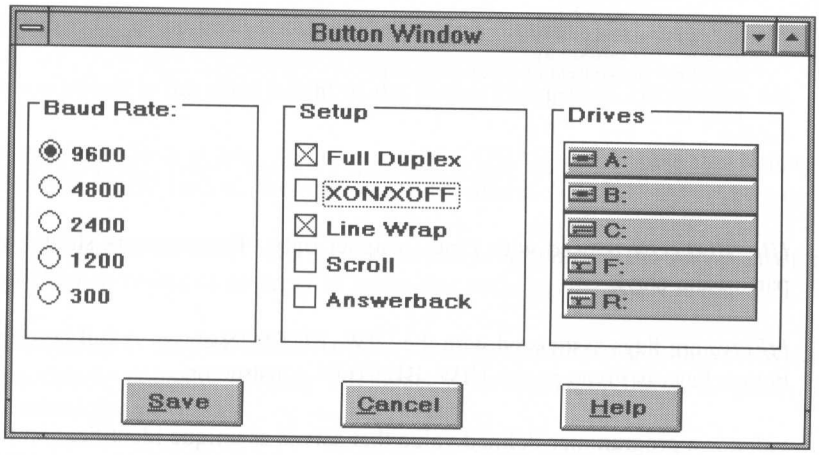
Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object. It is declared virtual so that any derived border class can override its default operation.

- *returnValue*$_{out}$ contains the information to be returned. If the information request did not require a return value, this value is *data*.
- *request*$_{in}$ contains the operation to be performed on the border object. Since **Information( )** passes the requests directly to UI_WINDOW_OBJECT, a more complete description is given in **UI_WINDOW_OBJECT::Information( )**.
- *data*$_{in}$ is a pointer to the data required by the *request* being sent. If *request* does not require any data, this value is simply returned.
- *objectID*$_{in}$ is the identification of the object that will process *request*. Since **Information( )** does not directly process *request* messages, *objectID* must be set to ID_WINDOW_OBJECT.

CHAPTER 47 – UIW_BUTTON

The UIW_BUTTON class is used to display and select options associated with a window. For example, the UIW_MINIMIZE_BUTTON, UIW_MAXIMIZE_BUTTON and UIW_-SYSTEM_BUTTON are all classes derived from the UIW_BUTTON class. These derived classes allow the user to minimize, maximize or perform general operations (e.g., size, move) on a window. The figure below shows the graphical implementation of UIW_BUTTON objects (i.e., the maximize button, minimize button, system button, radio buttons, check boxes and bitmap buttons):



The UIW_BUTTON class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_BUTTON : public UI_WINDOW_OBJECT
{
    friend class EXPORT UIF_BUTTON;
public:
    // Members described in UIW_BUTTON reference chapter.
    BTF_FLAGS btFlags;
    EVENT_TYPE value;

    UIW_BUTTON(int left, int top, int width, char *text,
        BTF_FLAGS btFlags = BTF_NO_TOGGLE | BTF_AUTO_SIZE,
        WOF_FLAGS woFlags = WOF_JUSTIFY_CENTER,
        USER_FUNCTION userFunction = NULL, EVENT_TYPE value = 0,
        char *bitmapName = NULL);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    char *DataGet(int stripText = FALSE);
    void DataSet(char *text);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);
    static EVENT_TYPE Message(UI_WINDOW_OBJECT *object, UI_EVENT &event,
        EVENT_TYPE ccode);
```

```
    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_BUTTON(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    static UI_WINDOW_OBJECT *New(const char *name,
        UI_STORAGE *file, UI_STORAGE_OBJECT *object);
    virtual ~UIW_BUTTON(void);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);

protected:
    // Members described in UIW_BUTTON reference chapter.
    BTS_STATUS btStatus;
    int depth;
    char *text;
    char *bitmapName;
    int bitmapWidth;
    int bitmapHeight;
    UCHAR *bitmapArray;
    HBITMAP colorBitmap, monoBitmap;
#if defined(ZIL_MOTIF)
    Pixmap pixmap;
#endif

    virtual EVENT_TYPE DrawItem(const UI_EVENT &event, EVENT_TYPE ccode);
};
```

- *UIF_BUTTON* is used with Zinc Designer only. Programmers should not use this part of the class.

- *btFlags* are flags associated with the UIW_BUTTON class. A full description of the button flags is given in the UIW_BUTTON constructor.

- *value* is a programmer-defined value that serves as unique identification for a button. For example, the programmer could associate the value 0 with an "ok" button and a value of 1 with a "cancel" button. This allows the programmer to define one user-function that looks at the button values, instead of several user-functions that are tied to each button object. If the BTF_SEND_MESSAGE flag is set, *value* must be an event type.

- *btStatus* are flags that indicate the current status of a button. The following internal status flags may be set for a UIW_BUTTON object.

  BTS_DEPRESSED—The button is in a depressed state.

  BTS_NO_STATUS—The button is in a normal state.

- *depth* determines the 3D or shadow appearance of the button. The following values are valid depths:

  0—the button is shown as depressed.

1—the button is shown in a normal 3D state. In text mode, the button is shown as flat (i.e., no 3D look).

2—In text mode, the button will be displayed with a shadow.

- *text* is the button text as it appears on the screen (including all trailing and leading spaces or check marks, etc.).
- *bitmapName* is a character string containing the bitmap's name. *bitmapName* is used only if the bitmap is to be read from a Windows **.RES** file or a Zinc **.DAT** file.
- *bitmapWidth* is the pixel width of the button's bitmap.
- *bitmapHeight* is the pixel height of the button's bitmap.
- *bitmapArray* is a pointer to an array of UCHAR composing the bitmap to be displayed on the button. In Windows, this value is NULL.
- *colorBitmap* is an HBITMAP structure that is specific to the native environment. *colorBitmap* points to an array of bytes that make up the XOR mask.
- *monoBitmap* is an HBITMAP structure that is specific to the native environment. *monoBitmap* points to an array of characters that make up the AND mask. This is the monochrome mask.
- *pixmap* is a pointer to the image to be displayed on the button. If the button is to contain both text and an image, then *pixmap* will contain the image and the text. This member is available in Motif only.

UIW_BUTTON::UIW_BUTTON

Syntax

```
#include <ui_win.hpp>

UIW_BUTTON(int left, int top, int width, char *text,
    BTF_FLAGS btFlags = BTF_NO_TOGGLE | BTF_AUTO_SIZE,
    WOF_FLAGS woFlags = WOF_JUSTIFY_CENTER,
    USER_FUNCTION userFunction = NULL, EVENT_TYPE value = 0,
    char *bitmapName = NULL);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_BUTTON class object.

- $left_{in}$ and top_{in} are the starting position of the button within its parent window.

- $width_{in}$ is the width of the button. (The height of the button is determined automatically by the UIW_BUTTON class object.)

- $text_{in}$ is a pointer to the text information associated with the button. The text information is copied into a buffer allocated by the UIW_BUTTON class object. A hotkey may be specified by inserting the '&' character into the string before the desired hotkey character. For example, if the string "HELLO" is to be displayed and 'E' is to be the hotkey, the string should be entered as "H&ELLO." The '&' will not be displayed, but will cause the hotkey character to be highlighted in text mode or underlined in graphics mode.

- $btFlags_{in}$ gives information on how to display the button information. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of a UIW_BUTTON class object:

  BTF_AUTO_SIZE—Automatically computes the run-time height of the button. If the application is running in text mode, the height is set to 1. If the application is running in graphics mode, the button is approximately 120% of the default cell height. This flag is a default.

  BTF_CHECK_BOX—Creates a check box that can be toggled when selected. The WOS_SELECTED flag is set when the button is selected. In graphics mode, a square box is drawn that is marked with an 'X' when selected. In text mode the check box is represented by '[ ]' when it is not selected and '[X]' when it is selected.

  BTF_DOUBLE_CLICK—Completes the button action when the button has been selected twice within a period of time specified by *UI_WINDOW_OBJECT::doubleClickRate*. If this flag is set, the user function will be called with a ccode of L_SELECT on the first click and L_DOUBLE_CLICK on the

second click. This allows different actions to be performed on a single- and double-click.

BTF_DOWN_CLICK—Completes the button action on a button down-click, rather than on a down-click and release action.

BTF_NO_FLAGS—Does not associate any special flags with the UIW_-BUTTON class object. In this case the button requires a down and up click from the mouse to complete an action.

BTF_NO_TOGGLE—Does not toggle the button's WOS_SELECTED status flag. If this flag is set, the WOS_SELECTED window object status flag is not set when the button is selected. This is a default flag.

BTF_NO_3D—Causes the button to be displayed without shadowing.

BTF_RADIO_BUTTON—Causes the button to appear and function as a radio button. In graphics mode, a graphical radio button is drawn, while in text mode, it appears as '(•)' when selected or '( )' when not selected. All of the radio buttons in a group, list box or window are considered to be members of the same group. Only one radio button in a group may be selected at any one time.

NOTE: To have multiple radio button groups on the same window use the UIW_GROUP object.

BTF_REPEAT—Causes the button to be re-selected (i.e., the *userFunction* is called) if it remains selected for a period of time greater than that specified by *UI_WINDOW_OBJECT::repeatRate*.

BTF_SEND_MESSAGE—Causes an event to be created and put on the Event Manager queue when the button is selected. The button's *value* is copied into the *event.type* field. Any temporary windows are removed from the display when this message is sent.

BTF_STATIC_BITMAPARRAY—Causes the bitmap array that is used for the image on a bitmapped button to not be deleted. By default, when a bitmapped button is created, the bitmap is converted to a native storage structure and the bitmap is deleted. If the bitmap should not be deleted after this conversion is performed (e.g., if the same bitmap image is to be used for multiple objects), then the BTF_STATIC_BITMAPARRAY flag should be set.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the button object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_BUTTON class object:

  WOF_BORDER—Draws a single-line border around the object, in graphics mode. In text mode, the border is displayed as a shadow on the button. This flag is set by default.

  WOF_JUSTIFY_CENTER—Center-justifies the *string* information associated with the button object. This flag is set by default.

  WOF_JUSTIFY_RIGHT—Right-justifies the *string* information associated with the button object.

  WOF_NO_ALLOCATE_DATA—Prevents the button object from allocating a text buffer to store the text information. If this flag is set, the programmer must allocate the text buffer (passed as the *text* parameter) that is used by the button object.

  WOF_NO_FLAGS—Does not associate any special flags with the button object. In this case, the button's string information will be left-justified. This flag should not be used in conjunction with any other WOF flag.

  WOF_NON_SELECTABLE—Prevents the button object from being selected. If this flag is set, the user will be able to see, but not select, the button.

- *userFunction*$_{in}$ is a programmer-defined function that is called whenever::

  1—the user moves onto the button field (i.e., S_CURRENT message),

  2—a button is selected with the mouse (either on a single-click, or a double-click if the BTF_DOUBLE_CLICK flag is set) or the <ENTER> key is pressed (i.e., L_SELECT or L_DOUBLE_CLICK message), or

  3—the user moves to a different field in the window or a different window on the screen (i.e., S_NON_CURRENT message).

The following parameters are passed to *userFunction* when the button is selected:

object$_{in}$ is a pointer to the UIW_BUTTON class object or to the class object derived from the UIW_BUTTON object base class. Since this argument is a UI_WINDOW_OBJECT pointer, it must be typecast by the programmer.

event$_{in}$ is a reference pointer to a copy of the event used to reach the programmer-defined function. Since this argument is a copy of the original event, it may be changed by the programmer.

ccode$_{in}$ is the logical event that the button object interpreted from *event.type*.

- *value*$_{in}$ is a unique value that a programmer can associate with the button. For example, the programmer could associate the value 0 with an "ok" button and a value of 1 with a "cancel" button. This allows the programmer to define one user-function that looks at the button values, instead of several user-functions that are tied to each button object. If the BTF_SEND_MESSAGE flag is set, *value* must be an event type.

- *bitmapName*$_{in}$ is the name of the bitmap to be displayed on the button. This name will be used when searching for the bitmap in the **.RES** and **.DAT** files.

Example

```
#include <ui_win.hpp>

const USHORT FILE_EXIT   = 1;
const USHORT FILE_CANCEL     = 2;
const USHORT FILE_HELP   = 3;

static EVENT_TYPE FileControl(UI_WINDOW_OBJECT *data, UI_EVENT &event,
EVENT_TYPE ccode)
{
    // Switch on the event value (which is the button value).
    UIW_BUTTON *button = (UIW_BUTTON *)data;
    switch (button->value)
    {
    case FILE_EXIT:
        // Exit the application.
        event.type = L_EXIT;
        button->eventManager->Put(event);
        break;
    case FILE_CANCEL:
        // Close the file window.
        event.type = S_CLOSE;
        button->eventManager->Put(event);
        break;

    case FILE_HELP:
        // Get help on the file program.
        _helpSystem->DisplayHelp(button->windowManager, HELP_FILE);
        break;
    }
    return ccode;
}
```

```
ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window with buttons.
    UIW_WINDOW *window = UIW_WINDOW::Generic(0, 0, 50, 10, "Window");
    *window
        + new UIW_BUTTON(7, 6, 10, "E~xit", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
          WOF_BORDER | WOF_JUSTIFY_CENTER, FileControl, FILE_EXIT)
        + new UIW_BUTTON(20, 6, 10, "~Cancel", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
          WOF_BORDER | WOF_JUSTIFY_CENTER, FileControl, FILE_CANCEL)
        + new UIW_BUTTON(33, 6, 10, "~Help", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
          WOF_BORDER | WOF_JUSTIFY_CENTER, FileControl, FILE_HELP,
          "HELP_BITMAP");
    *windowManager + window;
    .
    .
    .
    // The buttons will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_BUTTON::DataGet

Syntax

#include <ui_win.hpp>

char *DataGet(int *stripText* = FALSE);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to return the text associated with the button object.

- *returnValue*$_{out}$ is a pointer to the string information associated with the button.
- *stripText*$_{in}$ is used to take out the leading and trailing spaces and the '&' character.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_BUTTON *button)
```

```
{
    char *text = button->DataGet();
    .
    .
    .
}
```

UIW_BUTTON::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(char *_text_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to set the text information associated with the button.

- $text_{in}$ is a pointer to the new text information to be displayed on the button. If the button object has the WOF_NO_ALLOCATE_DATA flag set, then **DataSet( )** will cause *text* to be used as the text buffer.

Example

```
#include <ui_win.hpp>

ExampleFunction1(UIW_BUTTON *button)
{
    .
    .
    .
    button->DataSet("&Close");
}
```

UIW_BUTTON::DrawItem

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE DrawItem(const UI_EVENT &*event*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual advanced routine is used to draw a button object on the screen. This function is called only if the object's woStatus has the WOS_OWNERDRAW status set.

- $returnValue_{out}$ is a response based on the success of the function call. If successful, the function returns a non-zero value. If the object was not drawn, 0 is returned.

- $event_{in}$ contains a run-time message for the specified button object. The button is drawn according to the type of event. The following logical events are handled by the **DrawItem( )** routine:

  S_CURRENT, S_NON_CURRENT, S_DISPLAY_ACTIVE and **S_-DISPLAY_INACTIVE**—These messages cause the button to be redisplayed. If S_CURRENT or S_NON_CURRENT are passed, the button will always be updated. If S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE are passed the button will only be updated if *event.region* overlaps the button region.

  WM_DRAWITEM—A message that causes the button to be redrawn. This message is specific to Windows and OS/2.

  Expose—A message that causes the button to be redrawn. This message is specific to Motif.

- $ccode_{in}$ contains the logical interpretation of *event*.

UIW_BUTTON::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT *&event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a button object. It is declared virtual so that any derived button class can override its default operation.

- $returnValue_{out}$ is a response based on the type of *event*. If successful, the button object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- $event_{in}$ contains a run-time message for the specified button object. The type of operation depends on the interpreted value for the event. The following logical events are handled by the **Event( )** routine:

  L_BEGIN_SELECT—If the BTF_DOWN_CLICK button flag is set and *event.position* overlaps the button region, this message causes the programmer defined *userFunction* to be called. Otherwise, the button is shown in a depressed state.

  L_CONTINUE_SELECT—Continues the depressed screen presentation of the button unless *event.position* of the event no longer overlaps the button. If the position does not overlap, the button is shown in a normal state.

  L_END_SELECT—Causes the programmer defined *userFunction* to be called if no special BTF selection flags are set and if *event.position* continues to overlap the button region.

L_SELECT—Causes the programmer defined *userFunction* to be called and sets the WOS_SELECTED *woStatus* flag.

S_CREATE and S_SIZE—Determines the size and position of the button within its parent object. The button is not shown until a display message (S_CURRENT, S_DISPLAY_INACTIVE, S_DISPLAY_ACTIVE) is sent to the button object.

S_CURRENT, S_DISPLAY_ACTIVE and S_DISPLAY_INACTIVE—If *event.region* overlaps the button region, these messages cause the button and text information to be displayed to the screen.

S_NON_CURRENT—Makes sure that the button is no longer in a depressed state.

All other events are passed by **Event( )** to **UI_WINDOW_OBJECT::Event( )** for processing.

UIW_BUTTON::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_BUTTON, the request is passed on to UI_WINDOW_OBJECT. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

  CLEAR_FLAGS—Clears the BTF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_BUTTON or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  CLEAR_STATUS—Clears the BTS_STATUS flags, specified by *data*, associated with an object if the value in *objectID* is ID_BUTTON or the WOS_STATUS if *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIS_STATUS.

  COPY_TEXT—Returns the *text* associated with the button. If this message is used, *data* is the address of a buffer where the button's text will be copied. This buffer must be large enough to contain all of the characters associated with the button and the terminating NULL character.

  GET_BITMAP_ARRAY—Returns a pointer to the button's bitmap array. If a bitmap does not exist, NULL is returned. If this message is sent, *data* must be a pointer to an array of UCHAR large enough to hold the bitmap data.

  GET_BITMAP_HEIGHT—Returns a pointer to the button's bitmap height. If this message is sent, *data* must be a pointer to an unsigned integer.

  GET_BITMAP_WIDTH—Returns a pointer to the button's bitmap width. If this message is sent, *data* must be a pointer to an unsigned integer.

  GET_FLAGS—Returns the BTF_FLAGS associated with an object if the value in *objectID* is ID_BUTTON or the woFlags if *objectID* is ID_WINDOW_-OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

GET_STATUS—Returns the BTS_STATUS flags associated with an object if the value in *objectID* is ID_BUTTON or the WOS_STATUS if *objectID* is ID_WINDOW_OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIS_STATUS.

GET_TEXT—Returns the *text* associated with the button. If this message is used, *data* must be the address of a character pointer that will point to the button's text string.

GET_VALUE—Returns the *value* associated with the button. If this message is sent, *data* must be a pointer to a programmer defined unsigned short where the button's value will be copied.

SET_BITMAP_ARRAY—Sets the bitmap array associated with a bitmapped button. If this message is sent, *data* must be a pointer to an array of UCHAR containing the button's new bitmap.

SET_BITMAP_HEIGHT—Sets the button's bitmap height. If this message is sent, *data* must be a pointer to an unsigned integer.

SET_BITMAP_WIDTH—Sets the button's bitmap width. If this message is sent, *data* must be a pointer to an unsigned integer.

SET_FLAGS—Sets the BTF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_BUTTON or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

SET_STATUS—Sets the BTS_STATUS flags, specified by *data*, associated with an object if the value in *objectID* is ID_BUTTON or the WOS_STATUS if *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIS_STATUS.

SET_TEXT—Sets the *text* associated with the button. If this message is sent, *data* must be a character pointer to the button's new text.

SET_VALUE—Sets the *value* associated with the button. If this message is sent, *data* must be a pointer to a programmer defined unsigned short that contains the button's new value.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

- $objectID_{in}$ is the identification code of the object to receive the request. If *objectID* is left blank or unrecognized, then the request will be interpreted by UI_WINDOW_-OBJECT.

Example

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

ExampleFunction()
{
        char string[30];
        button->Information(COPY_TEXT, string, ID_BUTTON);
        .
        .
        .

        button1->Information(SET_TEXT, "&New", ID_BUTTON);
        button2->Information(SET_TEXT, "E&xit", ID_BUTTON);
        .
        .
        .

}
```

UIW_BUTTON::Message

Syntax

#include <ui_win.hpp>

EVENT_TYPE Message(UI_WINDOW_OBJECT **object*, UI_EVENT &*event*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is a library-supplied user function that is used to put a message on the event

queue. If the BTF_SEND_MESSAGE flag is set, the button's value will be put into an event structure and then put on the event queue. This function is not called directly by the programmer; rather, it is called by the library.

- *returnValue*$_{out}$ is the *ccode* argument.
- *event*$_{in}$ contains the run-time message that caused the button to be selected.
- *ccode*$_{in}$ contains the logical interpretation of the event's type.

Example 1

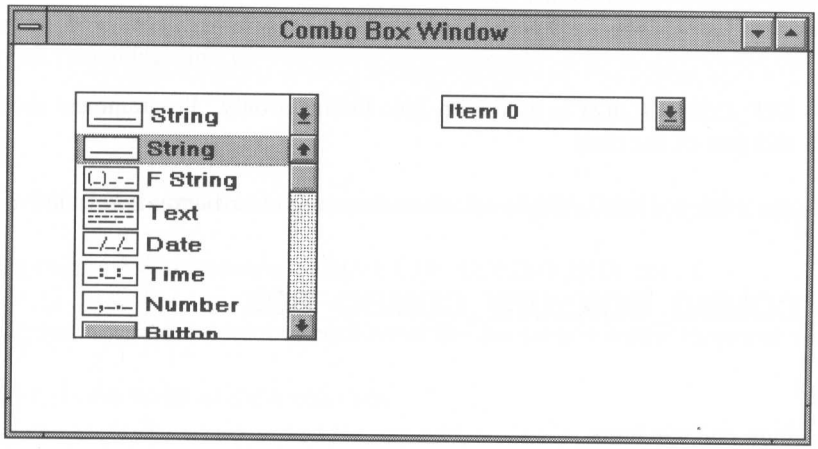
```
EVENT_TYPE UIW_BUTTON::Message(UI_WINDOW_OBJECT *object, UI_EVENT &event,
    EVENT_TYPE ccode)
{
    event.type = ((UIW_BUTTON *)object)->value;
    event.data = object;
    object->eventManager->Put(event);
    return (ccode);
}
```

Example 2

```
ExampleFunction()
{
    UIW_BUTTON = new UIW_BUTTON(1, 5, 20, "E&xit", BTF_AUTO_SIZE |
        BTF_NO_TOGGLE | BTF_SEND_MESSAGE, WOF_JUSTIFY_CENTER, NULL, L_EXIT);
    .
    .
    .

}
```

CHAPTER 48 – UIW_COMBO_BOX

The UIW_COMBO_BOX class is used to provide a list of items in a combo box object. A combo box is a combination of a default selection field, a prompt box and a scrollable list box. The prompt box (located to the right of the default selection field) contains a button that causes a scrollable list of items to appear. When one of the items is selected, it is copied into the default selection field and the list box disappears. (To remove the selection box without selecting an item, press <Esc> in DOS, OS/2 or Motif, or <Shift+F4> in Windows.) In addition to string objects, the combo box may contain other window objects such as icons and bitmapped buttons, etc. The figure below shows the graphical implementation of a UIW_COMBO_BOX object:



The UIW_COMBO_BOX class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_COMBO_BOX : public UIW_WINDOW
{
    friend class EXPORT UIF_COMBO_BOX;
public:
    // Members described in UIW_COMBO_BOX reference chapter.
    UIW_VT_LIST list;

    UIW_COMBO_BOX(int left, int top, int width, int height,
        int (*compareFunction)(void *element1, void *element2) = NULL,
        WNF_FLAGS wnFlags = WNF_NO_WRAP, WOF_FLAGS woFlags = WOF_NO_FLAGS,
        WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS);
    UIW_COMBO_BOX(int left, int top, int width, int height,
        int (*compareFunction)(void *element1, void *element2),
        WOF_FLAGS flagSetting, UI_ITEM *item);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);
```

```
    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_COMBO_BOX(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_COMBO_BOX(void);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);

    // Members described in UI_LIST reference chapter.
    virtual void Destroy(void);
    UI_WINDOW_OBJECT *Add(UI_WINDOW_OBJECT *object);
    int Count(void);
    UI_WINDOW_OBJECT *Current(void);
    UI_WINDOW_OBJECT *First(void);
    UI_WINDOW_OBJECT *Get(int index);
    int Index(UI_WINDOW_OBJECT const *element);
    UI_WINDOW_OBJECT *Last(void);
    UI_WINDOW_OBJECT *Subtract(UI_WINDOW_OBJECT *object);
    UIW_COMBO_BOX &operator+(UI_WINDOW_OBJECT *object);
    UIW_COMBO_BOX &operator-(UI_WINDOW_OBJECT *object);
};
```

- *UIF_COMBO_BOX* is used with Zinc Designer only. Programmers should not use this part of the class.

- *list* is the list that is displayed when the combo box button is selected.

UIW_COMBO_BOX::UIW_COMBO_BOX

Syntax

#include <ui_win.hpp>

UIW_COMBO_BOX(int *left*, int *top*, int *width*, int *height*,
int (**compareFunction*)(void *element1, void *element2) = NULL,
WNF_FLAGS *wnFlags* = WNF_NO_WRAP,
WOF_FLAGS *woFlags* = WOF_NO_FLAGS,
WOAF_FLAGS *woAdvancedFlags* = WOAF_NO_FLAGS)
or
UIW_COMBO_BOX(int *left*, int *top*, int *width*, int *height*,
int (**compareFunction*)(void *element1, void *element2), WOF_FLAGS *flagSetting*,
UI_ITEM **item*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

In Motif, a combo box is represented as an "Option menu" which has a different operation than a standard combo box. The "Option menu" is a non-editable object whose selection box is positioned on the screen according to the position of the current object in the list, rather than always below the combo box object. For example, if the object currently displayed in the combo box is the first object in the list, the selection box will appear at the same coordinates as the combo box object and extend downward so that all the list items are displayed (an Option menu is not scrollable). If the object currently displayed is the third item in the list, the selection box will be positioned two cells above the combo box object (to display the first two list objects) and extend downward to display all remaining objects.

Remarks

These overloaded constructors return a pointer to a new UIW_COMBO_BOX class object.

The first overloaded constructor creates a UIW_COMBO_BOX object.

- *left*$_{in}$ and *top*$_{in}$ are the starting position of the combo box within its parent window.
- *width*$_{in}$ is the width of the combo box.
- *height*$_{in}$ is the height of the combo box's associated list box.
- *compareFunction*$_{in}$ is a programmer defined function used to determine the order of combo box items. New items are placed at the end of a combo box list if this value is NULL. The following arguments are passed to *compareFunction*:

  element1$_{in}$—A pointer to the first argument to compare. This argument must be typecast by the programmer.

  element2$_{in}$—A pointer to the second argument to compare. This argument must be typecast by the programmer.

  The compare function's *returnValue* should be 0 if the two elements exactly match. If a negative value is returned, then *element1* is less than *element2*. A positive value

indicates that *element1* is greater than *element2*. For more information about the *compareFunction* argument see the UI_LIST reference chapter.

- *wnFlags*$_{in}$ gives information on how to display the combo box information. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of a UIW_COMBO_BOX class object:

  WNF_NO_FLAGS—Does not associate any special WNF_FLAGS with the UIW_COMBO_BOX class object. This flag should not be used in conjunction with any other WNF flag.

  WNF_NO_WRAP—Does not allow the user to scroll to the top of the combo box list by cursoring down at the bottom of the list. This flag is used by default.

  WNF_AUTO_SORT—Assigns a compare function to alphabetize the strings within the list. If this flag is used, *compareFunction* should be NULL.

  WNF_BITMAP_CHILDREN—Should be set when items other than strings are added to the combo box.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the combo box object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_COMBO_BOX class object:

  WOF_BORDER—Draws a single-line border around the combo box object, in graphics mode. In text mode, no border is drawn.

  WOF_JUSTIFY_CENTER—Center-justifies the *string* information associated with the combo box's string field.

  WOF_JUSTIFY_RIGHT—Right-justifies the *string* information associated with the combo box's string field.

  WOF_NO_FLAGS—Does not associate any special flags with the combo box object. In this case, the combo box's string information will be left-justified. This flag should not be used in conjunction with any other WOF flag.

  WOF_NON_SELECTABLE—Prevents the combo box object from being selected. If this flag is set, the user will be able to see, but not select, the combo box.

- *woAdvancedFlags*$_{in}$ are flags (general to all window objects) that determine the advanced operation of the combo box object.

  WOAF_NO_FLAGS—Does not associate any special advanced flags with the combo box. Setting this flag allows the user to interact with the combo box in a normal fashion. This flag should not be used in conjunction with any other WOAF flag.

  WOAF_NO_DESTROY—Prevents the Window Manager from calling the combo box destructor. If this flag is set, the combo box can be removed from the screen display, but the programmer must call the associated combo box destructor.

  WOAF_NON_CURRENT—Prevents the combo box from becoming the current object. If this flag is set, users will not be able to select the object from the keyboard nor with the mouse.

The second overloaded constructor creates a UIW_COMBO_BOX object. It takes an array of UI_ITEM structures, constructs a UIW_STRING for each object, and then adds it to the combo box. This allows a programmer to add a group of strings into a combo box without having to use the **operator +** for each item.

- *left*$_{in}$ and *top*$_{in}$ are the starting position of the combo box within its parent window.

- *width*$_{in}$ is the width of the combo box.

- *height*$_{in}$ is the height of the combo box's associated list box.

- *compareFunction*$_{in}$ is a programmer defined function used to determine the order of combo box items. For a more complete explanation, see *compareFunction* under the first constructor.

- *flagSetting*$_{in}$ is the variable against which the item's flag setting is compared. If the item's *value* and this *flagSetting* match, the item will be marked as having been selected.

- *item*$_{in}$ is an array of UI_ITEM structures used to construct UIW_STRING objects and then add them to the combo box.

Example

```
#include <ui_win.hpp>

void ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    UIW_WINDOW *window = UIW_WINDOW::Generic(0, 0, 45, 8, "Window");
    *window
        + new UIW_PROMPT(2, 1, "Combo box:")
        + &(* new UIW_COMBO_BOX(20, 1, 17, 6)
            + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_VERTICAL)
            + new UIW_STRING(0, 0, 15, "Combo 1")
            + new UIW_STRING(0, 0, 15, "Combo 2")
            + new UIW_STRING(0, 0, 15, "Combo 3")
            + new UIW_STRING(0, 0, 15, "Combo 4")
            + new UIW_STRING(0, 0, 15, "Combo 5")
            + new UIW_STRING(0, 0, 15, "Combo 6")
            + new UIW_STRING(0, 0, 15, "Combo 7")
            + new UIW_STRING(0, 0, 15, "Combo 8")
            + new UIW_STRING(0, 0, 15, "Combo 9")
            + new UIW_STRING(0, 0, 15, "Combo 10"));
    *windowManager + window;
}
```

UIW_COMBO_BOX::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a combo box object. It is declared virtual so that any derived combo box class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the combo box object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified combo box object. The type of operation depends on the interpreted value for the event. The following logical event is handled by the **Event( )** routine:

  S_CREATE—Determines the size and position of the button within its parent object. The combo box is not shown until a display message (i.e., S_-CURRENT, S_DISPLAY_INACTIVE, S_DISPLAY_ACTIVE) is sent to the combo box object.

All other events are passed by **Event( )** to **UIW_WINDOW::Event( )** for processing.

UIW_COMBO_BOX::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_COMBO_BOX, the request is passed on to UIW_WINDOW. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

CLEAR_FLAGS—Clears the WNF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_WINDOW or clears the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

GET_BITMAP_ARRAY—Returns a pointer to the bitmap array associated with the item in the default selection field. If a bitmap does not exist, NULL is returned. If this message is sent, *data* must be a pointer to an array of UCHAR.

GET_FLAGS—Returns the WNF_FLAGS associated with an object if the value in *objectID* is ID_WINDOW or returns the WOF_FLAGS if *objectID* is ID_-WINDOW_OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

GET_NUMBERID_OBJECT—Returns a pointer to the object in the list whose numberID matches the numberID passed in through the *data* argument. If this message is sent, *data* must be a pointer to a programmer defined unsigned short.

GET_STRINGID_OBJECT—Returns a pointer to the object in the list whose stringID matches the stringID passed in through the *data* argument. If this message is sent, *data* must be a pointer to a string.

GET_TEXT—Returns the *text* associated with the item in the default selection field.

SET_BITMAP_ARRAY—Sets the bitmap array associated with the item in the default selection field. If this message is sent, *data* must be a pointer to an array of UCHAR containing the item's new bitmap.

SET_FLAGS—Sets the WNF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_WINDOW or sets the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

SET_TEXT—Sets the *text* associated with the item in the default selection field. If this message is sent, *data* must be a character pointer to the item's new text.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

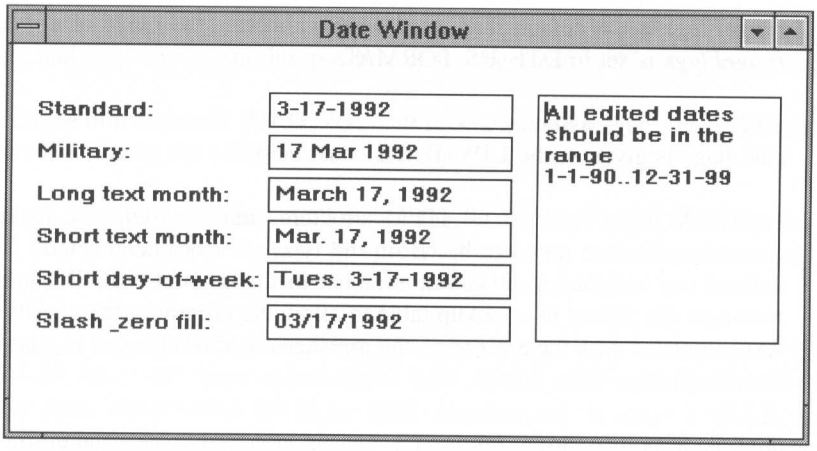
- *objectID*$_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, the request will be interpreted by UI_WINDOW_OBJECT.

Example

```
#include <ui_win.hpp>
ExampleFunction()
{
        UIW_COMBO_BOX *comboBox;
        .
        .
        .

        ushort flags;
        comboBox->Information(GET_FLAGS, &flags);
        .
        .
        .

}
```

CHAPTER 49 – UIW_DATE

The UIW_DATE class is used to display date information to the screen and to collect information, in date form, from an end user. It is not the low-level date storage object. (See "Chapter 4—UI_DATE" of this manual for information about the low-level date storage object.) The figure below shows the graphical implementation of a window with several variations of the UIW_DATE class object:



The UIW_DATE class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class UIW_DATE : public UIW_STRING
{
    friend class EXPORT UIF_DATE;
public:
    // Members described in UIW_DATE reference chapter.
    static DTF_FLAGS rangeFlags;
    DTF_FLAGS dtFlags;
    static UI_ITEM *errorTable;

    UIW_DATE(int left, int top, int width, UI_DATE *date,
        const char *range = NULL, DTF_FLAGS dtFlags = DTF_NO_FLAGS,
        WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
        USER_FUNCTION userFunction = NULL);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);
    UI_DATE *DataGet(void);
    void DataSet(UI_DATE *date);
    virtual int Validate(int processError = TRUE);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_DATE(const char *name, UI_STORAGE *file, UI_STORAGE_OBJECT *object);
    virtual ~UIW_DATE(void);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);

protected:
    // Members described in UIW_DATE reference chapter.
    UI_DATE *date;
    char *range;
};
```

- *UIF_DATE* is used only by Zinc Designer. Programmers should not use this item.
- *rangeFlags* is the default flags used when validating the range of a date. Initially, *rangeFlags* is set to DTF_US_FORMAT.
- *dtFlags* are flags associated with the UIW_DATE class. A full description of the date flags is given in the UIW_DATE constructor.
- *errorTable* is an array of UI_ITEM structures that is used as a lookup table to determine an error message based on the type of error encountered. The array is defined and assigned to this static member in the **G_DATE1.CPP** module. These messages are placed in a lookup table to allow the programmer to easily change the text associated with the error (e.g., the message could be changed to a language other than English).
- *date* is a pointer to a UI_DATE that is used to manage the low-level date information. If the **WOF_NO_ALLOCATE_DATA** flag is set, this member is unused.
- *range* is a copy of the string range of acceptable dates that is passed down to the constructor.

UIW_DATE::UIW_DATE

Syntax

```
#include <ui_win.hpp>

UIW_DATE(int left, int top, int width, UI_DATE *date, const char *range = NULL,
    DTF_FLAGS dtFlags = DTF_NO_FLAGS,
    WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
    USER_FUNCTION userFunction = NULL);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_DATE class object.

- $left_{in}$ and top_{in} are the starting position of the date field within its parent window.
- $width_{in}$ is the width of the date field. (The height of the date field is determined automatically by the UIW_DATE class object.)
- $date_{in/out}$ is a pointer to the initial date value. If the WOF_NO_ALLOCATE_DATA flag is set, this argument must be space, allocated by the programmer, that is not destroyed until the UIW_DATE class object is destroyed.
- $range_{in}$ is a string that gives all the valid date ranges. For example, if a range of "1-1-90..12-31-90" were specified, the UIW_DATE class object would only accept those dates whose values fell in the 1990 calendar year. If *range* is NULL, any date is within the absolute date range (1-1-100 through 12-31-32767). This string is always copied by the UIW_DATE class object.
- $dtFlags_{in}$ gives information on how to display and interpret the date information. The following flags (declared in **UI_GEN.HPP**) override the country dependent information supplied by all DOS based systems:

DTF_DASH—Separates each date variable with a dash, regardless of the default country date separator.

```
3-28-1990
12-04-1980
1-3-2003
```

DTF_DAY_OF_WEEK—Adds an ASCII string day-of-week value to the date.

```
Monday May 4, 1992
Friday Dec. 5, 1980
Sunday Jan. 4, 2003
```

DTF_EUROPEAN_FORMAT—Forces the date to be displayed and interpreted in the European format (i.e., *day/month/year*), regardless of the default country information.

```
28/3/1990
4 December, 1980
3 Jan., 2003
```

WOF_NO_FLAGS—Does not associate any special window object flags with the date object. Setting this flag left-justifies the date information. This flag should not be used in conjunction with any other WOF flag.

WOF_NON_SELECTABLE—Prevents the date object from being selected. If this flag is set, the user will not be able to edit the date information.

WOF_UNANSWERED—Sets the initial status of the date field to be "unanswered." An unanswered date field is displayed as blank space on the screen.

- *userFunction*$_{in}$ is a programmer defined function that is called whenever:

  1—the user moves onto the date field (i.e., S_CURRENT message.)

  2—the <ENTER> key is pressed (i.e., L_SELECT message.)

  3—the user moves to a different field in the window or a different window (i.e., S_NON_CURRENT message.)

  If no *userFunction* function is specified, **Validate( )** is automatically called. However, if a user function is given, the programmer must call **Validate( )** in order for the date to be validated. The following arguments are passed to *userFunction* when a new date is entered:

  returnValue should be 0 if the date is valid. Otherwise, the programmer should call the error system with an appropriate error message and return -1.

  object$_{in}$—A pointer to the UIW_DATE class object or the class object derived from the UIW_DATE object base class. This argument must be typecast by the programmer.

  event$_{in}$—the event that caused *userFunction* to be called.

  ccode$_{in}$—The logical or system code that caused the user function to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

  L_SELECT—The <ENTER> key was pressed.

  S_CURRENT—The date object is about to be edited. This code is sent before any editing operations are permitted.

S_NON_CURRENT—A different field or window has been selected.

Example

```
#include <ui_win.hpp>

ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    UI_DATE date; // system date
    UI_TIME time; // system time

    // Create a window with a date and time field.
    UIW_WINDOW *window = UIW_WINDOW::Generic(0, 0, 45, 8, NULL, "Window");
    *window
        + new UIW_PROMPT(2, 1, "Date..")
        + new UIW_DATE(9, 1, 20, &date, "", DTF_ALPHA_MONTH | DTF_SYSTEM)
        + new UIW_PROMPT(2, 3, "Time..")
        + new UIW_TIME(9, 3, 20, &time, "", TMF_SECONDS);
        .
        .
        .

    // The date object will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_DATE::DataGet

Syntax

#include <ui_win.hpp>

UI_DATE *DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to return the UI_DATE object associated with the UIW_DATE.

- *returnValue*$_{out}$ is a pointer to a UI_DATE containing the date information.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_DATE *dateObject)
{
    UI_DATE *date = dateObject->DataGet();
    .
    .
    .

}
```

UIW_DATE::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(const UI_DATE **date*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to set the date information associated with the date object.

- $date_{in}$ is the new date information. If the *date* is "blank" (see the Import section of "Chapter 4—UI_DATE" for information on setting a blank date field) then the UIW_DATE object will be re-displayed as blank space.

Example

```
#include <ui_win.hpp>

ExampleFunction1(UIW_DATE *date)
{
    .
    .
    .
```

```
    UI_DATE dateInfo(92, 10, 19);
    date->DataSet(&dateInfo);
}
```

UIW_DATE::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT *&event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a date object. It is declared virtual so that any derived date class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the date object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified date object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_CREATE—Ensures that the date string (shown to the screen) reflects the internal date associated with the date object.

  S_CURRENT—Calls the programmer-specified user function (if any). The function gets a pointer to the UIW_DATE object, the event and the *ccode* value S_CURRENT.

  S_NON_CURRENT—Calls the programmer-specified user function (if any). The user function gets a pointer to the UIW_DATE object, the event and the *ccode* value S_NON_CURRENT.

NOTE: This message is received when the user moves from a date field to another field.

All other events are passed by **Event( )** to **UIW_STRING::Event( )** for processing.

UIW_DATE::Information

Syntax

```
#include <ui_win.hpp>

void *Information(INFO_REQUEST request, void *data, OBJECTID objectID);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_DATE, then the request is passed on to UIW_STRING. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

  GET_FLAGS—Returns the DTF_FLAGS associated with an object if the value in *objectID* is ID_DATE, STF_FLAGS if *objectID* is ID_STRING, or WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

SET_FLAGS—Sets the DTF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_DATE, STF_FLAGS if *objectID* is ID_STRING, or WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

CLEAR_FLAGS—Clears the DTF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_DATE, STF_FLAGS if *objectID* is ID_STRING, or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

- $objectID_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, then the request will be interpreted by UIW_STRING.

Example

```
#include <ui_win.hpp>

ExampleFunction()
{
        UIW_DATE *date, *date1, *date2;
        .
        .
        .

        char string[30];
        date->Information(COPY_TEXT, string);
        .
        .
        .

        date1->Information(SET_TEXT, "12/31/91");
        date2->Information(SET_TEXT, "10/19/91");
        .
        .
        .

}
```

UIW_DATE::Validate

Syntax

#include <ui_win.hpp>

virtual int Validate(int *processError* = TRUE);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to validate date objects. When a date object receives the L_SELECT, S_CURRENT or S_NON_CURRENT messages, it will call **Validate( )** to check to see if the date entered was valid.

- *returnValue*$_{out}$ is 0 if the validation was positive or non-zero in the event of an error.

  DTI_INVALID—The date was in an invalid format.

  DTI_AMBIGUOUS—The date contained ambiguous data.

  DTI_INVALID_NAME—The date contained an invalid month name.

  DTI_VALUE_MISSING—No date value was entered.

  DTI_OUT_OF_RANGE—The date was not within the valid range for dates.

  DTI_OK—The date was entered in a correct format and within the valid range.

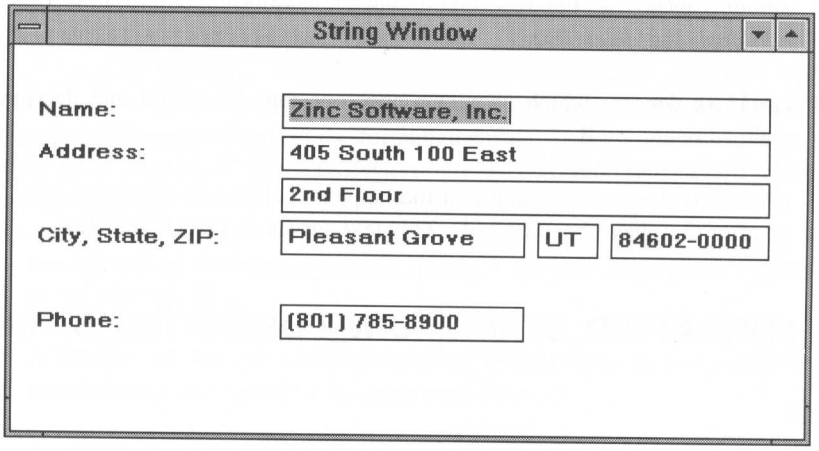
- *processError*$_{in}$ determines whether **Validate( )** should call **UI_ERROR_SYSTEM::-ReportError( )** if an error occurs. If *processError* is TRUE, ReportError() is called.

Example

```
#include <ui_win.hpp>

EVENT_TYPE DateUserFunction(UI_WINDOW_OBJECT *object, UI_EVENT &,
    EVENT_TYPE ccode)
{
    if (ccode != S_NON_CURRENT)
        return (ccode);

    // Do specific validation.
    UI_DATE currentDate;
    UI_DATE *date = ((UIW_DATE *)object)->DataGet();

    // Call the default Validate function to check for valid date.
    int valid = object->Validate(TRUE);

    // Call error system if the date entered is later than the system date.
    if (valid == DTI_OK && currentDate < *date)
    {
        valid = DTI_INVALID;
        char dateString[64];
        currentDate.Export(dateString, 64, DTF_NO_FLAGS);
        object->errorSystem->ReportError(object->windowManager, WOS_NO_STATUS,
            "The date must be before %s.", dateString);
    }

    // Return error status.
    if (valid == DTI_OK)
        return (0);
    else
        return (-1);
}
```

CHAPTER 50 – UIW_FORMATTED_STRING

The UIW_FORMATTED_STRING class is used to display string information to the screen and to collect information, in a formatted context, from an end user. The figure below shows the graphical implementation of two UIW_FORMATTED_STRING class objects:



The UIW_FORMATTED_STRING class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_FORMATTED_STRING : public UIW_STRING
{
    friend class EXPORT UIF_FORMATTED_STRING;
public:
    // Members described in UIW_FORMATTED_STRING reference chapter.
    UIW_FORMATTED_STRING(int left, int top, int width, char *compressedText,
        char *editMask, char *deleteText,
        WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
        USER_FUNCTION userFunction = NULL);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    char *DataGet(int compressedText = FALSE);
    void DataSet(char *text);
    void Export(char *destination, int expanded);
    FMI_RESULT Import(char *source);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_FORMATTED_STRING(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_FORMATTED_STRING(void);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
protected:
    // Members described in UIW_FORMATTED_STRING reference chapter.
    char *compressedText;
    char *editMask;
    char *deleteText;
};
```

- *UIF_FORMATTED_STRING* is used with Zinc Designer only. Programmers should not use this part of the class.
- *compressedText* is the raw text string as it appears before it is formatted with *editMask* and *deleteText*.
- *editMask* denotes which character positions may be edited and determines which characters are valid.
- *deleteText* shows the default text that appears before any characters have been typed or after they have been erased. This text serves as placeholder text.

UIW_FORMATTED_STRING::UIW_FORMATTED_STRING

Syntax

#include <ui_win.hpp>

UIW_FORMATTED_STRING(int *left*, int *top*, int *width*, char **compressedText*,
char **editMask*, char **deleteText*,
WOF_FLAGS *woFlags* = WOF_BORDER | WOF_AUTO_CLEAR,
USER_FUNCTION *userFunction* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_FORMATTED_STRING class object.

- $left_{in}$ and top_{in} are the starting position of the formatted string field within its parent window.

- *width*$_{in}$ is the width of the formatted string field. (The height of the formatted string field is determined automatically by the UIW_FORMATTED_STRING class object.)

- *compressedText*$_{in}$ is an initial raw text string that will be formatted according to the *editMask* and *deleteText* arguments. The string is copied into a buffer, allocated by the UIW_FORMATTED_STRING object, and all updates are made to that buffer. To preserve any changes, **Information( )** must be called to get the modified text. However, if the **WOF_NO_ALLOCATE_DATA** flag is set, a programmer-allocated character buffer passed in as *compressedText* will be used by the UIW_-FORMATTED_STRING object.

- *editMask*$_{in}$ is a character array which indicates the type of characters that can be entered at different positions in the formatted string. This argument is always copied by the UIW_FORMATTED_STRING class object. Valid characters used to define the edit mask are:

  a—Allows the end-user to enter a space (' ') or any letter (i.e., 'a' through 'z' or 'A' through 'Z').

  A—Same as the 'a' character option except that a lower-case letter is automatically converted to an upper-case letter.

  c—Allows the end-user to enter a space (' '), a number (i.e., '0' through '9') or any alphabetic character (i.e., 'a' through 'z' or 'A' through 'Z').

  C—Same as the 'c' character option except that a lower-case character is automatically converted to upper-case.

  L—Uses this position as a literal place holder. Using this character causes the formatted string to get the character to be read and displayed from the literal mask. The end-user cannot position on nor edit this character.

  N—Allows the end-user to enter any digit.

  x—Allows the end-user to enter any printable character (i.e., ' ' through '~').

  X—Same as the 'x' character option except that a lower-case letter is automatically converted to an upper-case alphanumeric character.

- *deleteText*$_{in}$ is a character array that contains the literal characters to be used whenever a character is deleted from a particular position in the formatted string. *deleteText* is also shown as the initial text in the field.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the formatted string object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_-FORMATTED_STRING class object:

  WOF_AUTO_CLEAR—Automatically clears the string buffer if the end-user tabs to the field (from another window field) then presses a key (without first having pressed any movement or editing keys). This flag is set by default.

  WOF_BORDER—Draws a single-line border around the formatted string object, in graphics mode. In text mode, no border is drawn. This is a default argument if no other argument is provided.

  WOF_INVALID—Sets the initial status of the formatted string field to be "invalid." By default, all formatted string information is valid. A programmer may specify a formatted string field as invalid by setting this flag upon creation of the string object or by re-setting the flag through the *userFunction* (discussed below). For example, a formatted string field for a phone number may initially be set to (000) 000-0000, but the final string edited by the end-user must contain some valid phone number. In this case the initial string information does not fulfill the programmer's requirements.

  WOF_NO_ALLOCATE_DATA—Prevents the formatted string object from allocating a text buffer to store the text information. If this flag is set, the programmer must allocate the text buffer (passed as the *compressedText* parameter) that is used by the formatted string object.

  WOF_NO_FLAGS—Does not associate any special flags with the formatted string object. In this case, the string buffer will be left-justified. This flag should not be used in conjunction with any other WOF flag.

  WOF_NON_SELECTABLE—Prevents the formatted string object from being selected. If this flag is set, the end-user will not be able to edit the formatted string information.

  WOF_UNANSWERED—Sets the initial status of the formatted string field to be "unanswered." An unanswered formatted string field is displayed as blank space on the screen.

  WOF_VIEW_ONLY—Prevents the formatted string object from being edited. However, the formatted string may become current.

- *userFunction*$_{in}$ is a programmer defined function that is called whenever:

  1—the user moves onto the formatted string field (i.e., S_CURRENT message),

  2—the <ENTER> key is pressed (i.e., L_SELECT message), or

  3—the user moves to a different field in the window or a different window on the screen (i.e., S_NON_CURRENT message).

  The following arguments are passed to *userFunction* when formatted string information is entered:

  object$_{in}$—A pointer to the UIW_FORMATTED_STRING class object or to the class object derived from the UIW_FORMATTED_STRING object base class. This argument must be typecast by the programmer.

  event$_{in}$—The actual event that caused the user function to be called.

  ccode$_{in}$—The logical or system code that caused the user function to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

  L_SELECT—The <ENTER> key is pressed.

  S_CURRENT—The formatted string object is about to be edited. This code is sent before any editing operations are permitted.

  S_NON_CURRENT—A different field or window has been selected. This code is sent after editing operations have been performed.

  The user function's *returnValue* should be 0 if the formatted string is valid. Otherwise, the programmer should call the error system with an appropriate error message and return -1.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Add formatted string fields to a window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ")
        + new UIW_PROMPT(2, 1, "Formatted strings..")
```

```
        + new UIW_FORMATTED_STRING(22, 1, 41, "8017858900",      // phone number
            "LNNNLLNNNLNNNN", "(...) ...-....")
        + new UIW_FORMATTED_STRING(43, 2, 20, "840620000",       // zip code
            "NNNNNLNNNN", ".....-....");
    *windowManager + window;
    .
    .
    .

    // The formatted strings will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_FORMATTED_STRING::DataGet

Syntax

#include <ui_win.hpp>

char *DataGet(int *compressedText* = FALSE);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function gets the current formatted string buffer information associated with the UIW_FORMATTED_STRING class object. It returns a pointer to a character array.

- $returnValue_{out}$ is a constant pointer to the formatted string buffer.
- $compressedText_{in}$ denotes whether the literal characters are kept in the text buffer that is returned.

Example

```
#include <ui_win.hpp>

static int CheckAreaCode(UI_WINDOW_OBJECT *data, UI_EVENT &event,
    EVENT_TYPE ccode)
{
    // Only look for a non-current message.
    if (ccode != S_NON_CURRENT)
        return (FMI_OK);     // Continue with program.
```

```
    // Make sure the area code is not 000.
    UIW_FORMATTED_STRING *stringField = (UIW_FORMATTED_STRING *)data;
    char *number = stringField->DataGet(TRUE);
    if (number[0] != '0' || number[1] != '0' || number[2] != '0')
        return (FMI_OK);
    data->errorSystem->ReportError(stringField->windowManager, WOF_UNANSWERED,
        "The phone number you entered, %s, does not have a valid area code.",
        stringField->DataGet(FALSE));
    return (FMI_INVALID);
}

ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    // Create a phone number string field.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 8);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ")
        + new UIW_PROMPT(2, 1, "Phone number:")
        + new UIW_FORMATTED_STRING(16, 1, 15, "8017858900",
            "LNNNLLNNNLNNNN", "(...) ...-....."), WOF_NO_FLAGS, CheckAreaCode);
    *windowManager + window;
    .
    .
    .

}
```

UIW_FORMATTED_STRING::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(char *_text_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function resets the string information associated with the UIW_FORMATTED_-STRING class object or tells the class object that key flags, associated with the formatted string object, have been changed.

- $text_{in}$ is a pointer to the new string. This string must conform to the *editMask* and *deleteText* arguments passed to the formatted string constructor. If the WOF_NO_-

ALLOCATE_DATA flag is set, this argument must be space, allocated by the programmer, that is not destroyed until the UIW_FORMATTED_STRING class object is destroyed. Otherwise, the information associated with this argument is copied by the UIW_FORMATTED_STRING class object. If this argument is NULL, no string information is changed, but the formatted string field is re-displayed.

Example

```
#include <ui_win.hpp>

struct COMPANY_INFO
{
    char name[64];
    char address1[64];
    char address2[64];
    char representative[64];
    char phone[16];
};

ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    // Manually add a formatted string field to a window.
    UIW_STRING *name, *address1, *address2;
    UIW_FORMATTED_STRING *phoneNumber;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 50, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Company Information ")
        + new UIW_PROMPT(2, 1, "Name:")
        + (name = new UIW_STRING(11, 1, 30, NULL, 64))
        + new UIW_PROMPT(2, 2, "Address:")
        + (address1 = new UIW_STRING(11, 2, 30, NULL, 64))
        + (address2 = new UIW_STRING(11, 3, 30, NULL, 64))
        + new UIW_PROMPT(2, 5, "Representative:")
        + (representative = new UIW_STRING(18, 5, 30, NULL, 64))
        + new UIW_PROMPT(2, 6, "Phone Number:")
        + (phone = new UIW_FORMATTED_STRING(18, 6, 15, NULL, "LNNNLLNNNLNNNN",
            "(...) ...-...."));
    .
    .
    .

    // Get the company information and set the window information.
    COMPANY_INFO company;
    .
    .
    .

    name->DataSet(company.name);
    address1->DataSet(company.address1);
    address2->DataSet(company.address2);
    representative->DataSet(company.representative);
    phone->DataSet(company.phone);
    *windowManager + window;
}
```

UIW_FORMATTED_STRING::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a formatted string object. It is declared virtual so that any derived formatted string class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the formatted string object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified formatted string object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_CREATE—Ensures that the formatted string text reflects the internal text buffer.

  S_CURRENT and **S_NON_CURRENT**—Cause the programmer-specified user function to be called if one exists. The programmer will get a pointer to the UIW_FORMATTED_STRING class, the event and the value for the user function *ccode* will be S_CURRENT or S_NON_CURRENT.

All other events are passed by **Event( )** to **UIW_STRING::Event( )** for processing.

UIW_FORMATTED_STRING::Export

Syntax

#include <ui_win.hpp>

void Export(char *_destination_, int _expanded_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function gets the current formatted string buffer information associated with the UIW_FORMATTED_STRING class object. The text, returned by *destination*, may either be in expanded (e.g., **(801) 785-8900**) or raw form (e.g., **8017858900**).

- $destination_{in}$ is a character array, allocated by the programmer, that will receive the formatted string text. *destination* should be at least the length of the formatted string buffer plus one (i.e., for the NULL character).

- $expanded_{in}$ is TRUE to return the expanded string or FALSE to return the raw data.

Example

```
#include <ui_win.hpp>

static int CheckAreaCode(UI_WINDOW_OBJECT *data, UI_EVENT &event,
    EVENT_TYPE ccode)
{
    // Only look for a non-current message.
    if (ccode != S_NON_CURRENT)
        return (FMI_OK);     // Continue with program.

    // Make sure the area code is not 000.
    UIW_FORMATTED_STRING *stringField = (UIW_FORMATTED_STRING *)data;
    char number[16];
    stringField->Export(number, FALSE);
    if (number[0] != '0' || number[1] != '0' || number[2] != '0')
        return (FMI_OK);
    data->errorSystem->ReportError(stringField->windowManager, WOF_NO_FLAGS,
        "The phone number you entered, %s, does not have a valid prefix",
        stringField->DataGet(FALSE));
    return (FMI_INVALID);
}
```

```
ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    // Create a phone number string field.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 8);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ")
        + new UIW_PROMPT(2, 1, "Phone number:")
        + new UIW_FORMATTED_STRING(16, 1, 15, "8017858900",
            "LNNNLLNNNLNNNN", "(...) ...-....."), WOF_NO_FLAGS, CheckAreaCode);
    *windowManager + window;
    .
    .
    .

}
```

UIW_FORMATTED_STRING::Import

Syntax

#include <ui_win.hpp>

FMI_RESULT Import(char *_source_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function resets the string information associated with the UIW_FORMATTED_-STRING class object.

- *returnValue*$_{in}$ is an error code denoting the outcome of the **Import( )** operation. Valid *returnValue* codes are:

  FMI_OK—Is returned if the **Import( )** operation was successful.

  FMI_INVALID_CHARACTERS—Is returned if *source* contained invalid characters.

- *source*$_{in}$ is a pointer to the new string. This string must conform to the *editMask* and *deleteText* arguments passed to the formatted string constructor. If the WOF_NO_-

ALLOCATE_DATA flag is set, this argument must be space, allocated by the programmer, that is not destroyed until the UIW_FORMATTED_STRING class object is destroyed. Otherwise, the information associated with this argument is copied by the UIW_FORMATTED_STRING class object.

Example

```
#include <ui_win.hpp>

struct COMPANY_INFO
{
    char name[64];
    char address1[64];
    char address2[64];
    char representative[64];
    char phone[16];
};

ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    // Manually add a formatted string field to a window.
    UIW_STRING *name, *address1, *address2;
    UIW_FORMATTED_STRING *phoneNumber;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 50, 10);
    *window.
        + new UIW_BORDER
        + new UIW_TITLE(" Company Information ")
        + new UIW_PROMPT(2, 1, "Name:")
        + (name = new UIW_STRING(11, 1, 30, NULL, 64))
        + new UIW_PROMPT(2, 2, "Address:")
        + (address1 = new UIW_STRING(11, 2, 30, NULL, 64))
        + (address2 = new UIW_STRING(11, 3, 30, NULL, 64))
        + new UIW_PROMPT(2, 5, "Representative:")
        + (representative = new UIW_STRING(18, 5, 30, NULL, 64))
        + new UIW_PROMPT(2, 6, "Phone Number:")
        + (phone = new UIW_FORMATTED_STRING(18, 6, 15, NULL, "LNNNLLNNNLNNNN",
            "(...) ...-...."));
    .
    .
    .

    // Get the company information and set the window information.
    COMPANY_INFO company;
    .
    .
    .

    name->Import(company.name);
    address1->Import(company.address1);
    address2->Import(company.address2);
    representative->Import(company.representative);
    phone->Import(company.phone);
    *windowManager + window;
}
```

UIW_FORMATTED_STRING::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. UIW_FORMATTED_STRING processes no information requests; it passes them all to **UIW_STRING::Information( )**.

- *data*$_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

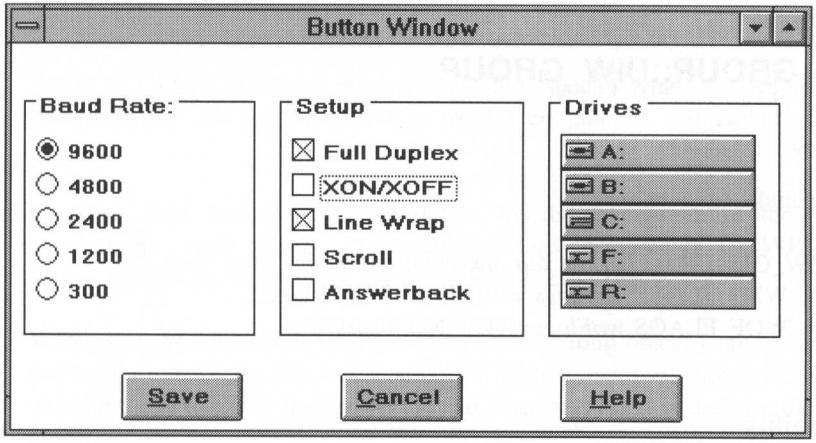
- *objectID*$_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, the request will be interpreted by UI_WINDOW_OBJECT.

Example

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

ExampleFunction()
{
        UIW_FORMATTED_STRING *fString, *fString1, *fString2;
        .
        .
        .

        char string[30];
        fString->Information(COPY_TEXT, string);
        .
        .
        .

        fString1->Information(SET_TEXT, "8017858900");
        fString2->Information(SET_TEXT, "8017858998");
        .
        .
        .

}
```

CHAPTER 51 – UIW_GROUP

The UIW_GROUP class is used to provide a physical grouping of window objects. The UIW_GROUP is displayed, in graphics mode, as a single pixel border around a specified region of the screen with an associated title. In text mode, no border is drawn. The picture below shows a graphical implementation of UIW_GROUP objects:



The UIW_GROUP class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_GROUP : public UIW_WINDOW
{
    friend class EXPORT UIF_GROUP;
public:
    // Members described in UIW_GROUP reference chapter.
    UIW_GROUP(int left, int top, int width, int height, char *text,
        WNF_FLAGS wnFlags = WNF_AUTO_SELECT,
        WOF_FLAGS woFlags = WOF_NO_FLAGS);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    char *DataGet(void);
    void DataSet(char *text);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_GROUP(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_GROUP(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
protected:
    // Members described in UIW_GROUP reference chapter.
    char *text;
};
```

- *UIF_GROUP* is for use with Zinc Designer. Programmers should not use this part of the class.

- *text* is a pointer to the text displayed as the UIW_GROUP title.

UIW_GROUP::UIW_GROUP

Syntax

#include <ui_win.hpp>

UIW_GROUP(int *left*, int *top*, int *width*, int *height*, char **text*,
WNF_FLAGS *wnFlags* = WNF_AUTO_SELECT,
WOF_FLAGS *woFlags*=WOF_NO_FLAGS);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_GROUP object.

- $left_{in}$ and top_{in} is the starting position of the group within its parent window.

- $width_{in}$ and $height_{in}$ is the size of the group.

- $text_{in}$ is the title displayed with the group. If the title has a '&' character in front of a string character, the character is displayed as a hot key character. At run-time, the group will be made current when the <Alt> key and the hot key character key are pressed simultaneously.

- $wnFlags_{in}$ are flags that determine the general operation of the group object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of a UIW_GROUP class object:

WNF_AUTO_SELECT—This flag is used if the items added to the group object are radio buttons. This permits the first radio button in the group's list to be displayed as selected by default. It also causes a button to become selected as it becomes current (i.e., if the user arrows through the buttons). This flag is set by default.

WNF_SELECT_MULTIPLE—This flag allows multiple objects within the group to be selected. This flag should be used when adding check boxes to a group object.

WNF_NO_FLAGS—Does not associate any special window flags with the group object. This flag should not be used in conjunction with any other WNF flag.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the group object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of a UIW_GROUP class object:

  WOF_NO_ALLOCATE_DATA—Prevents the group object from allocating a text buffer to store the title information. If this flag is set, the programmer must allocate the text buffer (passed as the *text* parameter) that is used by the group object.

  WOF_NO_FLAGS—Does not associate any special window object flags with the group object. This flag should not be used in conjunction with any other WOF flag. This flag is set by default.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window and add it to the window manager.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 62, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Company Information ")
        + &(*new UIW_GROUP(2, 9, 20, 6, "Group:")
            + new UIW_BUTTON(1, 1, 16, "Radio-button 1", BTF_RADIO_BUTTON,
                WOF_NO_FLAGS)
            + new UIW_BUTTON(1, 2, 16, "Radio-button 2", BTF_RADIO_BUTTON,
                WOF_NO_FLAGS)
            + new UIW_BUTTON(1, 3, 16, "Radio-button 3", BTF_RADIO_BUTTON,
                WOF_NO_FLAGS)
            + new UIW_BUTTON(1, 4, 16, "Radio-button 4", BTF_RADIO_BUTTON,
                WOF_NO_FLAGS)
            + new UIW_BUTTON(1, 5, 16, "Radio-button 5", BTF_RADIO_BUTTON,
                WOF_NO_FLAGS))
```

```
    *windowManager + window;
    .
    .
    .
}
```

UIW_GROUP::DataGet

Syntax

#include <ui_win.hpp>

char *DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to return the text information associated with the group object.

- *returnValue*$_{out}$ is a pointer to the group object's text string.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_GROUP *groupObject)
{
    char *text = groupObject->DataGet();
    .
    .
    .

}
```

UIW_GROUP::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(char *_text_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to set the text information associated with the group object.

- $value_{in}$ is a pointer to the new text string.

Example

```
#include <ui_win.hpp>

ExampleFunction1(UIW_GROUP *group)
{
    .
    .
    .
    char text[] = "Baud Rates: ";
    group->DataSet(text);
}
```

UIW_GROUP::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &_event_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a group object. It is declared virtual so that any derived group class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the group object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified group object. The type of operation depends on the interpreted value for the event. The following events are processed by **Event( )**:

  S_CURRENT, **S_NON_CURRENT**, **S_DISPLAY_ACTIVE** and **S_DISPLAY_INACTIVE**—These messages cause the group to be redisplayed. If S_CURRENT or S_NON_CURRENT are passed, the group will always be updated. If S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE are passed, the group will only be updated if *event.region* overlaps the group region.

  S_CREATE and **S_SIZE**—If the group receives one of these messages, it will automatically determine the positions of the objects in the group.

  All other events are passed by **Event( )** to **UIW_WINDOW::Event( )** for processing.

UIW_GROUP::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_GROUP, the request is passed on to UIW_WINDOW. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

  COPY_TEXT—Returns the *text* associated with the group. If this message is used, *data* is the address of a buffer where the group's text will be copied. This buffer must be large enough to contain all of the characters associated with the group and the terminating NULL character.

  GET_TEXT—Returns the *text* associated with the group. If this message is used, *data* must be the address of a character pointer that will point to the group's text string.

  SET_TEXT—Sets the *text* associated with the group box. If this message is sent, *data* must be a character pointer to the group box's new text.

  All other requests are passed by **Information( )** to **UIW_WINDOW::Information( )** for processing.

- *data*$_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

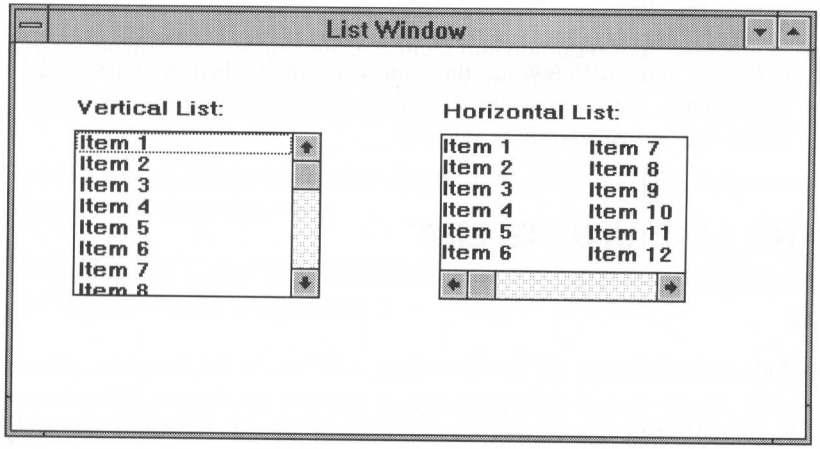
- *objectID*$_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, the request will be interpreted by UI_WINDOW_OBJECT.

Example

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

ExampleFunction()
{
        UIW_GROUP *group1, *group2;
        char string[30];

        group1->Information(COPY_TEXT, string);
        group2->Information(SET_TEXT, "Select Baud Rate:");
        .
        .
        .

}
```

CHAPTER 52 – UIW_HZ_LIST

The UIW_HZ_LIST class is used to display related information in a multiple-column fashion within a window. The list is only scrollable horizontally. If the list is taller than the *cellHeight*, it will be displayed with multiple rows. If the horizontal list is wider than the *cellWidth*, the list will be displayed with multiple columns. The figure below shows the graphical implementation of a UIW_HZ_LIST object with several string objects:



The UIW_HZ_LIST class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_HZ_LIST : public UIW_WINDOW
{
    friend class EXPORT UIF_HZ_LIST;
public:
    // Members described in UIW_HZ_LIST reference chapter.
    UIW_HZ_LIST(int left, int top, int width, int height,
        int cellWidth, int cellHeight,
        int (*compareFunction)(void *element1, void *element2) = NULL,
        WNF_FLAGS wnFlags = WNF_NO_WRAP, WOF_FLAGS woFlags = WOF_BORDER,
        WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS);
    UIW_HZ_LIST(int left, int top, int width, int height,
        int (*compareFunction)(void *element1, void *element2),
        WOF_FLAGS flagSetting, UI_ITEM *item);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_HZ_LIST(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_HZ_LIST(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
        virtual void Store(const char *name, UI_STORAGE *file,
            UI_STORAGE_OBJECT *object);

        // Members described in UI_LIST reference chapter.
        virtual void Destroy(void);

    protected:
        // Members described in UIW_HZ_LIST reference chapter.
        int cellWidth;
        int cellHeight;
};
```

- *UIF_HZ_LIST* is used with Zinc Designer only. Programmers should not use this part of the class.

- *cellWidth* and *cellHeight* are the values specified when the UIW_HZ_LIST class is constructed. In the case of the second constructor, *cellWidth* is set to be the width of the list and *cellHeight* is set to 1.

UIW_HZ_LIST::UIW_HZ_LIST

Syntax

#include <ui_win.hpp>

UIW_HZ_LIST(int *left*, int *top*, int *width*, int *height*, int *cellWidth*, int *cellHeight*,
int (**compareFunction*)(void *element1, void *element2) = NULL,
WNF_FLAGS *wnFlags* = WNF_NO_WRAP,
WOF_FLAGS *woFlags* = WOF_BORDER,
WOAF_FLAGS *woAdvancedFlags* = WOAF_NO_FLAGS);
or
UIW_HZ_LIST(int *left*, int *top*, int *width*, int *height*,
int (**compareFunction*)(void *element1, void *element2),
WOF_FLAGS *flagSetting*, UI_ITEM **item*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors return a pointer to a new UIW_HZ_LIST object.

The first constructor takes the following arguments:

- *left*$_{in}$ and *top*$_{in}$ are the starting position of the list within its parent window.

- *width*$_{in}$ and *height*$_{in}$ are the width and height of the list field.

- *cellWidth*$_{in}$ specifies the maximum cell width of a single list item.

- *cellHeight*$_{in}$ specifies the maximum cell height of a single list item.

- *compareFunction*$_{in}$ is a programmer defined function used to determine the order of list items. New items are placed at the end of a list if this value is NULL. The following arguments are passed to *compareFunction*:

  element1$_{in}$—A pointer to the first argument to compare. This argument must be typecast by the programmer.

  element2$_{in}$—A pointer to the second argument to compare. This argument must be typecast by the programmer.

  The compare function's *returnValue* should be 0 if the two elements exactly match. A negative value should be returned if *element1* is less than *element2*. Otherwise, a positive value indicates that *element1* is greater than *element2*. For more information about the *compareFunction* argument see the UI_LIST reference chapter.

- *wnFlags*$_{in}$ are flags that determine the display pattern of the list items. The following flags (declared in **UI_WIN.HPP**) control the general presentation of the list items:

  WNF_BITMAP_CHILDREN—Indicates that some of the list's sub objects contain bitmaps.

  WNF_NO_FLAGS—Does not associate any WNF_FLAGS with the list object. This flag should not be used with any other WNF flag.

  WNF_NO_WRAP—Prevents the highlight from moving from the bottom item to the top item when the down arrow is pressed. This is the default flag if no other WNF flags are specified.

  WNF_SELECT_MULTIPLE—Allows multiple items within the list to be selected.

- *woFlags*$_{in}$ are flags (general to all window objects) that determine the general

operation of the list object. The following flags (declared in **UI_WIN.HPP**) change the presentation of, or interaction with, a UIW_HZ_LIST class object:

WOF_BORDER—Draws a single line border around the list object, in graphics mode. In text mode, no border is drawn. This is the default argument if no other argument is provided.

WOF_NO_FLAGS—Does not associate any special flags with the list. This flag should not be used in conjunction with any other WOF flag.

WOF_NON_FIELD_REGION—The list object is not a form field. If this flag is set and the list is attached to a higher-level window, then the *left*, *top*, *width* and *height* arguments are ignored and the list will occupy any remaining space within the parent window. Otherwise, this advanced flag should only be used when attaching a list object directly to the screen display.

WOF_NON_SELECTABLE—The list object, and items within the object, cannot be selected. If this flag is set, the user will not be able to edit any of the list items.

- *woAdvancedFlags*$_{in}$ are flags (general to all window objects) that determine the advanced operation of the list object.

  WOAF_NO_FLAGS—Does not associate any special advanced flags with the list. Setting this flag allows the user to interact with the list in a normal fashion. This flag should not be used in conjunction with any other WOAF flag. This is the default argument if no other argument is provided.

The second constructor creates UIW_STRING objects and adds them into the list. This allows the programmer to add strings to a list without having to use the **+ operator** for each one.

- *left*$_{in}$ and *top*$_{in}$ are the starting position of the list within its parent window.

- *width*$_{in}$ and *height*$_{in}$ are the width and height of the list field.

- *compareFunction*$_{in}$ is a programmer defined function used to determine the order of list items. For more specific information see *compareFunction* under the first constructor.

- *flagSetting*$_{in}$ is the variable against which the item's flag setting is compared. If the

item's *value* and this *flagSetting* match, then the item will be marked as having been selected.

- $item_{in}$ is an array that is used to construct UIW_STRING objects and then add them to the UIW_HZ_LIST. For more information regarding the use of UI_ITEM, see "Chapter 16—UI_ITEM" of this manual.

Example

```
#include <ui_win.hpp>

ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    // Create the list field.
    UIW_HZ_LIST * list = new UIW_HZ_LIST(22, 1, 41, 6, 14, 1);
    *list
        + new UIW_STRING(0, 0, 19, "Item 1", 64, STF_NO_FLAGS)
        + new UIW_STRING(0, 0, 19, "Item 2", 64, STF_NO_FLAGS)
        + new UIW_STRING(0, 0, 19, "Item 3", 64, STF_NO_FLAGS)
        + new UIW_STRING(0, 0, 19, "Item 4", 64, STF_NO_FLAGS);

    // Attach the list to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + UIW_BORDER
        + new UIW_TITLE(" Sample list ")
        + new UIW_PROMPT(2, 1, "List: ")
        + list;
    *windowManager + window;
    .
    .
    .

    // The list will automatically be destroyed when the window
    // is destroyed.
}

ExampleFunction2(UI_WINDOW_MANAGER *windowManager)
{
    UI_ITEM listItems[] =
    {
        { 11,   NULL,   "Item 1.1", STF_NO_FLAGS },
        { 12,   NULL,   "Item 1.2", STF_NO_FLAGS },
        { 21,   NULL,   "Item 2.1", STF_NO_FLAGS },
        { 22,   NULL,   "Item 2.2", STF_NO_FLAGS },
        { 0,    NULL,   NULL, 0 }
    };

    // Create the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + UIW_BORDER
        + new UIW_TITLE(" Sample list ")
        + new UIW_PROMPT(2, 1, "List:")
        + new UIW_HZ_LIST(10, 1, 20, 6, NULL, WOF_NO_FLAGS, listItems);
    *windowManager + window;
```

```
    .
    .
    .
    // The list will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_HZ_LIST::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a list object. It is declared virtual so that any derived list class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the list object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified list object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  L_LEFT and **L_RIGHT**—If the list is displayed with multiple columns, these similar messages move the current highlight from one object to another in the same row and adjoining column.

  L_PGDN and **L_PGUP**—These similar messages move the current highlighted field one page up or down in the list.

L_PREVIOUS, **L_NEXT**, **L_UP** and **L_DOWN**—These messages move the highlight either one item up or one item down.

S_CREATE—This message causes the size of the items within the list to be automatically calculated.

S_CURRENT—This message causes the list to become current.

All other events are passed by **Event( )** to **UIW_WINDOW::Event( )** for processing.

UIW_HZ_LIST::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. UIW_HZ_LIST does not process any requests. Information requests are passed on to the current item in the list or to **UIW_WINDOW::Information( )**.

- *data*$_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

- *objectID*$_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, then the request will be interpreted by UI_WINDOW_OBJECT.

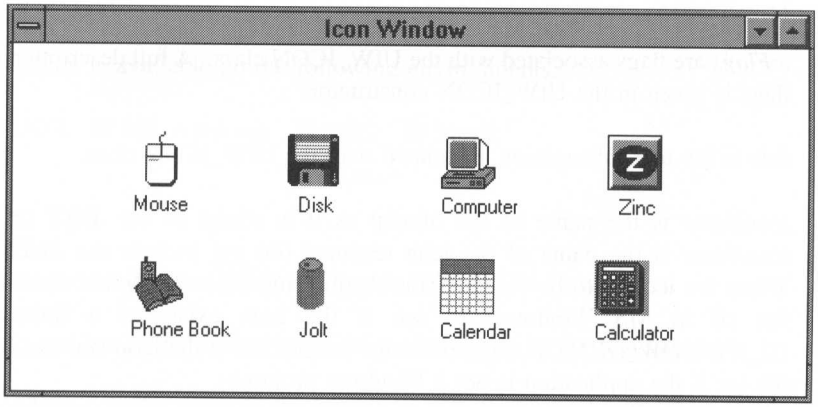
Example

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

ExampleFunction()
{
    UIW_HZ_LIST *list;
    .
    .
    .

    WOF_FLAGS flags;
    list->Information(GET_FLAGS, &flags);

    .
    .
    .

    list->Information(SET_FLAGS, (void far *)&WOF_BORDER);
    list->Information(CLEAR_FLAGS, (void far *)&WOF_NON_SELECTABLE);
    .
    .
    .

}
```

CHAPTER 53 – UIW_ICON

The UIW_ICON class is used to display a 32x32 pixel bitmap image (in graphics modes) to the screen. In text mode, only the icon's associated text (if any) will be displayed. The icon can be an image used for display only, or it can be a selectable object similar to a button, or it can be used as the icon that a window minimizes to.

Icons used within Zinc Application Framework may be created using Zinc Designer or converted from a Windows **.ICO** file. The figure below shows the graphic implementation of several UIW_ICON objects:



The UIW_ICON class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_ICON : public UI_WINDOW_OBJECT
{
    friend class EXPORT UIF_ICON;
public:
    // Members described in UIW_ICON reference chapter.
    ICF_FLAGS icFlags;

    UIW_ICON(int left, int top, char *iconName, char *title = NULL,
        ICF_FLAGS icFlags = ICF_NO_FLAGS,
        WOF_FLAGS woFlags = WOF_JUSTIFY_CENTER | WOF_NON_SELECTABLE,
        USER_FUNCTION userFunction = NULL);
    char *DataGet(void);
    void DataSet(char *text);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_ICON(const char *name, UI_STORAGE *file, UI_STORAGE_OBJECT *object);
    virtual ~UIW_ICON(void);
    static UI_ELEMENT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);

protected:
    // Members described in UIW_ICON reference chapter.
    char *title;
    char *iconName;
    int iconWidth;
    int iconHeight;
    UCHAR *iconArray;
    HICON icon;
#if defined(ZIL_MSDOS)
    UI_REGION iconRegion;
    UI_REGION titleRegion;
#endif

    EVENT_TYPE DrawItem(const UI_EVENT &event, EVENT_TYPE ccode);
};
```

- *icFlags* are flags associated with the UIW_ICON class. A full description of the icon flags is given in the UIW_ICON constructor.

- *title* is the text information associated with the UIW_ICON class.

- *iconName* is the name of the bitmap as it is stored in the **.DAT** or **.RES** file. *iconName* is the name of the icon resource (do not include the **.DAT** filename). When the icon is to be loaded, Zinc Application Framework first searches the **.res** file (if in MS Windows) to see if the icon exists as a linked resource. *UI_WINDOW_OBJECT::defaultStorage* is searched if the icon was not in the **.RES** file (or if the application is not a Windows program).

- *iconWidth* is the width of the icon bitmap.

- *iconHeight* is the height of the icon bitmap.

- *iconArray* is an array of UCHAR that contains the actual bitmap data.

- *icon* is the Windows HICON structure used to display the icon in Windows. This member is only available in Windows.

- *iconRegion* is the region occupied by the icon image. This member is available in DOS only.

- *titleRegion* contains the region occupied by the icon's title string (if any). This member is available in DOS only.

UIW_ICON::UIW_ICON

Syntax

```
#include <ui_win.hpp>

UIW_ICON(int left, int top, char *iconName, char *title = NULL,
    ICF_FLAGS icFlags = ICF_NO_FLAGS,
    WOF_FLAGS woFlags = WOF_JUSTIFY_CENTER | WOF_NON_SELECTABLE,
    USER_FUNCTION userFunction = NULL);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_ICON class object.

- $left_{in}$ and top_{in} are the starting position of the icon in the window.
- $iconName_{in}$ is the name of the bitmap as it is stored in the **.DAT** or **.RES** file.
- $title_{in}$ is the text to place directly under the icon when it is displayed on the screen.
- $icFlags_{in}$ are flags that determine the type of interaction that the icon object accepts. The following flags (declared in **UI_WIN.HPP**) control the general interaction of a UIW_ICON class object:

  ICF_NO_FLAGS—Indicates that no special options are selected with the UIW_ICON class object. In this state the icon requires two down and up clicks from the mouse to complete an action. This is the default argument if no other argument is provided.

  ICF_MINIMIZE_OBJECT—Indicates that the icon is a minimized object, such as a window that is minimized to an icon.

  ICF_DOUBLE_CLICK—The *userFunction* function, if one exists, will be called if the user double-clicks on the icon.

ICF_STATIC_ICONARRAY—Causes the bitmap array that is used for the icon image to not be deleted. By default, when an icon is created, the icon image array is converted to a native storage structure and the array is deleted. If the icon array should not be deleted after this conversion is performed (e.g., if the same icon image is to be used for multiple objects), then the ICF_STATIC_-ICONARRAY flag should be set.

- *woFlags*$_{in}$ are flags that determine the general operation of the icon object. These flags are general to all window objects. The following flags (declared in **UI_WIN.HPP**) change the presentation of, or interaction with, a UIW_ICON class object:

  WOF_BORDER—Draws a single-line border around the icon bitmap and another border around the icon's title.

  WOF_NO_FLAGS—Associates no additional window object flags with the icon object. This is the default argument if no other argument is provided.

  WOF_NON_FIELD_REGION—Indicates that the icon object is not a form field. If this flag is set and the icon is attached to a higher-level window, then the *left* and *top* arguments are ignored and the icon will occupy any remaining space within the parent window. Otherwise, this advanced flag should only be used when attaching an icon object directly to the screen display.

  WOF_NON_SELECTABLE—Indicates that the icon object cannot be selected. If this flag is set, the user will not be able to select the icon.

- *userFunction*$_{in}$ is a programmer-defined function that is called whenever the icon object is selected. An icon object is selected whenever the user positions on the icon and presses <Enter> or when the left mouse button is clicked (or double-clicked if the ICF_DOUBLE_CLICK flag is set). The following parameters are passed to *userFunction* when the icon is selected:

  object$_{in}$ is a pointer to the UIW_ICON class object or to the class object derived from the UIW_ICON object base class. Since this argument is a UI_-WINDOW_OBJECT pointer, it must be typecast by the programmer.

  event$_{in}$ is a reference pointer to a copy of the event used to reach the programmer-defined function. Since this argument is a copy of the original event, it may be changed by the programmer.

  ccode$_{in}$ is the logical event that the icon object interpreted from *event.type*.

NOTE: To change the data associated with the icon object, see **DataGet( )**, **DataSet( )** or **Information( )**.

Example

```
#include <ui_win.hpp>

main()
{
    .
    .
    .

    UI_WINDOW_OBJECT::defaultStorage = new UI_STORAGE("resource.dat");
    .
    .
    .

    // Attach the icon to a window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *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(0, 0, "minIcon", NULL, ICF_MINIMIZE_OBJECT)
        + new UIW_ICON(7, 3, "iconLogo", "Box Logo");
    *windowManager + window;
    .
    .
    .

    delete UI_WINDOW_OBJECT::defaultStorage;
    .
    .
    .

}
```

UIW_ICON::DataGet

Syntax

#include <ui_win.hpp>

char *DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to return the text information associated with the icon object.

- *returnValue*$_{out}$ is a pointer to the icon's text string.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_ICON *iconObject)
{
    char *text = iconObject->DataGet();
    .
    .
    .

}
```

UIW_ICON::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(char **text*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to set the text information associated with the icon object.

- *value*$_{in}$ is a pointer to the new text string.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_ICON *icon)
{
     .
     .
     .

     char text[] = "FILE";
     icon->DataSet(text);
}
```

UIW_ICON::DrawItem

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE DrawItem(const UI_EVENT &*event*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual advanced routine is used to draw an icon object on the screen. This function is called only if the object's woStatus has the WOS_OWNERDRAW status set.

- $returnValue_{out}$ is a response based on the success of the function call. If successful, the function returns a non-zero value. If the object was not drawn, 0 is returned.

- $event_{in}$ contains a run-time message for the specified icon object. The icon is drawn according to the type of event. The following logical events are handled by the **DrawItem( )** routine:

  S_CURRENT, **S_NON_CURRENT**, **S_DISPLAY_ACTIVE** and **S_DISPLAY_INACTIVE**—These messages cause the icon to be redisplayed. If S_CURRENT or S_NON_CURRENT are passed, the icon will always be updated. If S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE are passed the icon will only be updated if *event.region* overlaps the icon region.

WM_DRAWITEM—A message that causes the icon to be redrawn. This message is specific to Windows and OS/2.

Expose—A message that causes the icon to be redrawn. This message is specific to Motif.

- $ccode_{in}$ contains the logical interpretation of *event*.

UIW_ICON::Event

Syntax

```
#include <ui_win.hpp>

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

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to an icon object. It is declared virtual so that any derived icon class can override its default operation.

- $returnValue_{out}$ is a response based on the type of *event*. If successful, the icon object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- $event_{in}$ contains a run-time message for the specified icon object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  L_BEGIN_SELECT, L_CONTINUE_SELECT and L_END_SELECT—If no ICF flags are set and the mouse has been clicked while *event.position* is overlapping the icon, any of these messages will call the user function. If the icon is attached directly to the screen, it sends the S_MOVE message to the Window Manager via the Event Manager.

S_CREATE, S_SIZE—Determines the actual region for the icon bitmap and associated string.

S_CURRENT, S_DISPLAY_ACTIVE, S_DISPLAY_INACTIVE and S_-NON_CURRENT—Cause the icons bitmap and string to be re-displayed to the screen. If the icon is current, the string is displayed with highlighting turned on.

All other events are passed by **Event( )** to **UI_WINDOW_OBJECT::Event( )** for processing.

UIW_ICON::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- $returnValue_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- $request_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_ICON, the request is passed on to UI_WINDOW_OBJECT. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

CLEAR_FLAGS—Clears the ICF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_ICON or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

COPY_TEXT—Returns the *text* associated with the icon. If this message is used, *data* is the address of a buffer where the icon's text will be copied. This buffer must be large enough to contain all of the characters associated with the icon and the terminating NULL character.

GET_FLAGS—Returns the ICF_FLAGS associated with an object if the value in *objectID* is ID_ICON or it returns the WOF_FLAGS if *objectID* is ID_-WINDOW_OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

GET_ICON_ARRAY—Returns a pointer to the icon bitmap array associated with an icon. If a bitmap does not exist, NULL is returned. If this message is sent, *data* must be a pointer to an array of UCHAR.

GET_TEXT—Returns the *text* associated with the icon. If this message is used, *data* must be the address of a character pointer that will point to the icon's text string.

SET_FLAGS—Sets the ICF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_ICON or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. If successful, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

SET_ICON_ARRAY—Sets the icon bitmap array associated with an icon. If this message is sent, *data* must be a pointer to an array of UCHAR containing the icon's new bitmap. Since icons are 32x32, the array, pointed at by *data*, must be 1024 UCHAR in length.

SET_TEXT—Sets the *text* associated with the icon. If this message is sent, *data* must be a character pointer to the icon's new text.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

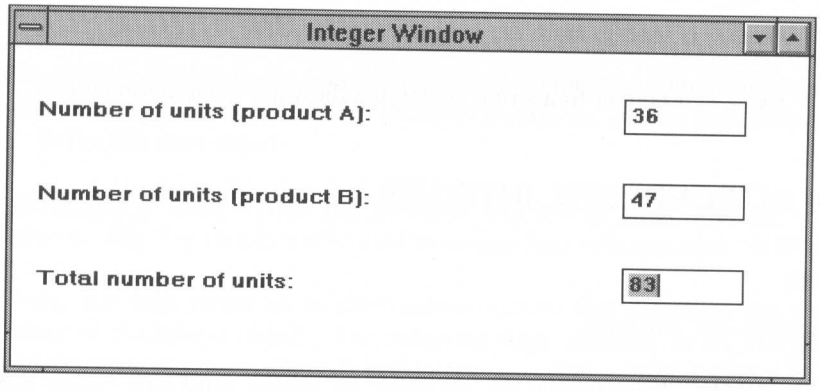
- *objectID*$_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, the request will be interpreted by UI_WINDOW_OBJECT.

Example

```
#include <ui_win.hpp>

ExampleFunction()
{
    UIW_ICON *icon, *icon1, *icon2;

    char string[30];
    icon->Information(COPY_TEXT, string);
    icon1->Information(SET_TEXT, "Clock");
    icon2->Information(SET_TEXT, "Phone Book");
    .
    .
    .

}
```

CHAPTER 54 – UIW_INTEGER

The UIW_INTEGER class is used to display numeric information to the screen and to collect information, in integer form, from an end user. For special formatting (e.g., commas, currency) see the UIW_BIGNUM class. The figure below shows a graphic implementation of a window with several variations of the UIW_INTEGER class object:



The UIW_INTEGER class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class UIW_INTEGER : public UIW_STRING
{
    friend class EXPORT UIF_INTEGER;
public:
    // Members described in UIW_INTEGER reference chapter.
    NMF_FLAGS nmFlags;
    static UI_ERROR_TABLE *errorTable;

    UIW_INTEGER(int left, int top, int width, char *value,
        const char *range = NULL, NMF_FLAGS nmFlags = NMF_NO_FLAGS,
        WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
        USER_FUNCTION userFunction = NULL);
    int DataGet(void);
    void DataSet(int *value);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    virtual ~UIW_INTEGER(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    UIW_INTEGER(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);

    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
protected:
    int *number;
    char *range;
};
```

- *UIF_INTEGER* is used with Zinc Designer only. Programmers should not use this part of the class.
- *nmFlags* contain the number flags associated with the UIW_INTEGER object.
- *number* is used to store the integer value for UIW_INTEGER.
- *range* is a pointer to a string containing the valid range information.

UIW_INTEGER::UIW_INTEGER

Syntax

#include <ui_win.hpp>

UIW_INTEGER(int *left*, int *top*, int *width*, int **value*, const char **range* = NULL,
NMF_FLAGS *nmFlags* = NMF_NO_FLAGS,
WOF_FLAGS *woFlags* = WOF_BORDER | WOF_AUTO_CLEAR,
USER_FUNCTION *userFunction* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_INTEGER class object. The type of integer object created depends on the *value* argument.

- $left_{in}$ and top_{in} is the starting position of the integer field within its parent window.
- $width_{in}$ is the width of the integer field. (The height of the integer field is determined automatically by the UIW_INTEGER class object.)

- *value*$_{in}$ is a pointer to the default numeric value. This pointer is used by the UIW_-INTEGER object if the WOF_NO_ALLOCATE_DATA flag is used. Otherwise, the value is copied into a member variable allocated by UIW_INTEGER.

- *range*$_{in}$ is a string that gives the range for valid numeric values. For example, if a range of "1000..10000" were specified, the UIW_INTEGER class object would only accept those numeric values that fell between 1,000 and 10,000. If *range* is NULL, any integer (within the absolute range) is accepted. This string is copied by the UIW_INTEGER class object.

- *nmFlags*$_{in}$ gives information on how to display and interpret the numeric information. The following flag (declared in **UI_WIN.HPP**) control the general presentation of a UIW_INTEGER class object:

  NMF_NO_FLAGS—Does not associate any special flags with the integer object. This flag should not be used in conjunction with any other NMF flags.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the integer object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_INTEGER class object:

  WOF_AUTO_CLEAR—Automatically clears the numeric buffer if the end-user tabs to the integer field (from another window field) then presses a key (without first having pressed any movement or editing keys). This is one of the default flags.

  WOF_BORDER—In graphics mode, setting this option draws a single line border around the object. In text mode, no border is drawn. This is one of the default flags.

  WOF_INVALID—Sets the initial status of the integer field to be "invalid." Invalid integers fit in the absolute range determined by the object type but do not fulfill all the requirements specified by the program. For example, an integer may initially be set to 200, but the final integer, edited by the end-user, must be in the range "10..100." The initial integer in this example fits the absolute requirements of an UIW_INTEGER class object but does not fit into the specified range.

  WOF_JUSTIFY_CENTER—Center-justifies the numeric information associated with the integer object.

WOF_JUSTIFY_RIGHT—Right-justifies the numeric information associated with the integer object.

WOF_NO_ALLOCATE_DATA—Prevents the integer object from allocating a numeric value to store the numeric information. If this flag is set, the programmer must allocate the integer (passed as the *value* parameter) that is used by the integer object.

WOF_NO_FLAGS—Does not associate any special window flags with the integer object. Setting this flag left-justifies the numeric information. This flag should not be used in conjunction with any other WOF flags.

WOF_NON_SELECTABLE—Prevents the integer object from being selected. If this flag is set, the user will not be able to edit the numeric information.

WOF_UNANSWERED—Sets the initial status of the integer field to be "unanswered." An unanswered integer field is displayed as blank space on the screen.

WOF_VIEW_ONLY—Prevents the integer object from being edited, however, it may become current.

- *userFunction*$_{in}$ is a programmer defined function that is called whenever:

  1—the user moves onto the integer field (i.e., S_CURRENT message),

  2—the <ENTER> key is pressed. (i.e., L_SELECT message), or

  3—the user moves to a different field on the window or a different window on the screen (i.e., S_NON_CURRENT message).

The following arguments are passed to *userFunction*:

object$_{in}$—A pointer to the UIW_INTEGER class object. This argument must be typecast by the programmer.

event$_{in}$—The event that caused *userFunction* to be called.

ccode$_{in}$—The logical or system code that caused the userFunction to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

L_SELECT—The <ENTER> key was pressed.

S_CURRENT—The integer object is about to be edited. This code is sent before any editing operations are permitted.

S_NON_CURRENT—A different field or window has been selected. This code is sent after editing operations have been performed.

If no *userFunction* is specified, then **Validate( )** is automatically called. However, if a user function is specified, *userFunction* should call **Validate( )** to check the range of the new integer value.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window and add it to the window manager.
    int iValue = 0;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample numbers ")
        + new UIW_PROMPT(2, 1, "Integer:")
        + new UIW_INTEGER(12, 1, 20, &iValue, "0..10000");
    *windowManager + window;
    .
    .
    .

    // The integer will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_INTEGER::DataGet

Syntax

#include <ui_win.hpp>

int DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to return the integer value of the integer object.

- *returnValue*$_{out}$ is the integer value.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_INTEGER *integerObject)
{
    int value = integerObject->DataGet();
    .
    .
    .
}
```

UIW_INTEGER::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(int **value*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to set the integer information associated with the integer object.

- *value*$_{in}$ is a pointer to the new integer value.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_INTEGER *number)
{
    .
    .
    .
    int amount = 100;
    number->DataSet(&amount);
}
```

UIW_INTEGER::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to an integer object. It is declared virtual so that any derived integer class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the integer object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified integer object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_CREATE and **S_NON_CURRENT**—Ensure that the integer (shown to the screen) reflects the internal integer associated with the integer object.

  S_CURRENT—Updates the integer text on the display.

All other events are passed by **Event( )** to **UIW_STRING::Event( )** for processing.

UIW_INTEGER::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

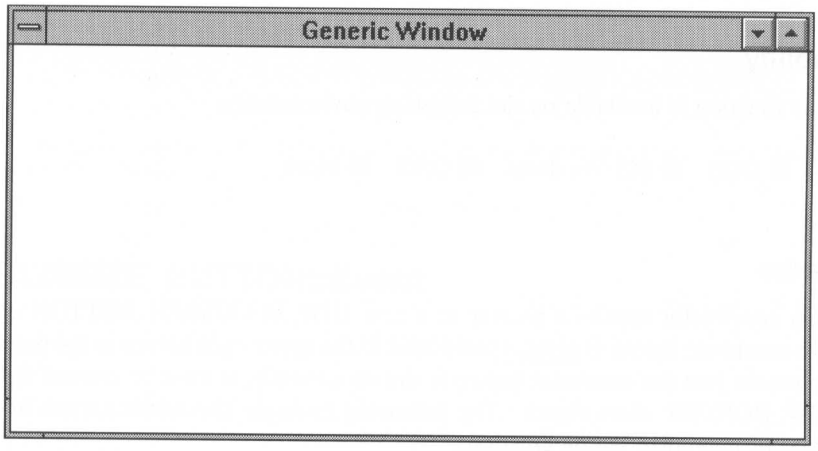
■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object. **Information( )** does not process any messages, but passes them to **UIW_STRING::Information( )** for processing.

CHAPTER 55 – UIW_MAXIMIZE_BUTTON

The UIW_MAXIMIZE_BUTTON class is used to maximize a window. A maximized window fills the whole screen. The figure below shows a graphical implementation of a window with a UIW_MAXIMIZE_BUTTON class object (the button with the '▲' character):



The UIW_MAXIMIZE_BUTTON class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_MAXIMIZE_BUTTON : public UIW_BUTTON
{
    friend class EXPORT UIF_MAXIMIZE_BUTTON;
public:
    // Members described in UIW_MAXIMIZE_BUTTON reference chapter.
    UIW_MAXIMIZE_BUTTON(void);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_MAXIMIZE_BUTTON(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_MAXIMIZE_BUTTON(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
};
```

- *UIF_MAXIMIZE_BUTTON* is used with Zinc Designer only. Programmers should not use this part of the class.

UIW_MAXIMIZE_BUTTON::UIW_MAXIMIZE_BUTTON

Syntax

#include <ui_win.hpp>

UIW_MAXIMIZE_BUTTON(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_MAXIMIZE_BUTTON class object. The maximize button is always positioned in the upper right corner in the parent window. To ensure that the maximize button is drawn correctly, it must be created right after the UIW_BORDER class object. The following example shows the correct and incorrect order of maximize button creation:

```
1)  // CORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1")
        .
        .
        .

2)  // INCORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1")
        + new UIW_MAXIMIZE_BUTTON
        .
        .
        .
```

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window and attach it to the window manager.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1");
    *windowManager + window;
    .
    .
    .

    // The maximize button will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_MAXIMIZE_BUTTON::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT *&event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a maximize button object. It is declared virtual so that any derived maximized button class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the maximize button object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- $event_{in}$ contains a run-time message for the specified maximize button object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_CREATE—This message will cause the **Event( )** to set the character to be displayed and recalculates its position and size within its parent window. This is an up arrow ('▲') if the window can be maximized or a double arrow ('↕') if the window is already maximized.

  S_DISPLAY_ACTIVE and **S_DISPLAY_INACTIVE**—In text mode, these messages determine whether the maximize button (as well as the minimize and system buttons) is displayed and updated. It is displayed and updated only if the window is in an active state. If the window is not active, the maximize button is not displayed nor updated.

All other events are passed by **Event( )** to **UIW_BUTTON::Event( )** for processing.

UIW_MAXIMIZE_BUTTON::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

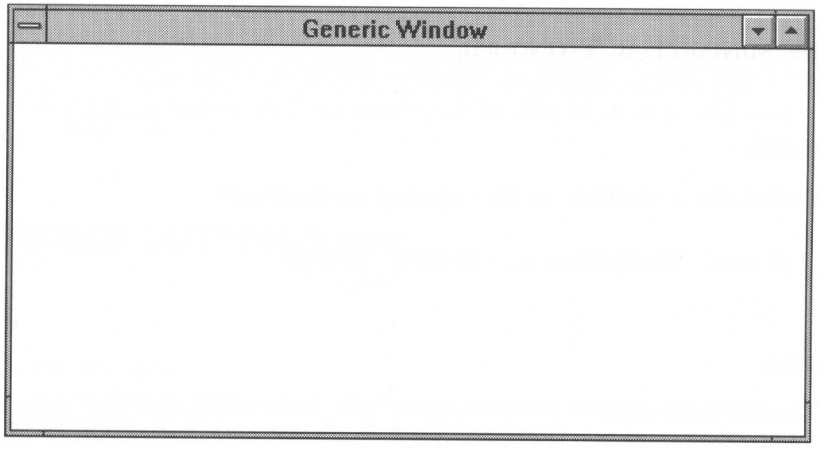
■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object. **Information( )** does not process any messages, but passes them to **UIW_BUTTON::Information( )** for processing.

CHAPTER 56 – UIW_MINIMIZE_BUTTON

The UIW_MINIMIZE_BUTTON class is used to minimize a window. If an icon with the ICF_MINIMIZE_OBJECT flag set has been added to the window, the window is reduced to that icon when the minimize button is selected. The figure below shows a graphical implementation of a window with a UIW_MINIMIZE_BUTTON object (the button with the '▾' character):



The UIW_MINIMIZE_BUTTON class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_MINIMIZE_BUTTON : public UIW_BUTTON
{
    friend class EXPORT UIF_MINIMIZE_BUTTON;
public:
    // Members described in UIW_MINIMIZE_BUTTON reference chapter.
    UIW_MINIMIZE_BUTTON(void);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_MINIMIZE_BUTTON(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_MINIMIZE_BUTTON(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
};
```

- *UIF_MINIMIZE_BUTTON* is used with Zinc Designer only. Programmers should not use this part of the class.

UIW_MINIMIZE_BUTTON::UIW_MINIMIZE_BUTTON

Syntax

#include <ui_win.hpp>

UIW_MINIMIZE_BUTTON(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_MINIMIZE_BUTTON class object. The minimize button object is always positioned in the upper right corner in the parent window. To ensure that the minimize button is drawn correctly, it must be added right after the UIW_MAXIMIZE_BUTTON class object. The following example shows the correct and incorrect order of minimize button creation:

```
1)  // CORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1")
        .
        .
        .

2)  // INCORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_MINIMIZE_BUTTON
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1")
        + new UIW_BORDER
        .
        .
        .
```

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window and attach it to the window manager.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1");
    *windowManager + window;
    .
    .
    .

    // The minimize button will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_MINIMIZE_BUTTON::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a minimize button object. It is declared virtual so that any derived minimized button class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the minimize button object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified minimize button object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_CREATE—This message will cause **Event( )** to recalculate the minimize button's position and size within its parent window.

  S_DISPLAY_ACTIVE and **S_DISPLAY_INACTIVE**—In text mode, these messages determine whether the minimize button (as well as the maximize and system buttons) is displayed and updated. It is displayed and updated only if the window is in an active state; if the window is not active, the minimize button is not displayed nor updated. If the application is running in graphics mode, these messages are processed by the **UIW_BUTTON::Event( )** function.

  All other events are passed by **Event( )** to **UIW_BUTTON::Event( )** for processing.

UIW_MINIMIZE_BUTTON::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

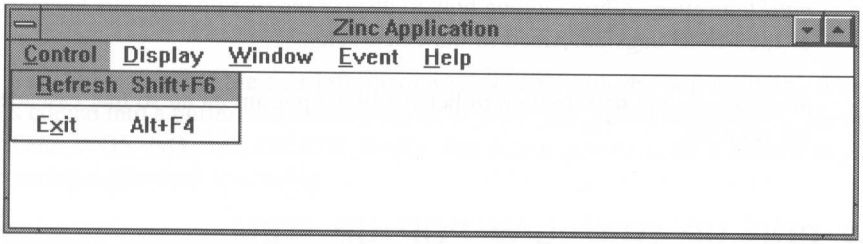
■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object. **Information( )** does not process any messages, but passes them to **UIW_BUTTON::Information( )** for processing.

CHAPTER 57 – UIW_POP_UP_ITEM

The UIW_POP_UP_ITEM class is used to display and select options associated with a list of menu items. The figure below shows a graphical implementation of the UIW_POP_UP_ITEM objects (shown as "Refresh" and "Exit" on the pop-up menu):



The UIW_POP_UP_ITEM class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_POP_UP_ITEM : public UIW_BUTTON
{
    friend class EXPORT UIF_POP_UP_ITEM;
public:
    // Members described in UIW_POP_UP_ITEM reference chapter.
    MNIF_FLAGS mniFlags;
    UIW_POP_UP_MENU menu;

    UIW_POP_UP_ITEM(void);
    UIW_POP_UP_ITEM(char *text, MNIF_FLAGS mniFlags = MNIF_NO_FLAGS,
        BTF_FLAGS btFlags = BTF_NO_3D, WOF_FLAGS woFlags = WOF_NO_FLAGS,
        USER_FUNCTION userFunction = NULL, EVENT_TYPE value = 0);
    UIW_POP_UP_ITEM(int left, int top, int width, char *text,
        MNIF_FLAGS mniFlags = MNIF_NO_FLAGS,
        BTF_FLAGS btFlags = BTF_NO_FLAGS, WOF_FLAGS woFlags = WOF_NO_FLAGS,
        USER_FUNCTION userFunction = NULL, EVENT_TYPE value = 0);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_POP_UP_ITEM(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_POP_UP_ITEM(void);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);

    // Members described in UI_LIST reference chapter.
    UI_WINDOW_OBJECT *Add(UI_WINDOW_OBJECT *object);
    UI_WINDOW_OBJECT *Subtract(UI_WINDOW_OBJECT *object);
    UIW_POP_UP_ITEM &operator+(UI_WINDOW_OBJECT *object);
    UIW_POP_UP_ITEM &operator-(UI_WINDOW_OBJECT *object);
```

```
protected:
    // Members described in UIW_POP_UP_ITEM reference chapter.
    virtual EVENT_TYPE DrawItem(const UI_EVENT &event, EVENT_TYPE ccode);
};
```

- *UIF_POP_UP_ITEM* is used with Zinc Designer only. Programmers should not use this part of the class.

- *mniFlags* gives information on how to display the pop-up item. Its value can be any of the MNIF flags described in the constructor.

- *menu* allows the pop-up item to behave like a pop-up menu so that cascaded menus are possible.

UIW_POP_UP_ITEM::UIW_POP_UP_ITEM

Syntax

#include <ui_win.hpp>

UIW_POP_UP_ITEM(void);
or
UIW_POP_UP_ITEM(char **text*, MNIF_FLAGS *mniFlags* = MNIF_NO_FLAGS,
BTF_FLAGS *btFlags* = BTF_NO_3D,WOF_FLAGS *woFlags* = WOF_NO_FLAGS,
USER_FUNCTION *userFunction* = NULL, EVENT_TYPE *value* = 0);
or
UIW_POP_UP_ITEM(int *left*, int *top*, int *width*, char **text*,
MNIF_FLAGS *mniFlags* = MNIF_NO_FLAGS,
BTF_FLAGS *btFlags* = BTF_NO_FLAGS,
WOF_FLAGS *woFlags* = WOF_NO_FLAGS,
USER_FUNCTION *userFunction* = NULL, EVENT_TYPE *value* = 0);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors return a pointer to a new UIW_POP_UP_ITEM class

object.

The first constructor takes no arguments. It places a menu item separator (horizontal line) in the parent pop-up menu.

The second constructor creates a pop-up item. This constructor can be used when the pop-up item is to be added to a parent pop-up menu or to a pull-down item. It takes the following arguments:

- *text*$_{in}$ is a pointer to the text information associated with the pop-up item. The text is copied into a buffer allocated by the UIW_POP_UP_ITEM class object. However, if the **WOF_NO_ALLOCATE_DATA** flag is set, *text* must be a pointer to a programmer-allocated text string.

- *mniFlags*$_{in}$ gives information on how to display the menu item. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of the pop-up item:

  MNIF_CHECK_MARK—Marks the first position of the menu item's string information with a check-mark if the item has been selected (i.e., the WOS_-SELECTED status flag is set).

  MNIF_NO_FLAGS—Does not associate any special flags with the pop-up item. This flag should not be used in conjunction with any other MNIF flags. This is the default flag if no other MNIF flag is used.

  MNIF_SEPARATOR—The menu item is a separator (i.e., a horizontal line used to separate menu items). (It has no text information associated with it.)

  MNIF_SEND_MESSAGE—Causes an event to be created and put on the event queue when the menu item is selected. The menu item's *value* is copied into the *event.type* field. If this flag is set, the *userFunction* parameter should be NULL. This flag is equivalent to the BTF_SEND_MESSAGE flag.

- *btFlags*$_{in}$ gives information on how to display the menu item. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of a UIW_POP_UP_ITEM class object:

  BTF_DOUBLE_CLICK—Completes the item's action when the item has been selected twice, with the mouse, within a period of time specified by *UI_-WINDOW_OBJECT::doubleClickRate*.

BTF_DOWN_CLICK—Completes the item's action on an mouse down-click, rather than on a down-click and release action.

BTF_NO_FLAGS—Does not associate any special flags with the UIW_-POP_UP_ITEM class object. In this case the item requires a down and up click from the mouse to complete an action. This flag should not be used in conjunction with any other BTF flags.

BTF_NO_TOGGLE—Does not toggle the item's WOS_SELECTED status flag. If this flag is set, the WOS_SELECTED window object status flag is not set when the menu item is selected.

BTF_SEND_MESSAGE—Creates a UI_EVENT with *value* as the type. This event is then placed on the Event Manager's queue. If this flag is set, the *userFunction* parameter should be NULL. This flag is equivalent to the MNIF_SEND_MESSAGE flag.

- $woFlags_{in}$ are flags (common to all window objects) that determine the general operation of the window object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a pop-up item:

  WOF_BORDER—Draws a single line border around the pop-up item, in graphics mode. In text mode, no border is drawn.

  WOF_JUSTIFY_CENTER—Center-justifies the *text* information associated with the pop-up item.

  WOF_JUSTIFY_RIGHT—Right-justifies the *text* information associated with the pop-up item.

  WOF_NO_ALLOCATE_DATA—Prevents the pop-up item object from allocating a string buffer to store the pop-up item's text information. If this flag is set, the programmer must allocate the text buffer (passed as the *text* parameter) that is used by the pop-up item.

  WOF_NO_FLAGS—Does not associate any special flags with the pop-up item. In this case, the pop-up item's string information will be left-justified. This flag should not be used in conjunction with any other WOF flags.

  WOF_NON_SELECTABLE—Prevents the pop-up item from being selected.

- *userFunction*$_{in}$ is a programmer-defined function that is called whenever the pop-up item is selected.

  A default function is provided if the MNIF_MAXIMIZE, MNIF_MINIMIZE, MNIF_MOVE, MNIF_RESTORE or MNIF_SIZE flag is set for a particular pop-up item and the supplied *userFunction* is NULL. These default functions send messages to maximize, minimize, move, restore or size the parent window.

  A pop-up item object is selected whenever the user is positioned on the item and presses <Enter> or when the left mouse button is clicked. The following parameters are passed to *userFunction* when the item is selected:

  object$_{in}$ is a pointer to the UIW_POP_UP_ITEM object or to the object derived from the UIW_POP_UP_ITEM object base class. This argument should be typecast by the programmer.

  event$_{in}$ is a reference pointer to a copy of the event used to reach the programmer-defined function. Since this argument is a copy of the original event, it may be changed by the programmer.

  ccode$_{in}$ is the logical event that was interpreted from *event*.

- *value*$_{in}$ is a unique value that the programmer can associate with a pop-up item. For example, the programmer could associate a value of 0 with an "Exit" item and a value of 1 with a "Save" item. This allows the programmer to define one user-function that looks at the item values, instead of several user-functions that are tied to each pop-up item object.

The third constructor creates a pop-up item. This constructor can be used when the pop-up item is to be added to a parent pop-up menu or to a pull-down item. It takes the following arguments:

- *left*$_{in}$ specifies the location of the left edge of the pop-up item within its parent.

- *top*$_{in}$ specifies the location of the top edge of the pop-up item within its parent.

- *width*$_{in}$ specifies the width of the pop-up item. The pop-up item's height is calculated automatically.

- *text*$_{in}$ is a pointer to the text information associated with the pop-up item. The text is copied into a buffer allocated by the UIW_POP_UP_ITEM class object. However,

if the **WOF_NO_ALLOCATE_DATA** flag is set, *text* must be a pointer to a programmer-allocated text string.

- *mniFlags*$_{in}$ gives information on how to display the menu item. For a description of valid MNIF_FLAGS, see the description for the second constructor, above.
- *btFlags*$_{in}$ gives information on how to display the menu item. For a description of valid MNIF_FLAGS, see the description for the second constructor, above.
- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the window object. For a description of valid MNIF_FLAGS, see the description for the second constructor, above.
- *userFunction*$_{in}$ is a programmer-defined function that is called whenever the pop-up item is selected. For a description of how the user function is used, see the description of the second constructor, above.
- *value*$_{in}$ is a unique value that the programmer can associate with a pop-up item. For example, the programmer could associate a value of 0 with an "Exit" item and a value of 1 with a "Save" item. This allows the programmer to define one user-function that looks at the item values, instead of several user-functions that are tied to each pop-up item object.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + &(*new UIW_SYSTEM_BUTTON
            + new UIW_POP_UP_ITEM("~Restore", MNIF_RESTORE)
            + new UIW_POP_UP_ITEM("~Move", MNIF_MOVE)
            + new UIW_POP_UP_ITEM("~Size", MNIF_SIZE))
        + new UIW_TITLE("Window 1");
    *windowManager + window;
    .
    .
    .

    // The pop-up items are automatically destroyed when the window
    // is destroyed.
}
```

UIW_POP_UP_ITEM::DrawItem

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE DrawItem(const UI_EVENT &*event*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual advanced routine is used to draw a pop-up item object on the screen. This function is called only if the object's woStatus has the WOS_OWNERDRAW status set.

- $returnValue_{out}$ is a response based on the success of the function call. If successful, the function returns a non-zero value. If the object was not drawn, 0 is returned.

- $event_{in}$ contains a run-time message for the specified pop-up item object. The pop-up item is drawn according to the type of event. The following logical events are handled by the **DrawItem( )** routine:

  S_CURRENT, **S_NON_CURRENT**, **S_DISPLAY_ACTIVE** and **S_DISPLAY_INACTIVE**—These messages cause the pop-up item to be redisplayed. If S_CURRENT or S_NON_CURRENT are passed, the pop-up item will always be updated. If S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE are passed the pop-up item will only be updated if *event.region* overlaps the pop-up item region.

  WM_DRAWITEM—A message that causes the pop-up item to be redrawn. This message is specific to Windows and OS/2.

  Expose—A message that causes the pop-up item to be redrawn. This message is specific to Motif.

- $ccode_{in}$ contains the logical interpretation of *event*.

UIW_POP_UP_ITEM::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a pop-up item object. It is declared virtual so that any derived pop-up item class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the pop-up item object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified pop-up item object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_CURRENT, **S_NON_CURRENT**, **S_DISPLAY_ACTIVE** and **S_DISPLAY_INACTIVE**—If a pop-up item receives any one of these messages and it has the MNIF_SEPARATOR flag set, a line is drawn inside of the item instead of text.

  L_SELECT and **L_CONTINUE_SELECT**—These messages cause the pop-up item to be selected and its *userFunction* to be called.

  All other events are passed by **Event( )** to **UIW_BUTTON::Event( )** for processing.

UIW_POP_UP_ITEM::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_POP_UP_ITEM, then the request is passed on to UIW_BUTTON The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

  CLEAR_FLAGS—Clears the MNIF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_POP_UP_ITEM, BTF_FLAGS if *objectID* is ID_BUTTON, or WOF_FLAGS if *objectID* is ID_WINDOW-_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  COPY_TEXT—Returns the *text* associated with the pop-up item. If this message is used, *data* is the address of a buffer where the pop-up item's text will be copied. This buffer must be large enough to contain all of the characters associated with the pop-up item and the terminating NULL character.

GET_FLAGS—Returns the MNIF_FLAGS associated with an object if the value in *objectID* is ID_POP_UP_ITEM, BTF_FLAGS if *objectID* is ID_BUTTON, or WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

GET_NUMBERID_OBJECT—Returns a pointer to the item in the menu whose numberID matches the numberID passed in through the *data* argument. If this message is sent, *data* must be a pointer to a programmer defined USHORT.

GET_STRINGID_OBJECT—Returns a pointer to the item in the menu whose stringID matches the stringID passed in through the *data* argument. If this message is sent, *data* must be a pointer to a string.

GET_TEXT—Gets the *text* associated with the pop-up item. If this message is sent, *data* must be a character pointer that will be set to the pop-up item's text.

GET_VALUE—Returns the *value* associated with the pop-up item. If this message is sent, *data* must be a pointer to a programmer defined EVENT_TYPE where the pop-up item's value will be copied.

SET_FLAGS—Sets the MNIF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_POP_UP_ITEM, BTF_FLAGS if *objectID* is ID_BUTTON, or WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

SET_TEXT—Sets the *text* associated with the pop-up item. If this message is sent, *data* must be a character pointer to the pop-up item's new text.

SET_VALUE—Sets the *value* associated with the pop-up item. If this message is sent, *data* must be a pointer to a programmer defined EVENT_TYPE that contains the pop-up item's new value.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

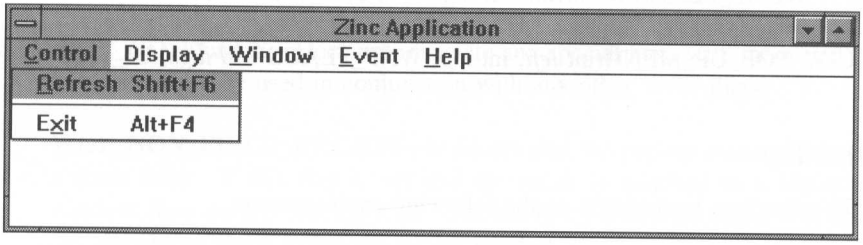
- $objectID_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, the request will be interpreted by UIW_BUTTON.

Example

```
#include <ui_win.hpp>

ExampleFunction()
{
        MNIF_FLAGS flags;
        item->Information(GET_FLAGS, &flags, ID_POP_UP_ITEM);
        .
        .
        .

}
```

CHAPTER 58 – UIW_POP_UP_MENU

The UIW_POP_UP_MENU class is used as the control structure for selectable menu items. This advanced class is to be used by the UIW_PULL_DOWN_ITEM class and should not be used directly by the programmer. The figure below shows a graphical implementation of the UIW_POP_UP_MENU class object with several pull-down menu items:



The UIW_POP_UP_MENU class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_POP_UP_MENU : public UIW_WINDOW
{
    friend class EXPORT UIF_POP_UP_MENU;
public:
    // Members described in UIW_POP_UP_MENU reference chapter.
    UIW_POP_UP_MENU(int left, int top, WNF_FLAGS wnFlags,
        WOF_FLAGS woFlags = WOF_BORDER,
        WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS);
    UIW_POP_UP_MENU(int left, int top, WNF_FLAGS wnFlags, UI_ITEM *item);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_POP_UP_MENU(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_POP_UP_MENU(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
};
```

- *UIF_POP_UP_MENU* is used with Zinc Designer only. Programmers should not use this part of the class.

UIW_POP_UP_MENU::UIW_POP_UP_MENU

Syntax

#include <ui_win.hpp>

UIW_POP_UP_MENU(int *left*, int *top*, WNF_FLAGS *wnFlags*,
WOF_FLAGS *woFlags* = WOF_BORDER,
WOAF_FLAGS *woAdvancedFlags* = WOAF_NO_FLAGS);
or
UIW_POP_UP_MENU(int *left*, int *top*, WNF_FLAGS *wnFlags*, UI_ITEM **item*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These constructors return a pointer to a new UIW_POP_UP_MENU class object.

The first constructor takes the following arguments:

- $left_{in}$ and top_{in} are the starting position of the menu within its parent window. (The ending position is computed automatically by the UIW_POP_UP_MENU class object according to the size of the menu items.)

- $wnFlags_{in}$ gives information on how to display the items in the pop-up menu. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of the pop-up menu:

  WNF_NO_FLAGS—Does not associate any special flags with the pop-up menu. This flag should not be used in conjunction with any other WNF flags.

  WNF_SELECT_MULTIPLE—Allows more than one menu item to be selected from the menu.

  WNF_NO_WRAP—Prevents the highlight from moving from the bottom item to the top item when the down arrow is pressed.

WNF_AUTO_SORT—Automatically sorts the menu items in alphabetical order.

- *woFlags*$_{in}$ are flags (general to all window objects) that determine the general operation of the pop-up menu. The following flags (declared in **UI_WIN.HPP**) change the presentation of, or interaction with, a UIW_POP_UP_MENU class object:

  WOF_BORDER—Draws a single line border around the menu, in graphics mode. In text mode, a border is drawn only if the pop-up menu has the WOAF_TEMPORARY flag set.

  WOF_NO_FLAGS—Does not associate any special flags with the menu. This flag should not be used in conjunction with any other WOF flags.

  WOF_NON_FIELD_REGION—Indicates that the pop-up menu object is not a form field. If this flag is set and the menu is attached to a higher-level window, then the *left* and *top* arguments are ignored and the menu will occupy any remaining space within the parent window.

  WOF_NON_SELECTABLE—Prevents the menu object and items within the object from being selected. If this flag is set, the user will not be able to position on any of the menu items.

- *woAdvancedFlags*$_{in}$ are flags that determine the advanced operation of the pop-up menu.

  WOAF_NO_FLAGS—Does not associate any special advanced flags with the menu. Setting this flag allows the user to interact with the menu in a normal fashion. This is the default flag if no other WOAF flag is set. It should not be used in conjunction with any other WOAF flags.

  WOAF_TEMPORARY—The menu only occupies the screen temporarily. Once another window is selected, the temporary menu is destroyed.

  WOAF_NO_DESTROY—Prevents the Window Manager from calling the pop-up menu destructor. If this flag is set, the menu can be removed from the screen display, but the programmer must call the associated menu destructor.

  WOAF_NO_SIZE—Prevents the end-user from changing the size of the pop-up menu during an application.

  WOAF_NO_MOVE—Prevents the end-user from changing the screen location of the window during an application.

WOAF_MODAL—Prevents any other window from receiving event information from the Window Manager. A modal window receives all event information until it is removed from the screen display.

WOAF_LOCKED—Prevents the Window Manager from removing the pop-up menu from the screen display.

The second constructor uses a pre-supplied list of pop-up items to construct the menu.

- $left_{in}$ and top_{in} are the starting position of the menu within its parent window. (The ending position is computed automatically by the UIW_POP_UP_MENU class object according to the size of the menu items.)

- $wnFlags_{in}$ gives information on how to display the items in the pop-up menu. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of the pop-up menu:

  WNF_NO_FLAGS—Does not associate any special flags with the pop-up menu. This flag should not be used in conjunction with any other WNF flags.

  WNF_SELECT_MULTIPLE—Allows more than one menu item to be selected from the menu.

  WNF_NO_WRAP—Prevents the highlight from moving from the bottom item to the top item when the down arrow is pressed.

  WNF_AUTO_SORT—Automatically sorts the menu items in alphabetical order.

- $item_{in}$ is an array of UI_ITEM structures that is used to construct a series of UIW_POP_UP_ITEM objects within the pop-up menu. For more information regarding UI_ITEM structures, see "Chapter 16—UI_ITEM" of this manual.

UIW_POP_UP_MENU::Event

Syntax

```
#include <ui_win.hpp>

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

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

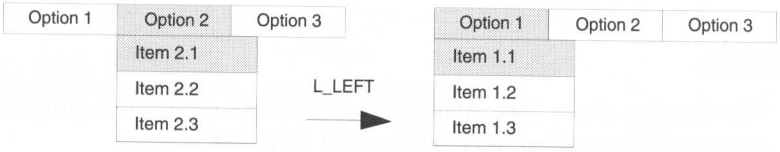
This advanced routine is used to send run-time information to a pop-up menu object. It is declared virtual so that any derived pop-up menu class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the pop-up menu returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified pop-up menu object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event**:

  S_CREATE and S_SIZE—If the pop-up menu receives one of these messages, it will automatically determine the positions of the items in the menu.

  L_LEFT—If the pop-up menu is attached to a pull-down item, the L_LEFT message causes the current pop-up menu to be removed and the pop-up menu to the left appears, if one exists. The figure below illustrates this change:



  L_RIGHT—If the pop-up menu is attached to a pull-down item, the L_RIGHT message causes the current pop-up menu to be removed and the pop-up menu to the right appears, if one exists. The figure below illustrates this change:

All other events are passed by **Event** to **UIW_WINDOW::Event** for processing.

UIW_POP_UP_MENU::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.
- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_POP_UP_MENU, then the request is passed on to UIW_WINDOW. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

GET_FLAGS—Returns the WNF_FLAGS associated with an object if the value in *objectID* is ID_WINDOW or returns the WOF_FLAGS if *objectID* is ID_-WINDOW_OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

SET_FLAGS—Sets the WNF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_WINDOW or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

CLEAR_FLAGS—Clears the WNF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_WINDOW or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

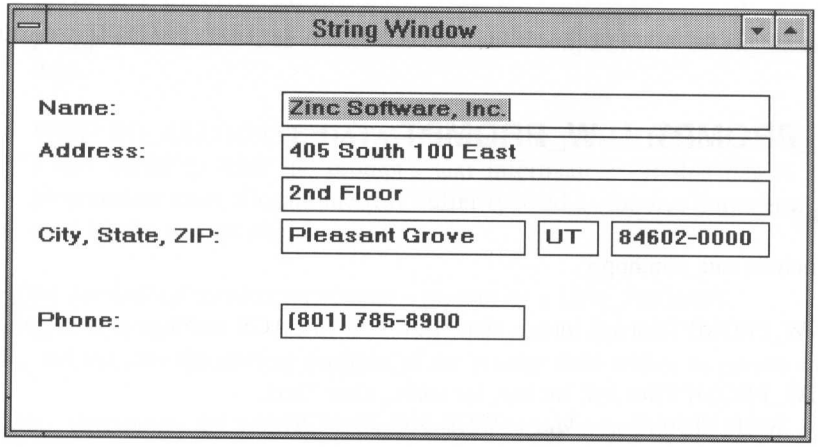
- $objectID_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, then the request will be interpreted by **UIW_WINDOW::-Information( )**.

Example

```
#include <ui_win.hpp>

ExampleFunction()
{
    WNF_FLAGS flags;
    item->Information(GET_FLAGS, &flags, ID_WINDOW);
    .
    .
    .

}
```

CHAPTER 59 – UIW_PROMPT

The UIW_PROMPT class is used to provide lead information about another window object. The picture below shows a graphical implementation of UIW_PROMPT objects (the fields with the ":" character to the right):



The UIW_PROMPT class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_PROMPT : public UI_WINDOW_OBJECT
{
    friend class EXPORT UIF_PROMPT;
public:
    // Members described in UIW_PROMPT reference chapter.
    UIW_PROMPT(int left, int top, char *text,
        WOF_FLAGS woFlags = WOF_NO_FLAGS);
    UIW_PROMPT(int left, int top, int width, char *text,
        WOF_FLAGS woFlags = WOF_NO_FLAGS);
    char *DataGet(void);
    void DataSet(char *text);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_PROMPT(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_PROMPT(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
protected:
    // Members described in UIW_PROMPT reference chapter.
    char *text;

    virtual EVENT_TYPE DrawItem(const UI_EVENT &event, EVENT_TYPE ccode);
};
```

- *UIF_PROMPT* is for use with Zinc Designer. Programmers should not use this part of the class.

- *text* is a pointer to the string passed down in the UIW_PROMPT constructor.

UIW_PROMPT::UIW_PROMPT

Syntax

#include <ui_win.hpp>

UIW_PROMPT(int *left*, int *top*, char **text*, WOF_FLAGS *woFlags* = WOF_NO_FLAGS);
or
UIW_PROMPT(int *left*, int *top*, int *width*, char **text*,
WOF_FLAGS *woFlags* = WOF_NO_FLAGS);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors return a pointer to a new UIW_PROMPT object.

The first overloaded constructor returns a pointer to a UIW_PROMPT.

- $left_{in}$ and top_{in} is the starting position of the prompt field within its parent window. The size of the prompt is determined automatically by the constructor.

- $text_{in}$ is the string representation of the prompt. If the prompt has a '&' character in front of a string character, the character is displayed as a hot key character. At run-time, the field added immediately after the prompt will be made current when the <Alt> key and the hot key character key are pressed simultaneously.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the prompt object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of a UIW_PROMPT class object:

  WOF_BORDER—Draws a single-line border around the prompt, in graphics mode. In text mode, no border is drawn.

  WOF_NO_FLAGS—Does not associate any special window flags with the prompt object. This flag should not be used in conjunction with any other WOF flags.

  WOF_NO_ALLOCATE_DATA—Prevents the prompt object from allocating a text buffer to store the prompt's text information. If this flag is set, the programmer must allocate the text buffer (passed as the *text* parameter) that is used by the prompt object.

The second overloaded constructor returns a pointer to a UIW_PROMPT.

- *left*$_{in}$ and *top*$_{in}$ are the starting position of the prompt field within its parent window.

- *width*$_{in}$ determines the width of the prompt object regardless of the length of *text*.

- *text*$_{in}$ is the string representation of the prompt. If the prompt has a '&' character in front of a string character, the character is displayed as a hot key character. At run-time, the field created immediately after the prompt will be made current when the <Alt> key and the hot key character key are pressed simultaneously.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the prompt object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of a UIW_PROMPT class object:

  WOF_BORDER—Draws a single-line border around the prompt, in graphics mode. In text mode, no border is drawn.

  WOF_JUSTIFY_CENTER—Center-justifies the text information within the prompt field.

  WOF_JUSTIFY_RIGHT—Right-justifies the text information within the prompt field.

  WOF_NO_ALLOCATE_DATA—Prevents the prompt object from allocating a text buffer to store the prompt's text information. If this flag is set, the

programmer must allocate the text buffer (passed as the *text* parameter) that is used by the prompt object.

WOF_NO_FLAGS—Does not associate any special window flags with the prompt object. This flag should not be used in conjunction with any other WOF flags.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window and add it to the window manager.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 62, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Company Information ")
        + new UIW_PROMPT(2, 1, "Name:")
        + new UIW_STRING(11, 1, 40, "", 256)
        + new UIW_PROMPT(2, 2, "Address:")
        + new UIW_STRING(11, 2, 40, "", 256)
        + new UIW_STRING(11, 3, 40, "", 256)
        + new UIW_BUTTON(10, 6, 10, "&Save", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER | WOF_JUSTIFY_CENTER)
        + new UIW_BUTTON(25, 6, 10, "&Cancel", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER | WOF_JUSTIFY_CENTER)
        + new UIW_BUTTON(40, 6, 10, "&Help", BTF_NO_TOGGLE | BTF_AUTO_SIZE,
            WOF_BORDER | WOF_JUSTIFY_CENTER);
    *windowManager + window;
    .
    .
    .

    // The prompt fields will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_PROMPT::DataGet

Syntax

#include <ui_win.hpp>

char *DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns the text information associated with the prompt object.

- *returnValue*$_{out}$ is a pointer to the text information associated with the prompt.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_PROMPT *prompt)
{
    char *text = prompt->DataGet();
    .
    .
    .
}
```

UIW_PROMPT::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(char **text*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function sets the text information associated with the prompt.

- *text*$_{in}$ is a pointer to the new text information to be displayed on the prompt.

Example

```
#include <ui_win.hpp>

ExampleFunction1(UIW_PROMPT *prompt)
{
    .
    .
    .
    prompt->DataSet("&Close");
}
```

UIW_PROMPT::DrawItem

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE DrawItem(const UI_EVENT &*event*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual advanced routine is used to draw a prompt object on the screen. This function is called only if the object's woStatus has the WOS_OWNERDRAW status set.

- $returnValue_{out}$ is a response based on the success of the function call. If successful, the function returns a non-zero value. If the object was not drawn, 0 is returned.

- $event_{in}$ contains a run-time message for the specified prompt object. The prompt is drawn according to the type of event. The following logical events are handled by the **DrawItem( )** routine:

  S_CURRENT, **S_NON_CURRENT**, **S_DISPLAY_ACTIVE** and **S_DISPLAY_INACTIVE**—These messages cause the prompt to be redisplayed. If S_CURRENT or S_NON_CURRENT are passed, the prompt will always be updated. If S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE are passed the prompt will only be updated if *event.region* overlaps the prompt region.

WM_DRAWITEM—A message that causes the prompt to be redrawn. This message is specific to Windows and OS/2.

Expose—A message that causes the prompt to be redrawn. This message is specific to Motif.

- $ccode_{in}$ contains the logical interpretation of *event*.

UIW_PROMPT::Event

Syntax

```
#include <ui_win.hpp>

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

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a prompt object. It is declared virtual so that any derived prompt class can override its default operation.

- $returnValue_{out}$ is a response based on the type of *event*. If successful, the prompt object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- $event_{in}$ contains a run-time message for the specified prompt object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_DISPLAY_ACTIVE and **S_DISPLAY_INACTIVE**—These messages cause the prompt to be redisplayed. The prompt will only be updated if *event.region* overlaps the prompt region.

All other events are passed by **Event( )** to **UI_WINDOW_OBJECT::Event( )** for processing.

UIW_PROMPT::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data, OBJECTID objectID);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_PROMPT, then the request is passed on to UI_WINDOW_OBJECT. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

  CLEAR_FLAGS—Clears the WOF_FLAGS, specified by *data*, associated with an object if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  COPY_TEXT—Returns the *text* associated with the prompt. If this message is used, *data* is the address of a buffer where the prompt's text will be copied. This buffer must be large enough to contain all of the characters associated with the prompt and the terminating NULL character.

GET_FLAGS—Returns the WOF_FLAGS associated with an object if *objectID* is ID_WINDOW_OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined unsigned short.

GET_TEXT—Returns the *text* associated with the prompt. If this message is used, *data* must be the address of a character pointer that will point to the prompt's text string.

SET_FLAGS—Sets the WOF_FLAGS, specified by *data*, associated with an object if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

SET_TEXT—Sets the *text* associated with the prompt. If this message is sent, *data* must be a character pointer to the prompt's new text.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

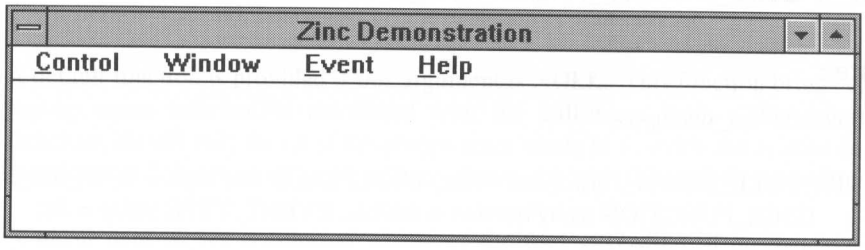
- $objectID_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, then the request will be interpreted by UI_WINDOW_OBJECT.

Example

```
#include <ui_win.hpp>

ExampleFunction()
{
        char string[30];
        prompt->Information(COPY_TEXT, string);
        .
        .
        .

        prompt1->Information(SET_TEXT, "First name:");
        prompt2->Information(SET_TEXT, "Last name:");
        .
        .
        .

}
```

CHAPTER 60 – UIW_PULL_DOWN_ITEM

The UIW_PULL_DOWN_ITEM class is used as the first-level selection within a pull-down menu. The figure below shows a graphical implementation of a pull-down menu containing several UIW_PULL_DOWN_ITEM objects (shown as "Control," "Window," "Event," and "Help"):



The UIW_PULL_DOWN_ITEM class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_PULL_DOWN_ITEM : public UIW_BUTTON
{
    friend class EXPORT UIF_PULL_DOWN_ITEM;
public:
    // Members described in UIW_PULL_DOWN_ITEM reference chapter.
    UIW_POP_UP_MENU menu;

    UIW_PULL_DOWN_ITEM(char *text, WNF_FLAGS wnFlags = WNF_NO_FLAGS,
        USER_FUNCTION userFunction = NULL, EVENT_TYPE value = 0);
    UIW_PULL_DOWN_ITEM(char *text, WNF_FLAGS wnFlags, UI_ITEM *item);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_PULL_DOWN_ITEM(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_PULL_DOWN_ITEM(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);

    // Members described in UI_LIST reference chapter.
    UI_WINDOW_OBJECT *Add(UI_WINDOW_OBJECT *object);
    UI_WINDOW_OBJECT *Subtract(UI_WINDOW_OBJECT *object);
    UIW_PULL_DOWN_ITEM &operator+(UI_WINDOW_OBJECT *object);
    UIW_PULL_DOWN_ITEM &operator-(UI_WINDOW_OBJECT *object);

protected:
    // Members described in UIW_PULL_DOWN_ITEM reference chapter.
    virtual EVENT_TYPE DrawItem(const UI_EVENT &event, EVENT_TYPE ccode);
};
```

- *UIF_PULL_DOWN_ITEM* is for use with Zinc Designer. Programmers should not use this part of the class.

- *menu* is the UIW_POP_UP_MENU that is displayed when the pull-down item is selected.

UIW_PULL_DOWN_ITEM::UIW_PULL_DOWN_ITEM

Syntax

#include <ui_win.hpp>

UIW_PULL_DOWN_ITEM(char **text*, WNF_FLAGS *wnFlags* = WNF_NO_FLAGS,
USER_FUNCTION *userFunction* = NULL, EVENT_TYPE *value* = 0);
or
UIW_PULL_DOWN_ITEM(char **text*, WNF_FLAGS *wnFlags*, UI_ITEM **item*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors return a pointer to a new UIW_PULL_DOWN_ITEM class object.

The first constructor takes the following arguments:

- $text_{in}$ is a pointer to the text information associated with the pull-down item. The text is copied into a buffer allocated by the UIW_PULL_DOWN_ITEM class object.

- $wnFlags_{in}$ gives information on how to display the item's pop-up menu. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of the item's pop-up menu:

  WNF_AUTO_SORT—Automatically sorts the menu items in alphabetical order.

WNF_NO_FLAGS—Does not associate any special flags with the pop-up menu. This flag is the default WNF_FLAG if no other flags are specified and should not be used in conjunction with any other WNF flags.

WNF_NO_WRAP—Prevents the highlight from moving from the bottom item to the top item when the down arrow is pressed.

- *userFunction*$_{in}$ is a programmer-defined function that is called whenever the pull-down item is selected, becomes current or becomes non-current.

  A default function is provided if this argument is NULL. This function brings up the pop-up menu information associated with the pull-down item. This argument, therefore, should only be set if the programmer wants to override the system default function, or if there are no pop-up items associated with this pull-down item.

  A menu item object is selected whenever the user is positioned on the item and presses <Enter>, or when the left mouse button is clicked. The following parameters are passed to *userFunction* when the pull-down item is selected:

  object$_{in}$ is a pointer to the UIW_PULL_DOWN_ITEM class object or to the class object derived from the UIW_PULL_DOWN_ITEM object base class. This argument must be typecast by the programmer.

  event$_{in}$ is a reference pointer to a copy of the event used to reach the programmer defined user function. Since this argument is a copy of the original event, it may be changed by the programmer.

  ccode$_{in}$ is the logical interpretation of *event*.

- *value*$_{in}$ is a unique value that the programmer can associate with a pull-down item. For example, the programmer could associate a value of 0 with an "ok" item and a value of 1 with a "cancel" item. This allows the programmer to define one user-function that looks at the pull-down item values, instead of several user-functions that are tied to each pull-down item.

The second constructor creates a pull-down item with a pre-defined item array.

- *text*$_{in}$ is a pointer to the text information associated with the pull-down item. The string is copied into a buffer allocated by the UIW_PULL_DOWN_ITEM class object.

- *wnFlags*$_{in}$ gives information on how to display the item's pop-up menu. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of the item's pop-up menu:

  WNF_AUTO_SORT—Automatically sorts the menu items in alphabetical order.

  WNF_NO_FLAGS—Does not associate any special flags with the pull-down menu. This flag should not be used in conjunction with any other WNF flags.

  WNF_NO_WRAP—Prevents the highlight from moving from the bottom item to the top item when the down arrow is pressed.

- *item*$_{in}$ is an array that contains UI_ITEM structures used to construct UIW_POP_-UP_ITEM objects within the pull-down item's pop-up menu. For more information regarding the UI_ITEM structure, see "Chapter 16—UI_ITEM" of this manual.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window with pull-down items.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample menus ")
        + &(*new UIW_PULL_DOWN_MENU(1)
            + &(*new UIW_PULL_DOWN_ITEM(" Item&1 ")
                + new UIW_POP_UP_ITEM("Option 1.1")
                + new UIW_POP_UP_ITEM("Option 1.2"))
            + &(*new UIW_PULL_DOWN_ITEM(" Item&2 ")
                + new UIW_POP_UP_ITEM("Option 2.1"))
            + new UIW_PULL_DOWN_ITEM(" Item&3 "));
    *windowManager + window;
    .
    .
    .

    // The pull-down items will automatically be destroyed when the window
    // is destroyed.
}

ExampleFunction2(UI_WINDOW_MANAGER *windowManager)
{
    UI_ITEM item1[] =
    {
        { 11,       NULL,       "Option 1.1", MNIF_NO_FLAGS },
        { 12,       NULL,       "Option 1.2", MNIF_NO_FLAGS },
        { 0,        NULL,       NULL,           0 }
    };

    UI_ITEM item2[] =
    {
        { 21,       NULL,       "Option 2.1", MNIF_NO_FLAGS },
        { 22,       NULL,       "Option 2.2", MNIF_NO_FLAGS },
```

```
    { 0,        NULL,       NULL,           0 }
};

// Create a window with pull-down items.
UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
*window
    + new UIW_BORDER
    + new UIW_TITLE(" Sample menus ")
    + &(*new UIW_PULL_DOWN_MENU(1)
        + new UIW_PULL_DOWN_ITEM(" Item&1 ", WNF_NO_FLAGS, item1)
        + new UIW_PULL_DOWN_ITEM(" Item&2 ", WNF_NO_FLAGS, item2)
        + new UIW_PULL_DOWN_ITEM(" Item&3 "));
*windowManager + window;
.
.
.

// The pull-down items will automatically be destroyed when the window
// is destroyed.
}
```

UIW_PULL_DOWN_ITEM::DrawItem

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE DrawItem(const UI_EVENT &*event*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual advanced routine is used to draw a pull-down item object on the screen. This function is called only if the object's woStatus has the WOS_OWNERDRAW status set.

- $returnValue_{out}$ is a response based on the success of the function call. If successful, the function returns a non-zero value. If the object was not drawn, 0 is returned.

- $event_{in}$ contains a run-time message for the specified pull-down item object. The pull-down item is drawn according to the type of event. The following logical events are handled by the **DrawItem( )** routine:

S_CURRENT, **S_NON_CURRENT**, **S_DISPLAY_ACTIVE** and **S_DISPLAY_INACTIVE**—These messages cause the pull-down item to be redisplayed. If S_CURRENT or S_NON_CURRENT are passed, the pull-down item will always be updated. If S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE are passed the pull-down item will only be updated if *event.region* overlaps the pull-down item region.

WM_DRAWITEM—A message that causes the pull-down item to be redrawn. This message is specific to Windows and OS/2.

Expose—A message that causes the pull-down item to be redrawn. This message is specific to Motif.

- $ccode_{in}$ contains the logical interpretation of *event*.

UIW_PULL_DOWN_ITEM::Event

Syntax

```
#include <ui_win.hpp>

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

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a pull-down item object. It is declared virtual so that any derived pull-down item class can override its default operation.

- $returnValue_{out}$ is a response based on the type of *event*. If successful, the pull-down item object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified pull-down item object. The type of operation depends on the interpreted value for the event. All events are passed by **Event** to **UIW_BUTTON::Event( )** for processing.

UIW_PULL_DOWN_ITEM::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_PULL_DOWN_ITEM, the request is passed on to UIW_BUTTON. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

  GET_NUMBERID_OBJECT—Returns a pointer to the item in the pull-down item's menu whose numberID matches the numberID passed in through the *data* argument. If this message is sent, *data* must be a pointer to a programmer defined unsigned short.

GET_STRINGID_OBJECT—Returns a pointer to the item in the pull-down item's menu whose stringID matches the stringID passed in through the *data* argument. If this message is sent, *data* must be a pointer to a string.

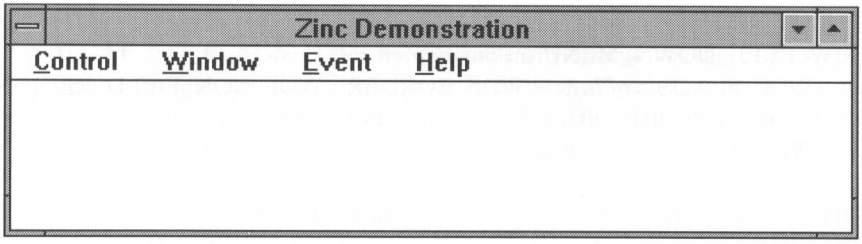
NOTE: Requests that are not handled by **Information( )** are passed on to **UIW_-BUTTON::Information( )**.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

- $objectID_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, the request will be interpreted by UIW_BUTTON.

CHAPTER 61 – UIW_PULL_DOWN_MENU

The UIW_PULL_DOWN_MENU class object is used as a controlling structure for a set of related menu items. The items in this menu are displayed across a single, horizontal line (if more pull-down items are added than can be displayed on a single line, multiple lines will be used.) The figure below shows a graphical implementation of a UIW_PULL_DOWN_MENU class object with four pull-down items (shown as "Control," "Window," "Event," and "Help"):



The public members of the UIW_PULL_DOWN_MENU class (declared in **UI_WIN.-HPP**) are:

```
class EXPORT UIW_PULL_DOWN_MENU : public UIW_WINDOW
{
    friend class EXPORT UIF_PULL_DOWN_MENU;
public:
    // Members described in UIW_PULL_DOWN_MENU reference chapter.
    UIW_PULL_DOWN_MENU(int indentation = 0,  WOF_FLAGS woFlags =
        WOF_BORDER | WOF_NON_FIELD_REGION | WOF_SUPPORT_OBJECT,
        WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS);
    UIW_PULL_DOWN_MENU(int indentation, UI_ITEM *item);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_PULL_DOWN_MENU(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_PULL_DOWN_MENU(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
protected:
    int indentation;
};
```

- *UIF_PULL_DOWN_MENU* is for use with Zinc Designer. Programmers should not use this part of the class.

- *indentation* is the number of cells over from the left edge of the menu where the first menu item should be displayed. The indented space is shown as blank space in the menu.

UIW_PULL_DOWN_MENU::UIW_PULL_DOWN_MENU

Syntax

```
#include <ui_win.hpp>

UIW_PULL_DOWN_MENU(int indentation = 0,
    WOF_FLAGS woFlags = WOF_BORDER | WOF_NON_FIELD_REGION |
    WOF_SUPPORT_OBJECT,
    WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS);
    or
UIW_PULL_DOWN_MENU(int indentation, UI_ITEM *item);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors return a pointer to a new UIW_PULL_DOWN_MENU object.

The first constructor overload returns a pointer to a UIW_PULL_DOWN_MENU object specified by the following arguments:

- $indentation_{in}$ is the indentation level where the first menu item should begin. The indented space is shown as blank space in the menu.

- $woFlags_{in}$ are flags (common to all window objects) that determine the general operation of the pull-down menu. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_PULL_DOWN_-MENU class object:

WOF_BORDER—Draws a single line border around the pull-down menu, in graphics mode. In text mode, no border is drawn.

WOF_NO_FLAGS—Does not associate any special flags with the menu. This flag should not be used in conjunction with any other WOF flags.

WOF_NON_SELECTABLE—Prevents the menu from being selected. If this flag is set, the user will not be able to move within the menu.

- *woAdvancedFlags*$_{in}$ are flags that determine the advanced operation of the menu.

  WOAF_NO_FLAGS—Does not associate any special advanced flags with the menu. Setting this flag allows the user to interact with the menu in a normal fashion. This flag should not be used in conjunction with any other WOAF flags.

The second constructor creates a pull-down menu with a pre-defined item array. These items are used to create UIW_PULL_DOWN_ITEM objects.

- *indentation*$_{in}$ is the indentation level where the first menu item should begin. The indented space is shown as blank space in the window.

- *item*$_{in}$ is an array of UI_ITEM structures that are used to construct a set of pull-down items within the pull-down menu. For more information regarding the use of the UI_ITEM structure, see "Chapter 16—UI_ITEM" in this manual.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    UI_ITEM items[] =
    {
        { 1,        MenuFunction,       "Option 1",     MNIF_NO_FLAGS },
        { 2,        MenuFunction,       "Option 2",     MNIF_NO_FLAGS },
        { 3,        MenuFunction,       "Option 3",     MNIF_NO_FLAGS },
        { 4,        MenuFunction,       "Option 4",     MNIF_NO_FLAGS },
        { 0,        NULL,               NULL,           MNIF_NO_FLAGS}
    };
    // Create a window with pull-down items.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample menus ")
        + new UIW_PULL_DOWN_MENU(1, items);

    *windowManager + window;
```

```
    .
    .
    .
    // The pull-down menu and pull-down items will automatically be destroyed
    // when the window is destroyed.
}
```

UIW_PULL_DOWN_MENU::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a pull-down menu object. It is declared virtual so that any derived pull-down menu class can override its default operation.

- $returnValue_{out}$ is a response based on the type of *event*. If successful, the pull-down menu object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- $event_{in}$ contains a run-time message for the specified pull-down menu object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  L_BEGIN_SELECT and L_DOWN—These messages are sent when the mouse is clicked on a menu item or when the <Down arrow> is pressed. If the cursor is positioned on one of the pull-down menu's items when this occurs, the pop-up menu associated with that item will appear.

  L_CONTINUE_SELECT—Indicates that the cursor has changed positions and that adjustments need to be made. If the cursor has moved to a different pull-

down item, the previous pop-up menu is removed and the menu associated with the current item appears.

S_CREATE—If the pull-down menu receives this message, it will automatically determine the positions of the items in the menu. The width of each item is determined by the amount of space needed to display its text. The height is dynamic, but it is usually one cell in graphics mode and one character in text mode. If all of the items cannot be displayed on one line, the menu will take up two lines.

All other events are passed by **Event( )** to **UIW_WINDOW::Event( )** for processing.

UIW_PULL_DOWN_MENU::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

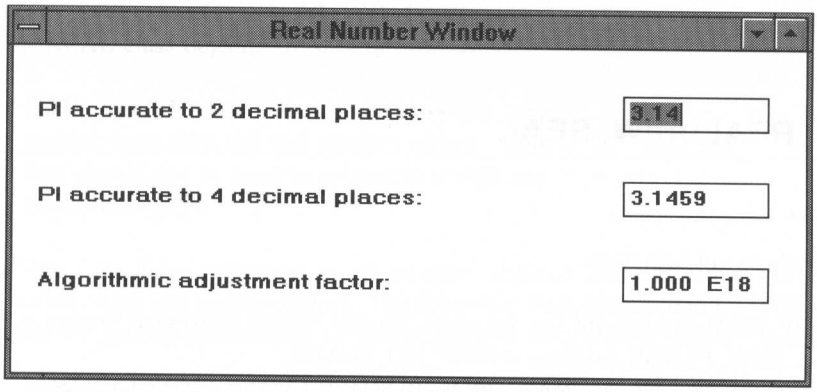
■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object. **Information( )** processes no messages, rather it passes them to **UIW_WINDOW::Information( )** for processing.

CHAPTER 62 – UIW_REAL

The UIW_REAL class is used to display floating-point information to the screen and to collect information from an end user. The UIW_REAL class will display decimal numbers using decimal notation. The figure below shows a graphic implementation of a window with UIW_REAL class objects:



The UIW_REAL class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class UIW_REAL : public UIW_STRING
{
    friend class EXPORT UIF_REAL;
public:
    // Members described in UIW_REAL reference chapter.
    NMF_FLAGS nmFlags;

    UIW_REAL(int left, int top, int width, double *value,
        const char *range = NULL, NMF_FLAGS nmFlags = NMF_NO_FLAGS,
        WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
        USER_FUNCTION userFunction = NULL);
    double DataGet(void);
    void DataSet(double *value);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);
    virtual int Validate(int processError = TRUE);

    // Public members described in UI_WINDOW_OBJECT reference chapter.
    UIW_REAL(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL);
    virtual ~UIW_REAL(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
protected:
    double *number;
    char *range;
};
```

- *UIF_REAL* is used with Zinc Designer only. Programmers should not use this part of the class.
- *number* is used to store the double value for UIW_REAL.
- *range* is a pointer to a string containing the valid range information.

UIW_REAL::UIW_REAL

Syntax

#include <ui_win.hpp>

UIW_REAL(int *left*, int *top*, int *width*, double **value*, const char **range* = NULL,
    NMF_FLAGS *nmFlags* = NMF_NO_FLAGS,
    WOF_FLAGS *woFlags* = WOF_BORDER | WOF_AUTO_CLEAR,
    USER_FUNCTION *userFunction* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_REAL class object.

- $left_{in}$ and top_{in} is the starting position of the real field within its parent window.
- $width_{in}$ is the width of the real field. (The height of the real field is determined automatically by the UIW_REAL class object.)
- $value_{in/out}$ is a double precision floating point real number.

- *range*$_{in}$ is a string that gives all the valid numeric ranges. For example, if a range of "100.0..1000.0" were specified, the UIW_REAL class object would only accept those numeric values that fell between 100.0 and 1000.0. If *range* is NULL, any real number (within the absolute range) is accepted. This string is copied by the UIW_-REAL class object.

- *nmFlags*$_{in}$ gives information on how to display and interpret the numeric information. For formatted numeric input and output, use the UIW_BIGNUM object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of a UIW_REAL class object:

  NMF_NO_FLAGS—Does not associate any special flags with the real number object. This flag should not be used in conjunction with any other NMF flags.

```
10000.2
4345.34
11.195E-40
```

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the real number object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_REAL class object:

  WOF_AUTO_CLEAR—Automatically clears the numeric buffer if the end-user tabs to the real field (from another window field) then presses a key (without first having pressed any movement or editing keys).

  WOF_BORDER—Draws a single line border around the real object, in graphics mode. In text mode, no border is drawn.

  WOF_INVALID—Sets the initial status of the real field to be "invalid." Invalid real numbers fit in the absolute range determined by the object type but do not fulfill all the requirements specified by the program. For example, a real number may initially be set to 200.0, but the final real number, edited by the end-user, must be in the range "10.0..100.0." The initial real number in this example fits the absolute requirements of an unsigned char UIW_REAL class object but does not fit into the specified range.

  WOF_JUSTIFY_CENTER—Center-justifies the numeric information associated with the real object.

  WOF_JUSTIFY_RIGHT—Right-justifies the numeric information associated with the real object.

WOF_NO_ALLOCATE_DATA—Prevents the real object from allocating a numeric value to store the numeric information. If this flag is set, the programmer must allocate the double value (passed as the *value* parameter) that is used by the real object.

WOF_NO_FLAGS—Does not associate any special window flags with the real object. Setting this flag left-justifies the numeric information. This flag should not be used in conjunction with any other WOF flags.

WOF_NON_SELECTABLE—Prevents the real object from being selected. If this flag is set, the user will not be able to edit or position on the numeric information.

WOF_UNANSWERED—Sets the initial status of the real field to be "unanswered." An unanswered real field is displayed as blank space on the screen.

WOF_VIEW_ONLY—Prevents the real number's text field from being modified. However, the field may be made current.

- *userFunction*$_{in}$ is a programmer defined function that is called whenever:

  1—the user moves onto the real field (i.e., S_CURRENT message),

  2—the <ENTER> key is pressed,(i.e., L_SELECT message) or

  3—the user moves to a different field in the window or a different window (i.e., S_NON_CURRENT).

  Validate( ) is not called if *userFunction* exists. If *userFunction* exists, **Validate( )** must be called by the programmer. The following arguments are passed to *userFunction* when a new real is entered:

  object$_{in}$—A pointer to the UIW_REAL class object. This argument must be typecast by the programmer.

  event$_{in}$—The event that caused the user function to be called.

  ccode$_{in}$—The logical or system code that caused the userFunction to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

  L_SELECT—The <ENTER> key was pressed.

S_CURRENT—The real object is about to be edited. This code is sent before any editing operations are permitted.

S_NON_CURRENT—A different field or window has been selected. This code is sent after editing operations have been performed.

The userFunction *returnValue* should be 0 if the field is valid. Otherwise, the programmer should call the error system with an appropriate error message and return -1.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window and add it to the window manager.
    double value = 0.0;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample numbers ")
        + new UIW_PROMPT(2, 1, "Standard:")
        + new UIW_REAL(12, 1, 20, &value, "0..10000")
        + new UIW_PROMPT(2, 2, "Currency:");
    *windowManager + window;
    .
    .
    .

    // The real number fields are automatically destroyed when the window
    // is destroyed.
}
```

UIW_REAL::DataGet

Syntax

#include <ui_win.hpp>

double DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to return the value of the UIW_REAL object.

- *returnValue*$_{out}$ is a double containing the value of the UIW_REAL object.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_REAL *realObject)
{
    double value = realObject->DataGet();
    .
    .
    .

}
```

UIW_REAL::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(double **value*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to set the real value associated with the UIW_REAL object. The field will be re-displayed with the new value.

- *value*$_{in}$ is the new real value.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_REAL *number)
{
    .
    .
    .
    double amount = 100.0;
    number->DataSet(&amount);
}
```

UIW_REAL::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT *&event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a real number object. It is declared virtual so that any derived real number class can override its default operation. The following events are handled by **Event( )**:

S_CURRENT and **S_NON_CURRENT**—These messages update the data displayed on the UIW_REAL field.

Any additional events are passed by **Event( )** to **UIW_STRING::Event( )** for processing.

UIW_REAL::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object. **Information( )** does not process any messages, it simply passes them to **UIW_STRING::Information( )**.

UIW_REAL::Validate

Syntax

```
#include <ui_win.hpp>

virtual int Validate(int processError = TRUE);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

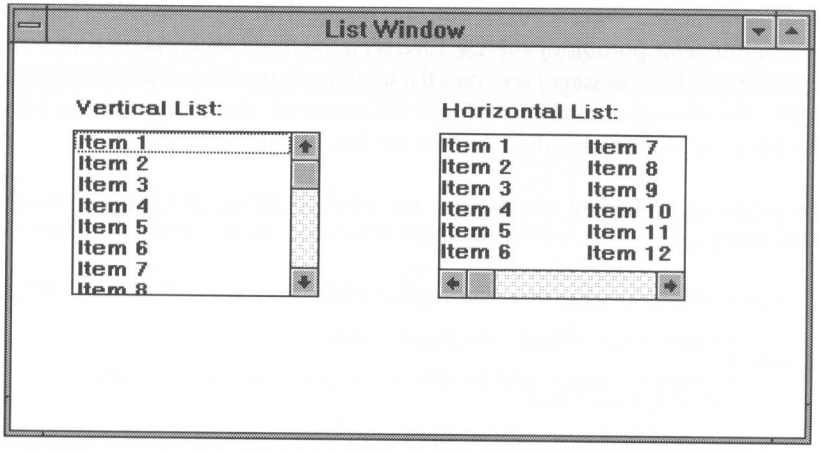
This function allows Zinc Application Framework to validate a newly entered number according to valid real values and with respect to the programmer-specified range.

- *returnValue*$_{out}$ is NMI_OUT_OF_RANGE if the new value is not within the programmer-specified range. NMI_OK is returned if the new value is valid.

- *processError*$_{in}$ is TRUE if the error system should be invoked in the case of an error. Otherwise, *processError* should be set to FALSE.

CHAPTER 63 – UIW_SCROLL_BAR

The UIW_SCROLL_BAR class is used to change displayed information in an associated window, horizontal or vertical list box, or text field so that additional data which is hidden outside of the displayed portion can be seen. The scroll bar should be added directly to the object it is to control. The figure below shows a graphical implementation of a list with a UIW_SCROLL_BAR class object:



Mouse interaction with the scroll bar produces the following effects:

On a vertical scroll bar, clicking on the up arrow (▲) causes the data to scroll up one line.

On a vertical scroll bar, clicking on the down arrow (▼) causes the data to scroll down one line.

On a horizontal scroll bar, clicking on the left arrow (◄) causes the data to scroll left one column.

On a horizontal scroll bar, clicking on the right arrow (►) causes the data to scroll right one column.

On a vertical scroll bar, clicking on empty space above the slider box causes the data to scroll one page up. Likewise, clicking on empty space below the box causes the data to scroll one page down.

On a horizontal scroll bar, clicking on empty space to the left of the slider box causes the data to scroll one page to the left. Likewise, clicking on empty space to the right of the box causes the data to scroll right one page.

Clicking on and dragging the slider box causes the data to scroll to a position proportional to the new slider box position. (The position is changed when the mouse button is released.)

When the vertical scroll bar detects input from the mouse, it sends the **S_VSCROLL** message to the receiving window object (i.e., the field to which it has been added). The *event.scroll.delta* portion of the UI_EVENT structure (which is used in communicating the **S_VSCROLL** message) specifies the total number of lines that the information should scroll. For example, a value of -5 tells the receiving object to scroll the information **up** five lines. A similar operation is done for horizontal scroll bars.

The public and inherited members of the UIW_SCROLL_BAR class (declared in **UI_-WIN.HPP**) are:

```
class EXPORT UIW_SCROLL_BAR : public UIW_WINDOW
{
    friend class EXPORT UIF_SCROLL_BAR;
public:
    // Members described in UIW_SCROLL_BAR reference chapter.
    SBF_FLAGS sbFlags;

    UIW_SCROLL_BAR(int left, int top, int width, int height,
        SBF_FLAGS sbFlags = SBF_VERTICAL, WOF_FLAGS woFlags = WOF_BORDER |
        WOF_SUPPORT_OBJECT | WOF_NON_FIELD_REGION);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

        // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_SCROLL_BAR(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_SCROLL_BAR(void);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);

protected:
    // Protected members described in UIW_SCROLL_BAR reference chapter.
    UI_SCROLL_INFORMATION scroll;
};
```

- *UIF_SCROLL_BAR* is for use with Zinc Designer. Programmers should not use this part of the class.

- *sbFlags* is a variable used to store specific flags associated with a UIW_-SCROLL_BAR class object. The valid SBF_FLAGS are described in the constructor.

- *scroll* contains the current scroll bar position and index information.

UIW_SCROLL_BAR::UIW_SCROLL_BAR

Syntax

#include <ui_win.hpp>

UIW_SCROLL_BAR(int *left*, int *top*, int *width*, int *height*,
SBF_FLAGS *sbFlags* = SBF_VERTICAL, WOF_FLAGS *woFlags* =
WOF_BORDER | WOF_SUPPORT_OBJECT | WOF_NON_FIELD_REGION);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_SCROLL_BAR class object.

- $left_{in}$ and top_{in} is the starting position of the scroll bar within its parent window. These values will be ignored if the WOF_NON_FIELD_REGION flag is set.

- $width_{in}$ is the width of the scroll bar. This value is determined automatically by the UIW_SCROLL_BAR class object when the SBF_VERTICAL flag is set. A value, however, must be entered, even though it will be ignored. When the SBF_HORIZONTAL flag is set, this is the width of the scroll bar. This value will be ignored if the WOF_NON_FIELD_REGION flag is set.

- $height_{in}$ is the height of the scroll bar. This value will be ignored if the WOF_-NON_FIELD_REGION flag is set.

- $sbFlags_{in}$ gives information on how to display the scroll bar.

  SBF_CORNER—defines the scroll bar object to be the corner area between the horizontal and vertical scroll bars. This flag has no effect under Windows or OS/2, since these environments automatically draw the corner area, nor under Motif, since corner scroll bars do not exist in the Motif environment.

SBF_HORIZONTAL—defines the scroll bar object to be a horizontal scroll bar.

SBF_NO_FLAGS—associates no special flags with the scroll bar object. In general, this flag should not be used since the SBF_FLAGS determine the type of scroll bar to create.

SBF_VERTICAL—defines the scroll bar object to be a vertical scroll bar.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general presentation of the scroll bar object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of a UIW_SCROLL_BAR class object:

  WOF_BORDER—Draws a single line border around the scroll bar, in graphics mode. In text mode, no border is drawn. This flag should only be used when the WOF_NON_FIELD_REGION flag is set.

  WOF_NO_FLAGS—Does not associate any special flags with the scroll bar object. This flag should not be used in conjunction with any other WOF flags.

  WOF_NON_FIELD_REGION—Indicates that the scroll bar object is not a form field. If this flag is set, the vertical scroll bar object will occupy the right-most vertical space of its parent window. Horizontal scroll bars will be placed along the bottom-most horizontal space of its parent window.

  WOF_SUPPORT_OBJECT—Indicates that the scroll bar object is to be put into the parent object's support list.

NOTE: Scroll bars are added directly to the object with which they are associated.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Add a text field to the window.
    UIW_WINDOW *window =
        UIW_WINDOW::Generic(0, 0, 40, 10, "Hello World Window");
    *window
        + &(*new UIW_TEXT(0, 0, 0, 0, "Hello, World!", 1024,
            WNF_NO_FLAGS, WOF_NON_FIELD_REGION)
            + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_CORNER)
            + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_VERTICAL)
            + new UIW_SCROLL_BAR(0, 0, 0, 0, SBF_HORIZONTAL));
    *windowManager + window;
    .
    .
    .
```

```
    // The scroll bar will automatically be destroyed when the text field
    // is destroyed.
}
```

UIW_SCROLL_BAR::Event

Syntax

include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a scroll bar object. It is declared virtual so that any derived scroll bar class can override its default operation.

- $returnValue_{out}$ is a response based on the type of *event*. If successful, the scroll bar object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- $event_{in}$ contains a run-time message for the specified scroll bar object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_CREATE—Causes the scroll bar to compute the size and position of the scroll bar with its associated buttons.

  S_HSCROLL—Causes the middle button of the horizontal scroll bar to be re-positioned according to *event.scroll.delta*.

  S_HSCROLL_SET—Causes the middle button of the horizontal scroll bar to be re-positioned according to the *event.scroll.current*, *event.scroll.maximum* and *event.scroll.showing* values.

S_VSCROLL—Causes the middle button of the vertical scroll bar to be re-positioned according to *event.scroll.delta*.

S_VSCROLL_SET—Causes the middle button of the vertical scroll bar to be re-positioned according to the *event.scroll.current*, *event.scroll.maximum* and *event.scroll.showing* values.

All other events are passed by **Event( )** to **UIW_WINDOW::Event( )** for processing.

UIW_SCROLL_BAR::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- $returnValue_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- $request_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_SCROLL_BAR, then the request is passed on to UIW_WINDOW. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

  GET_FLAGS—Returns the SBF_FLAGS associated with an object if *objectID* is ID_SCROLL_BAR. Otherwise, the request is passed on to UI_WINDOW_-

OBJECT to get the WOF_FLAGS. On error, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

SET_FLAGS—Sets the SBF_FLAGS, specified by *data*, associated with an object if *objectID* is ID_SCROLL_BAR. Otherwise, the request is passed on to UI_WINDOW_OBJECT to set the WOF_FLAGS. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

CLEAR_FLAGS—Clears the SBF_FLAGS, specified by *data*, associated with an object if *objectID* is ID_SCROLL_BAR. Otherwise, the request is passed on to UI_WINDOW_OBJECT to clear the WOF_FLAGS. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

All other requests are passed by **Information( )** to **UIW_WINDOW::Information( )** for processing.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

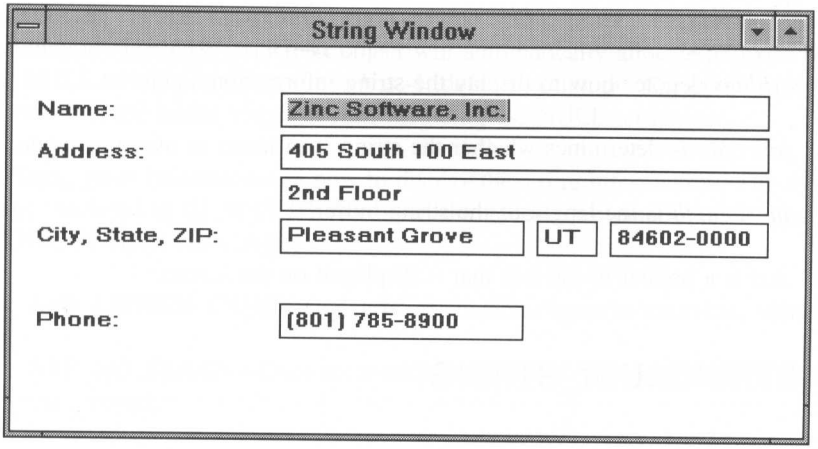
- $objectID_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, then the request will be interpreted by UIW_WINDOW.

Example

```
#include <ui_win.hpp>

ExampleFunction()
{
    SBF_FLAGS flags;
    scrollBar->Information(GET_FLAGS, &flags, ID_SCROLL_BAR);
    .
    .
    .

}
```

CHAPTER 64 – UIW_STRING

The UIW_STRING class is used to display string information to the screen and to collect information, in string form, from an end user. The figure below shows a graphical implementation of a UIW_STRING object:



The public members of the UIW_STRING class (declared in **UI_WIN.HPP**) are:

```
class EXPORT UIW_STRING : public UI_WINDOW_OBJECT
{
    friend class EXPORT UIW_TEXT;
    friend class EXPORT UIF_STRING;
public:
    // Members described in UIW_STRING reference chapter.
    STF_FLAGS stFlags;
    int insertMode;

    UIW_STRING(int left, int top, int width, char *text, int maxLength = -1,
        STF_FLAGS stFlags = STF_NO_FLAGS,
        WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
        USER_FUNCTION userFunction = NULL);
    char *DataGet(void);
    void DataSet(char *text, int maxLength = -1);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_STRING(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_STRING(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
protected:
    // Members described in UIW_STRING reference chapter.
    int maxLength;
    char *text;

    virtual EVENT_TYPE DrawItem(const UI_EVENT &event, EVENT_TYPE ccode);
    char *ParseRange(char *buffer, char *minValue, char *maxValue);
};
```

- *UIF_STRING* is for use with Zinc Designer. Programmers should not use this part of the class.
- *stFlags* denotes how to display the string information.
- *insertMode* determines whether the string is in insert or overtype mode.
- *maxLength* is the length of the string buffer.
- *text* is a pointer to the text that is displayed on the screen.

UIW_STRING::UIW_STRING

Syntax

#include <ui_win.hpp>

UIW_STRING(int *left*, int *top*, int *width*, char **text*, int *maxLength* = -1,
    STF_FLAGS *stFlags* = STF_NO_FLAGS,
    WOF_FLAGS *woFlags* = WOF_BORDER | WOF_AUTO_CLEAR,
    USER_FUNCTION *userFunction* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_STRING class object.

- $left_{in}$ and top_{in} are the starting position of the string field within its parent window.

- *width*$_{in}$ is the width of the string field. (The height of the string field is determined automatically by the UIW_STRING class object.)

- *text*$_{in}$ is a pointer to the initial text to be displayed to the screen. The text is copied into a buffer, allocated by the UIW_STRING object, which is *maxLength* + 1 characters in length.

- *maxLength*$_{in}$ is the maximum length of the string buffer, excluding the NULL terminator. The UIW_STRING object will automatically allocate an extra byte for the NULL terminator. If *maxLength* is -1 (default value), the size of the string buffer allocated is the initial length of *text*, including the NULL terminator.

- *stFlags*$_{in}$ gives information on how to display the string information. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of a UIW_STRING class object:

  STF_LOWER_CASE—Converts all character input to lowercase values.

  STF_NO_FLAGS—Does not associate any special flags with the UIW_STRING class object.

  STF_PASSWORD—Characters entered into the string field are not echoed to the screen. Instead, the "." or " " (space) character is printed for each character typed, depending on the environment.

  STF_UPPER_CASE—Converts all character input to uppercase values.

  STF_VARIABLE_NAME—Converts the space character to an underscore value.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the string object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_STRING class object:

  WOF_AUTO_CLEAR—Automatically clears the string buffer if the end-user tabs to the string field (from another window field) then presses a key (without first having pressed any movement or editing keys).

  WOF_BORDER—In graphics mode, setting this flag draws a single line border around the string object. In text mode, no border is drawn.

WOF_INVALID—Sets the initial status of the string field to be "invalid." By default, all string information is valid. A programmer may specify a string field as invalid by setting this flag upon creation of the string object or by re-setting the flag through the *userFunction* function (discussed below). For example, a string field may initially be set to be blank, but the final string edited by the end-user must contain some instructional information. In this case the initial string information does not fulfill the programmer's requirements.

WOF_JUSTIFY_CENTER—Center-justifies the string information within the string field.

WOF_JUSTIFY_RIGHT—Right-justifies the string information within the string field.

WOF_NO_ALLOCATE_DATA—Prevents the string object from allocating a text buffer to store the text information. If this flag is set, the programmer must allocate the text buffer (passed as the *text* parameter) that is used by the string object. The text buffer should be *maxLength* + 1 (for the NULL terminator) characters.

WOF_NO_FLAGS—Does not associate any special flags with the string object. In this case, the string buffer will be left-justified. This flag should not be used in conjunction with any other WOF flags.

WOF_NON_SELECTABLE—Prevents the string object from being selected. If this flag is set, the end-user will not be able to edit or position on the string information.

WOF_UNANSWERED—Sets the initial status of the string field to be "unanswered." An unanswered string field is displayed as blank space on the screen.

WOF_VIEW_ONLY—Prevents the string from being edited. If this flag is set, the end-user will not be able to edit the string information but will be able to browse through the string.

- *userFunction*$_{in}$ is a programmer-defined function that is called whenever:

  1—the user moves onto the string field (i.e., S_CURRENT message),

  2—the <ENTER> key is pressed (i.e., L_SELECT message) or

3—the user moves to a different field in the window or a different window (i.e., S_NON_CURRENT message).

The following arguments are passed to *userFunction* when string information is entered:

object$_{in}$—A pointer to the UIW_STRING class object or to the class object derived from the UIW_STRING object base class. This argument must be typecast by the programmer.

event$_{in}$—The event that caused the user function to be called.

ccode$_{in}$—The logical or system code that caused the user function to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

L_SELECT—This message is sent when the <ENTER> key is pressed.

S_CURRENT—The string object is about to be edited. This code is sent before any editing operations are permitted.

S_NON_CURRENT—A different field or window has been selected. This code is sent after editing operations have been performed.

The user function's *returnValue* should be 0 if the string is valid. Otherwise, the programmer should call the error system with an appropriate error message and return -1.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Add a string field to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_STRING(10, 1, 20, "Sample string", 256);
    .
    .
    .

    // The string object will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_STRING::DataGet

Syntax

#include <ui_win.hpp>

char *DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns the text information associated with the string object.

- *returnValue*$_{out}$ is a pointer to the text information associated with the string.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_STRING *string)
{
    char *text = string->DataGet();
    .
    .
    .

}
```

UIW_STRING::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(char **text*, int *maxLength* = -1);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function sets the text information associated with the string. The field will be re-displayed with the new text.

- $text_{in}$ is a pointer to the new text information to be displayed on the string.
- $maxLength_{in}$ is the number of characters to allocate for the string buffer. If *maxLength* is -1, the function uses the previously allocated buffer. If *maxLength* is greater than the previous string's length, a new buffer of size *maxLength* + 1 (for the NULL terminator) is allocated. Otherwise, the previous buffer is used.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_STRING *string)
{
    .
    .
    .

    string->DataSet("Hello World!");
}
```

UIW_STRING::DrawItem

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE DrawItem(const UI_EVENT &*event*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual advanced routine is used to draw a string object on the screen. This function is called only if the object's woStatus has the WOS_OWNERDRAW status set.

- *returnValue*$_{out}$ is a response based on the success of the function call. If successful, the function returns a non-zero value. If the object was not drawn, 0 is returned.

- *event*$_{in}$ contains a run-time message for the specified string object. The string is drawn according to the type of event. The following logical events are handled by the **DrawItem( )** routine:

  S_CURRENT, **S_NON_CURRENT**, **S_DISPLAY_ACTIVE** and **S_DISPLAY_INACTIVE**—These messages cause the string to be redisplayed. If S_CURRENT or S_NON_CURRENT are passed, the string will always be updated. If S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE are passed the string will only be updated if *event.region* overlaps the string region.

  WM_DRAWITEM—A message that causes the string to be redrawn. This message is specific to Windows and OS/2.

  Expose—A message that causes the string to be redrawn. This message is specific to Motif.

- *ccode*$_{in}$ contains the logical interpretation of *event*.

UIW_STRING::Event

Syntax

```
#include <ui_win.hpp>

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

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a string object. It is declared virtual so that any derived string class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the string object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified string object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **UIW_STRING::Event( )**:

  E_KEY—This message is sent whenever a alphanumeric key is pressed that does not directly map to a logical event. When **Event( )** receives the E_KEY message, it checks to see if the key pressed is valid for the string field. If it is valid, the character associated with it is inserted into the string field.

  L_BEGIN_MARK, L_CONTINUE_MARK and L_END_MARK—These messages communicate the progress of a mark operation. L_BEGIN_MARK indicates the starting position of the marked region, L_CONTINUE_MARK indicates the growth or decrease of the marked region, and L_END_MARK indicates the end of the marking operation. Using a mouse, for example, L_BEGIN_MARK indicates that the mouse button has been pressed, L_CONTINUE_MARK indicates that the mouse is currently being dragged with the left button depressed, and L_END_MARK indicates that the mouse button has been released. If the cursor enters a previously marked region and the mouse button is pressed again, the highlight on the previous mark disappears and a new mark is started from the current cursor position.

  L_BOL—This message causes the cursor to move to the beginning of the line, or to the first editable character or space of the string field. For example, where the underscore represents the cursor position, the string Stand and be counted would change to Stand and be counted as a result of the L_BOL message. If the mark feature is on, L_BOL extends the marked region to the beginning of the line.

  L_COPY_MARK—This message causes the marked region to be copied. The copy is stored in the global paste buffer.

  L_CUT—This message cuts the entire string. The cut region is stored in the global paste buffer.

L_CUT_PASTE and L_PASTE—These messages cause the contents of the global paste buffer to be pasted into the string field at the cursor position. For example, if the contents of the paste buffer were up, the string Stand_and be counted would change to Stand up and be counted. If the contents are too large for the string field, only that portion which fits will be pasted. For example, if a thousand characters were stored in the paste buffer and the programmer had specified a limit of 100 characters for the string field, the first 100 characters of the buffer would be pasted into the string field.

L_CUT_PASTE differs slightly from L_PASTE in that it is a toggle. If no region is marked, it acts like L_PASTE and pastes the contents of the global paste buffer into the field. If a region is marked, it causes the entire field to be cut and stored in the global paste buffer.

L_DELETE—This message causes the character at the cursor position to be deleted. For example, where the underscore represents the cursor position, the string Stand and be counted would change to Stand ad be counted as a result of the L_DELETE message.

L_DELETE_EOL—This message causes all characters from the current cursor position to the end of the line to be deleted. For example, where the underscore represents the cursor position, the string Stand and be counted would change to Stand a_ as a result of the L_DELETE_EOL message.

L_DELETE_WORD—This message causes the word at the cursor position to be deleted, along with any trailing spaces. For example, where the underscore represents the cursor position, the string Stand and be counted would change to Stand be counted as a result of the L_DELETE_WORD message.

L_EOL—This message causes the cursor to move to the end of the line, or to the last editable character or space of the string field. For example, where the underscore represents the cursor position, the string Stand and be counted would change to Stand and be counted_ as a result of the L_EOL message. If the mark feature is on, L_EOL extends the marked region to the end of the line.

L_INSERT_TOGGLE—This message causes the insert mode to toggle. If insert is the current mode, any entered character will be inserted into the string at the cursor position. For instance, if the character 'n' were entered, the string Stand ad be counted would change to Stand and be counted.

If overstrike is the current mode, any entered character will replace the character at the cursor position. For instance, if the character 'd' were entered, the string Stand and be counted would change to Stand add be counted.

L_LEFT—This message causes the cursor to move one character or space to the left of its current position, if it is not in the field's first editable position. If the mark feature is on, L_LEFT extends the marked region to include the character.

L_MARK—This message causes the mark capability to toggle to on or off.

L_RIGHT—This message causes the cursor to move one character or space to the right, if it is not in the field's last editable position. If the mark feature is on, L_RIGHT extends the marked region to include the character.

L_WORD_LEFT—This message causes the cursor to move to the beginning of the previous word or to the beginning of the same word if the cursor is positioned in the middle of that word. For example, where the underscore represents the cursor position, the string Stand and be counted would change to Stand and be counted, as a result of the L_WORD_LEFT message.

L_WORD_RIGHT—This message causes the cursor to move to the beginning of the next word. For example, where the underscore represents the cursor position, the string Stand and be counted would change to Stand and be counted as a result of the L_WORD_RIGHT message.

S_CURRENT and **S_NON_CURRENT**—These messages cause the programmer-specified userFunction to be called, if one exists. The programmer will get a pointer to the UIW_STRING class, and the value for the userFunction *ccode* will be S_CURRENT or S_NON_CURRENT.

All other events are passed by **Event( )** to **UI_WINDOW_OBJECT::Event( )** for processing.

UIW_STRING::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_STRING, then the request is passed on to UI_WINDOW_OBJECT. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

  CLEAR_FLAGS—Clears the STF_FLAGS, specified by *data*, associated with an object if *objectID* is ID_STRING. Otherwise, the request is passed to UI_-WINDOW_OBJECT to clear the WOF_FLAGS. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  COPY_TEXT—Returns the *text* associated with the string. If this message is used, *data* is the address of a buffer where the string's text will be copied. This buffer must be large enough to contain all of the characters associated with the string and the terminating NULL character.

  GET_FLAGS—Returns the STF_FLAGS associated with an object if *objectID* is ID_STRING. Otherwise, the request is passed on to UI_WINDOW_OBJECT to get the WOF_FLAGS. On error, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  GET_TEXT—Returns the *text* associated with the string. If this message is used, *data* must be the address of a character pointer that will point to the string's text string.

  GET_TEXT_LENGTH—Returns the length of the string field's text buffer.

SET_FLAGS—Sets the STF_FLAGS, specified by *data*, associated with an object if *objectID* is ID_STRING. Otherwise, the request is passed on to UI_-WINDOW_OBJECT to set the WOF_FLAGS. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

SET_TEXT—Sets the *text* associated with the string. If this message is sent, *data* must be a character pointer to the string's new text. The field will be re-displayed with the new text.

SET_TEXT_LENGTH—Sets the length of the string field's text buffer. If this message is sent, *data* must be a pointer to an integer containing the new length of the string's text buffer.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

- $objectID_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, then the request will be interpreted by UI_WINDOW_OBJECT.

Example

```
#include <ui_win.hpp>

ExampleFunction()
{
    char *text;
    string->Information(GET_TEXT, &text);
    string1->Information(SET_TEXT, "First name");
    .
    .
    .

}
```

UIW_STRING::ParseRange

Syntax

#include <ui_win.hpp>

char *ParseRange(char **buffer*, char **minValue*, char **maxValue*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to parse one of the ranges passed in within an object derived from UIW_STRING.

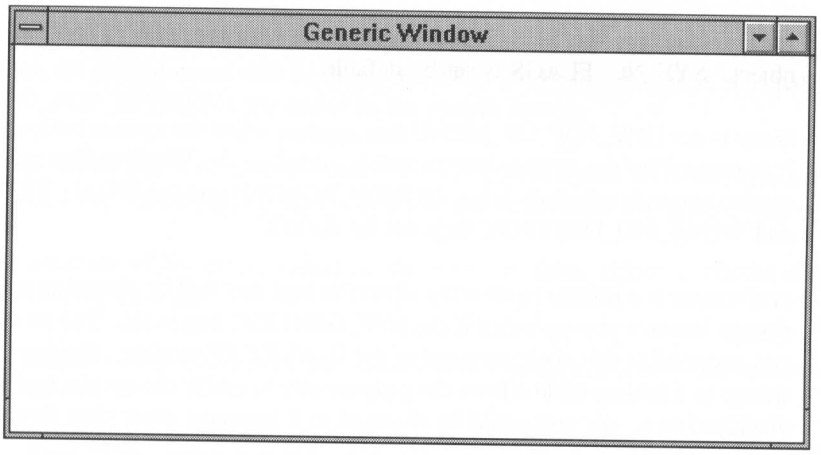
- *returnValue*$_{out}$ indicates the new position of the parsing value.
- *buffer*$_{in}$ is the range value.
- *minValue*$_{out}$ and *maxValue*$_{out}$ are the minimum and maximum values that are returned.

Example

```
#include <ui_win.hpp>

int UIW_DATE::Validate(int processError = TRUE)
{
    .
    .
    .

    // Check for an absolute date error.
    UI_DATE currentDate;
    char *stringDate = (char *)UIW_STRING::Information(GET_TEXT, NULL);
    DTI_RESULT errorCode = currentDate.Import(stringDate, dtFlags);

    // Check for a range error.
    if (range && errorCode == DTI_OK)
        errorCode = DTI_OUT_OF_RANGE;
    for (char *tRange = range; tRange && errorCode == DTI_OUT_OF_RANGE; )
    {
        char minDate[64], maxDate[64];
        tRange = ParseRange(tRange, minDate, maxDate);
        if (currentDate >= UI_DATE(minDate, rangeFlags) &&
            currentDate <= UI_DATE(maxDate, rangeFlags))
            errorCode = DTI_OK;
    }
    .
    .
    .

    return (errorCode);
}
```

CHAPTER 65 – UIW_SYSTEM_BUTTON

The UIW_SYSTEM_BUTTON class is used to select general operations on a window (e.g., size, move, maximize, minimize). In addition, if the end user double-clicks on the system button, it causes the window to be closed. The figure below shows a graphical implementation of a UIW_SYSTEM_BUTTON class object (the button with the '–' character in DOS, Motif and Windows or the minimize icon in OS/2):



The UIW_SYSTEM_BUTTON class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_SYSTEM_BUTTON : public UIW_BUTTON
{
    friend class EXPORT UIF_SYSTEM_BUTTON;
public:
    SYF_FLAGS syFlags
    UIW_POP_UP_MENU menu;
    static char **sysPrompts;

    // Members described in UIW_SYSTEM_BUTTON reference chapter.
    UIW_SYSTEM_BUTTON(SYF_FLAGS syFlags = SYF_NO_FLAGS);
    UIW_SYSTEM_BUTTON(UI_ITEM *item);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    static UIW_SYSTEM_BUTTON *Generic(void);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_SYSTEM_BUTTON(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_SYSTEM_BUTTON(void);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);

    // Members described in UI_LIST reference chapter.
    UI_WINDOW_OBJECT *Add(UI_WINDOW_OBJECT *object);
    UI_WINDOW_OBJECT *Subtract(UI_WINDOW_OBJECT *object);
    UIW_SYSTEM_BUTTON &operator+(UI_WINDOW_OBJECT *object);
    UIW_SYSTEM_BUTTON &operator-(UI_WINDOW_OBJECT *object);
};
```

- *UIF_SYSTEM_BUTTON* is for use with Zinc Designer. Programmers should not use this part of the class.

- *syFlags* contains the SYF_FLAGS associated with the UIW_SYSTEM_BUTTON object. SYF_NO_FLAGS is set by default.

- *menu* is the UIW_POP_UP_MENU that appears when the system button is selected. It is created by the system button and is added to the Window Manager when the system button is selected. It has the WOF_BORDER and the WOAF_TEMPORARY and WOAF_NO_DESTROY flags set by default.

- *sysPrompts* is a pointer to an array of text strings that will be placed as options in the system button's pop-up menu if the SYF_GENERIC flag is set. The array is defined and assigned to this static member in the **G_SYS.CPP** module. Placing these literal strings in a lookup table allows the programmer to easily change the text that will be displayed (e.g., the text could be changed to a language other than English).

UIW_SYSTEM_BUTTON::UIW_SYSTEM_BUTTON

Syntax

#include <ui_win.hpp>

UIW_SYSTEM_BUTTON(SYF_FLAGS *syFlags* = SYF_NO_FLAGS);
or
UIW_SYSTEM_BUTTON(UI_ITEM **item*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors return a pointer to a new UIW_SYSTEM_BUTTON class object.

The first overloaded constructor creates a UIW_SYSTEM_BUTTON.

- *syFlags*$_{in}$ contains the SYF_FLAGS to be associated with the UIW_SYSTEM_-BUTTON. The following SYF_FLAGS are accepted:

  SYF_NO_FLAGS—Associates no special flags with the system button. If this flag is set, no pop-up menu will be displayed when the system button is selected, unless UIW_POP_UP_ITEMs are added to the system button.

  SYF_GENERIC—Creates a generic system button menu with the system button object. The following pop-up item entries are included in the system button menu:

  Restore—This option restores the window from either a maximized or a minimized state.

  Move—This option puts the window into a mode that allows the window to be moved.

  Size—This option puts the window into a mode that allows the window to be sized.

  Minimize—This option minimizes the window.

  Maximize—This option maximizes the window.

  Close—This option closes the window.

The second overloaded constructor creates a UIW_SYSTEM_BUTTON with UIW_POP_-UP_ITEM objects within the system menu.

- *item*$_{in}$ is an array of UI_ITEM structures that will be used to create the UIW_POP_-UP_ITEM structures for the system button's menu. For more information regarding the UI_ITEM structure, see "Chapter 16—UI_ITEM" of this manual.

The system button object always occupies the outer-most left corner space available in the parent window. To ensure that the system button is drawn correctly, it must be added right after the UIW_MINIMIZE_BUTTON class object. The following example shows the correct and incorrect order of system button creation:

```
1)  // CORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1")
        .
        .
        .

2)  // INCORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_SYSTEM_BUTTON
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_TITLE("Window 1")
        + new UIW_BORDER
        .
        .
        .
```

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a new window and attach it to the window manager.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + &(*new UIW_SYSTEM_BUTTON
            + new UIW_POP_UP_ITEM("&Move", MNIF_MOVE)
            + new UIW_POP_UP_ITEM("&Size", MNIF_SIZE)
            + new UIW_POP_UP_ITEM
            + new UIW_POP_UP_ITEM("&Close", MNIF_CLOSE))
        + new UIW_TITLE("Window 1");
    *windowManager + window;

    // The system button will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_SYSTEM_BUTTON::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced function is used to send run-time information to a system button object. It is declared virtual so that any derived system button class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the system button object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified system button object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_CREATE and **S_SIZE**—Either of these messages will cause **Event( )** to recalculate the system button's position and size within its parent window.

All other events are passed by **Event( )** to **UIW_BUTTON::Event( )** for processing.

UIW_SYSTEM_BUTTON::Generic

Syntax

```
#include <ui_win.hpp>

static UIW_SYSTEM_BUTTON *Generic(void);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function creates a system button, including an associated menu with the following options: Restore, Move, Size, Minimize, Maximize and Close.

Example

```
#include <ui_win.hpp>

UIW_WINDOW *UIW_WINDOW::Generic(int left, int top, int width, int height,
    char *title, UI_WINDOW_OBJECT *minObject, WOF_FLAGS woFlags,
    WOAF_FLAGS woAdvancedFlags, int _helpContext)
{
    UIW_WINDOW *window = new UIW_WINDOW(left, top, width, height, _icon,
        woFlags, woAdvancedFlags, _helpContext);

    // Add default window objects.
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + UIW_SYSTEM_BUTTON::Generic()
        + new UIW_TITLE(title);

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

UIW_SYSTEM_BUTTON::Information

Syntax

#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST *request*, void **data*,
    OBJECTID *objectID* = 0);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_SYSTEM_BUTTON, the request is passed on to UIW_BUTTON. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

  GET_NUMBERID_OBJECT—Returns a pointer to the pop-up item in the system menu whose numberID matches the numberID passed in through the *data* argument. If this message is sent, *data* must be a pointer to a programmer defined unsigned short. This request is passed on to **UI_WINDOW_OBJECT::Information( )** for processing.

  GET_STRINGID_OBJECT—Returns a pointer to the pop-up item in the system menu whose stringID matches the stringID passed in through the *data* argument. If this message is sent, *data* must be a pointer to a string. This request is passed on to **UI_WINDOW_OBJECT::Information( )** for processing.

  GET_FLAGS—Returns the SYF_FLAGS associated with the object if the value in *objectID* is ID_SYSTEM_BUTTON, BTF_FLAGS if *objectID* is ID_BUTTON, or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  SET_FLAGS—Sets the SYF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_SYSTEM_BUTTON, BTF_FLAGS if *objectID* is ID_BUTTON, or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  CLEAR_FLAGS—Clears the SYF_FLAGS, specified by *data*, associated with an object if the value in *objectID* is ID_SYSTEM_BUTTON, BTF_FLAGS if *objectID* is ID_BUTTON, or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

- *data*$_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

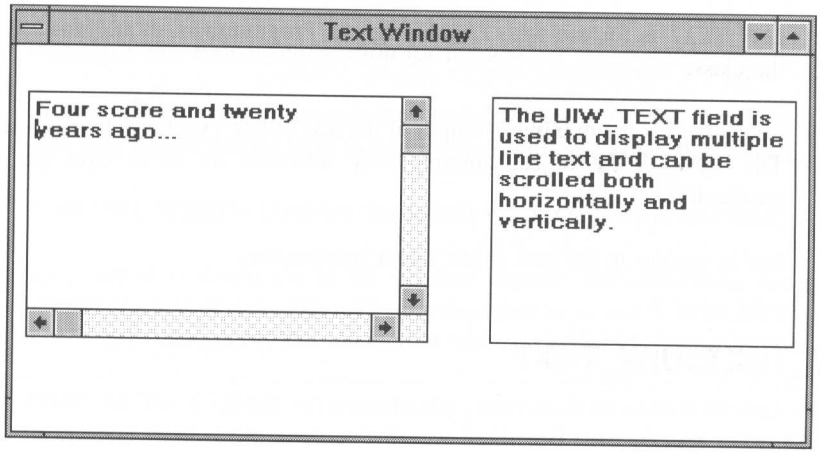
- *objectID*$_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, then the request will be interpreted by UIW_BUTTON.

Example

```
#include <ui_win.hpp>

ExampleFunction()
{
    UIF_FLAGS flags;
    systemButton->Information(GET_FLAGS, &flags);
    .
    .
    .
}
```

CHAPTER 66 – UIW_TEXT

The UIW_TEXT class is used to display multiple-line text information to the screen and to collect information, in string form, from an end user. The figure below shows graphical implementations of a UIW_TEXT class object:



The UIW_TEXT class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_TEXT : public UIW_WINDOW
{
    friend class EXPORT UIF_TEXT;
public:
    int insertMode;

    // Members described in UIW_TEXT reference chapter.
    UIW_TEXT(int left, int top, int width, int height, char *text,
        int maxLength = -1, WNF_FLAGS wnflags = WNF_NO_WRAP,
        WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
        USER_FUNCTION userFunction = NULL);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    char *DataGet(void);
    void DataSet(char *text, int maxLangth = -1);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_TEXT(const char *name, UI_STORAGE *file, UI_STORAGE_OBJECT *object);
    virtual ~UIW_TEXT(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
protected:
    // Members described in UIW_TEXT reference chapter.
    int maxLength;
    char *text;

    virtual EVENT_TYPE DrawItem(const UI_EVENT &event, EVENT_TYPE ccode);
};
```

- *insertMode* determines whether the text object is in insert or overtype mode.
- *UIF_TEXT* is for use with Zinc Designer. Programmers should not use this part of the class.
- *maxLength* is the maximum length of the text buffer excluding the NULL terminator. The UIW_TEXT object automatically allocates an extra byte for the NULL terminator.
- *text* is pointer to the text object's text information.

UIW_TEXT::UIW_TEXT

Syntax

#include <ui_win.hpp>

UIW_TEXT(int *left*, int *top*, int *width*, int *height*, char **text*, int *maxLength* = -1,
    WNF_FLAGS *wnFlags* = WNF_NO_WRAP,
    WOF_FLAGS *woFlags* = WOF_BORDER | WOF_AUTO_CLEAR,
    USER_FUNCTION *userFunction* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_TEXT class object.

- $left_{in}$ and top_{in} are the starting position of the text field within its parent window.
- $width_{in}$ and $height_{in}$ are the size of the text field within the parent window.

- $text_{in}$ is a pointer to the initial text to be displayed within the text field. Unless the **WOF_NO_ALLOCATE_DATA** flag is set, *text* is copied into a buffer, allocated by the UIW_TEXT object, which is *maxLength* + 1 (for the NULL terminator) in length.

- $maxLength_{in}$ is the maximum length of the text buffer, excluding the NULL terminator. The UIW_TEXT object automatically allocates an extra character for the NULL terminator.

- $wnFlags_{in}$ specifies format information for the text.

  WNF_NO_FLAGS—Does not associate any special flags with the text object. This flag should not be used in conjunction with any other WNF flags.

  WNF_NO_WRAP—Disables the default word wrap in the text field.

- $woFlags_{in}$ are flags (common to all window objects) that determine the general operation of the text object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_TEXT class object:

  WOF_AUTO_CLEAR—Automatically clears the text buffer if the end-user tabs to the text field (from another window field) and then presses a key (without having previously pressed any movement or editing keys).

  WOF_BORDER—Draws a single line border around the text object, in graphics mode. In text mode, no border is drawn.

  WOF_INVALID—Sets the initial status of the text field to be "invalid." By default, all text information is valid. For example, a text field may initially be set to be blank, but the final text field edited by the end-user must contain some instructional text. In this case the initial text information does not fulfill the programmer's requirements.

  WOF_NO_ALLOCATE_DATA—Prevents the text object from allocating a text buffer to store the text information. If this flag is set, the programmer must allocate the text buffer (passed as the *text* parameter) that is used by the text object.

  WOF_NO_FLAGS—Does not associate any special flags with the text object. This flag should not be used in conjunction with any other WOF flags.

  WOF_NON_FIELD_REGION—The text object is not a normal form field. If this flag is set and the text is attached to a higher-level window, then the *left*,

top, *width* and *height* arguments are ignored and the text occupies any remaining space within the parent window.

WOF_NON_SELECTABLE—Prevents the text object from being selected. If this flag is set, the user will not be able to edit nor move within the text field.

WOF_UNANSWERED—Sets the initial status of the text field to be "unanswered." An unanswered text field is displayed as blank space on the screen.

WOF_VIEW_ONLY—The text object cannot be edited. If this flag is set, the end-user will not be able to edit the text information but will be able to browse through the text field.

- *userFunction*$_{in}$ is a programmer defined function that is called whenever:

  1—the user moves onto the text field (i.e., S_CURRENT message),

  2—the user moves to a different field in the window or a different window on the screen (i.e., S_NON_CURRENT message).

The following arguments are passed to *userFunction* when text information is entered:

object$_{in}$—A pointer to the UIW_TEXT object or to the class object derived from the UIW_TEXT object base class. This argument must be typecast by the programmer.

event$_{in}$—The event that caused the user function to be called.

ccode$_{in}$—The logical or system code that caused the user function to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

S_CURRENT—The text object is about to be edited. This code is sent before any editing operations are permitted.

S_NON_CURRENT—A different field, or window, has been selected. This code is sent after editing operations have been performed.

The userFunction's *returnValue* should be 0 if the text is valid. Otherwise, the programmer should call the error system with an appropriate error message and return -1.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Add a text field to the window.
    UIW_WINDOW *window =
        UIW_WINDOW::Generic(0, 0, 40, 10, "Hello World Window");
    *window
        + new UIW_TEXT(0, 0, 0, 0, "Hello, World!", 1024,
            WNF_NO_WRAP, WOF_NON_FIELD_REGION);
    *windowManager + window;
    .
    .
    .

    // The text object will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_TEXT::DataGet

Syntax

#include <ui_win.hpp>

char *DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to return the text information associated with the text object.

- *returnValue*$_{out}$ is a pointer to the text information associated with the text object.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_TEXT *text1, UIW_TEXT *text2)
{
    .
    .
    .

    char *textData = text1->DataGet();
    text2->DataSet(textData);
}
```

UIW_TEXT::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(char **text*, int *maxLength* = -1);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to set the text information associated with the text object. The field will be re-displayed with the new text.

- $text_{in}$ is a pointer to the new text information to be displayed on the text field
- $maxLength_{in}$ is the maximum length of the text object. This value will update the text object's *maxLength* value and re-allocate a buffer, if necessary. If -1 is entered, the text object's *maxLength* value remains unchanged.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_TEXT *text1, UIW_TEXT *text2)
{
    .
    .
    .
```

```
    char *textData = text1->DataGet();
    text2->DataSet(textData);
}
```

UIW_TEXT::DrawItem

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE DrawItem(const UI_EVENT *&event*, EVENT_TYPE *ccode*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This virtual advanced routine is used to draw a text object on the screen. This function is called only if the object's woStatus has the WOS_OWNERDRAW status set.

- $returnValue_{out}$ is a response based on the success of the function call. If successful, the function returns a non-zero value. If the object was not drawn, 0 is returned.

- $event_{in}$ contains a run-time message for the specified text object. The text is drawn according to the type of event. The following logical events are handled by the **DrawItem( )** routine:

  S_CURRENT, **S_NON_CURRENT**, **S_DISPLAY_ACTIVE** and **S_DISPLAY_INACTIVE**—These messages cause the text to be redisplayed. If S_CURRENT or S_NON_CURRENT are passed, the text will always be updated. If S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE are passed the text will only be updated if *event.region* overlaps the text region.

  WM_DRAWITEM—A message that causes the text to be redrawn. This message is specific to Windows and OS/2.

  Expose—A message that causes the text to be redrawn. This message is specific to Motif.

- *ccode*$_{in}$ contains the logical interpretation of *event*.

UIW_TEXT::Event

Syntax

```
#include <ui_win.hpp>

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

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a text object. It is declared virtual so that any derived text class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the text object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified text object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  L_DOWN—Causes the text cursor to move down one line in the text buffer. Where possible, the cursor position stays in the same column in the new line.

  L_LEFT—Causes the text cursor to move to the left in the text buffer.

  L_PGDN—Causes the text cursor to move one page down within the text object. For example, if four lines of an eight line text object are visible and the user is positioned anywhere within the first four lines, L_PGDN causes lines five through eight to replace lines one through four as the visible portion of the text

object. L_PGDN has no effect if the user is positioned at the bottom of the text buffer.

L_PGUP—Causes the text cursor to move one page up within the text object. For example, if a user is positioned anywhere within the last four lines of the text object described above, L_PGUP causes lines one through four to replace lines five through eight as the visible portion of the text object. The cursor would now be positioned in the same column in line number four. While in that position, another L_PGUP message would cause the cursor to move to line number one. L_PGUP has no effect if the user is already positioned at the top of the text buffer.

L_RIGHT—Causes the text cursor to move to the right in the text buffer.

L_UP—Causes the text cursor to move up one line. Where possible, the cursor position stays in the same column in the new line.

L_WORD_LEFT—Causes the text cursor to move one word to the left in the text buffer.

L_WORD_RIGHT—Causes the text cursor to move one word to the right in the text buffer.

S_CREATE—Causes the text object to recompute its regions preparatory for being redisplayed.

S_HSCROLL—Causes the text field to scroll according to the information in *event.scroll.delta*, which contains the number of columns to move left or right. For example, if the cursor is positioned in the first column of the text field, an *event.scroll.delta* value of -1 causes the field to scroll right one column. Likewise, a positive value causes the text field to scroll left. The cursor will remain on its original line as long as the line is visible on the screen. If the cursor is positioned in the last column and the *event.scroll.delta* information is a negative number, or the cursor is in the first column and the value is positive, no scrolling takes place.

S_VSCROLL—Causes the text field to scroll according to the information in *event.scroll.delta*, which contains the number of lines to move up or down. For example, if the cursor is positioned in the first line of the text field, an *event.scroll.delta* value of -1 causes the field to scroll down one line. Likewise, a positive value causes the text field to scroll up. The cursor will remain in its original column as long as the column is visible on the screen and there is a

character in that column. If the cursor is positioned in the last line and the *event.scroll.delta* information is a negative number, or the cursor is on the first line and the value is positive, no scrolling takes place.

Movement and marking messages are handled by **Event( )** if the text occupies more than one line; otherwise, messages are handled by **UIW_STRING::Event( )**. The descriptions of such events are conceptually the same for both UIW_TEXT and UIW_STRING, although the effects of multiple lines in text objects should be taken into consideration. (For example, L_LEFT causes the cursor to move one space to the left in both a text and a string object; however, if the original position was the beginning of a line in a text object, the new position could be at the end of the line above it.) For more information on these events, refer to **UIW_STRING::Event( )**.

UIW_TEXT::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- $returnValue_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- $request_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_TEXT, then the request is passed on to UIW_-

WINDOW. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

CLEAR_FLAGS—Clears the WNF_FLAGS, specified by *data*, associated with the object if *objectID* is ID_WINDOW. Clears the WOF_FLAGS associated with an object if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

COPY_TEXT—Returns the *text* associated with the text field. If this message is used, *data* is the address of a buffer where the text will be copied. This buffer must be large enough to contain all of the characters associated with the text field and the terminating NULL character.

GET_FLAGS—Returns the WNF_FLAGS associated with the object if *objectID* is ID_WINDOW. The WOF_FLAGS associated with the object are returned if *objectID* is ID_WINDOW_OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

GET_TEXT—Returns the *text* associated with the text field. If this message is used, *data* must be the address of a character pointer that will point to the text field's text information.

GET_TEXT_LENGTH—Returns the length of the text field's text buffer.

SET_FLAGS—Sets the WNF_FLAGS, specified by *data*, associated with the object if *objectID* is ID_TEXT. The WOF_FLAGS are set if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

SET_TEXT—Sets the *text* associated with the text field. If this message is sent, *data* must be a character pointer to the text field's new text. The field will be re-displayed with the new text.

SET_TEXT_LENGTH—Sets the length of the text field's text buffer. If this message is sent, *data* must be a pointer to an integer containing the text field's new length.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

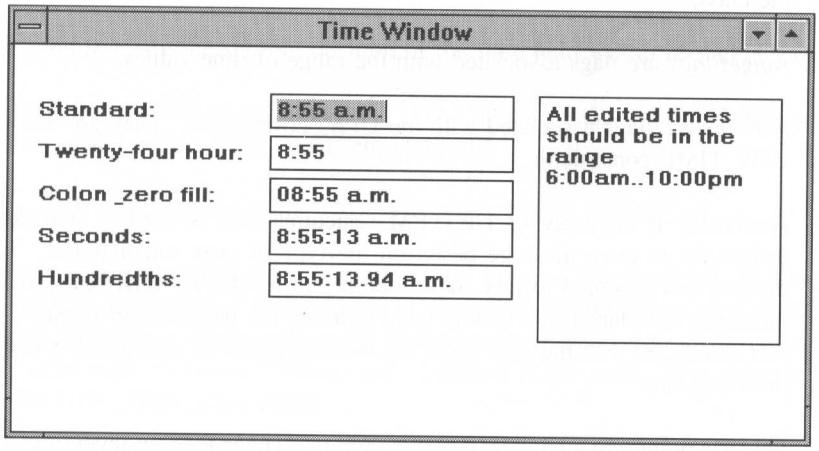
- *objectID*$_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, the request will be interpreted by UIW_WINDOW.

Example

```
#include <ui_win.hpp>

ExampleFunction()
{
    char string[30];
    text->Information(COPY_TEXT, string);

    text1->Information(SET_TEXT, "First name:");
    text2->Information(SET_TEXT, "Last name:");
    .
    .
    .

}
```

CHAPTER 67 – UIW_TIME

The UIW_TIME class is used to display time information to the screen and to collect information, in time format, from an end user. It is not the low-level time storage object. (See "Chapter 38—UI_TIME" of this manual for information about the low-level time storage object.) The figure below shows graphical implementations of UIW_TIME objects:



The UIW_TIME class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_TIME : public UIW_STRING
{
    friend class EXPORT UIF_TIME;
public:
    // Members described in UIW_TIME reference chapter.
    static TMF_FLAGS rangeFlags;
    TMF_FLAGS tmFlags;
    static UI_ITEM *errorTable;

    UIW_TIME(int left, int top, int width, UI_TIME *time,
        const char *range = NULL, TMF_FLAGS tmFlags = TMF_NO_FLAGS,
        WOF_FLAGS woFlags = WOF_BORDER | WOF_AUTO_CLEAR,
        USER_FUNCTION userFunction = NULL);
    UI_TIME *DataGet(void);
    void DataSet(UI_TIME *time);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);
    virtual int Validate(int processError = TRUE);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_TIME(const char *name, UI_STORAGE *file, UI_STORAGE_OBJECT *object);
    virtual ~UIW_TIME(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);

protected:
    // Members described in UIW_TIME reference chapter.
    UI_TIME *time;
    char *range;
};
```

- *UIF_TIME* is for use with Zinc Designer. Programmers should not use this part of the class.

- *rangeFlags* are flags associated with the range of time values.

- *tmFlags* are flags associated with the UIW_TIME class. They are described in the UIW_TIME constructor.

- *errorTable* is an array of UI_ITEM structures that is used as a lookup table to determine an error message based on the type of error encountered. The array is defined and assigned to this static member in the **G_TIME1.CPP** module. These messages are placed in a lookup table to allow the programmer to easily change the text associated with the error (e.g., the message could be changed to a language other than English).

- *time* is a pointer to a UI_TIME that is used to manage the low-level time information. If the **WOF_NO_ALLOCATE_DATA** flag is set, this member is used to initialize the UI_TIME object created by UIW_TIME.

- *range* is the copy of the string range of acceptable times passed down to the constructor.

UIW_TIME::UIW_TIME

Syntax

#include <ui_win.hpp>

UIW_TIME(int *left*, int *top*, int *width*, UI_TIME **time*, const char **range* = NULL,
    TMF_FLAGS *tmFlags* = TMF_NO_FLAGS,
    WOF_FLAGS *woFlags* = WOF_BORDER | WOF_AUTO_CLEAR,
    USER_FUNCTION *userFunction* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_TIME class object.

- $left_{in}$ and top_{in} are the starting position of the time field within its parent window.
- $width_{in}$ is the width of the time field. (The height of the time field is determined automatically by the UIW_TIME class object.)
- $time_{in}$ is a pointer to the initial UI_TIME object.
- $range_{in}$ is a string that gives the valid time ranges. For example, if a range of "12:00pm..11:59:59pm" were specified, the UIW_TIME class object would only accept those times whose values fall in the post-meridian time. If *range* is NULL, any time value (between 12:00am and 11:59pm) is accepted. This string is copied by the UIW_TIME class object.
- $tmFlags_{in}$ gives information on how to display and interpret the time information. The following flags (declared in **UI_GEN.HPP**) override the system country-dependent information:

TMF_COLON_SEPARATOR—Separates each time variable with a colon.

```
12:00
13:00:00
12:00 a.m.
```

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

```
1:05:00:00
23:15:05:99
7:45:59:00 a.m.
```

TMF_LOWER_CASE—Converts the time to lower-case.

```
12:00 p.m.
1:00 a.m.
```

TMF_NO_FLAGS—Does not associate any special flags with the UIW_TIME class object. In this case, the time will be displayed and interpreted using the default country information.

```
(United States)
1:05 p.m.
11:15 a.m.
```

This flag should not be used in conjunction with any other TMF flags.

TMF_NO_HOURS—Does not display nor interpret an hour value for the UI_TIME object.

```
12:15
(12 = minutes)
(15 = seconds)
```

TMF_NO_MINUTES—Does not display nor interpret a minute value for the UIW_TIME class object.

```
12
(12 = hours)
```

TMF_NO_SEPARATOR—Does not use any separator characters to delimit the time values.

```
1200
130000
700
```

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

```
8:09:30
14:00:00
3:24:59 p.m.
```

TMF_SYSTEM—Fills a blank time with the system time. For example, if a blank ASCII time value were entered by the end-user and the TMF_SYSTEM flag were set, the time would be set to the current system time.

```
1:10pm
(system time)
```

TMF_TWELVE_HOUR—Forces the time to be displayed and interpreted using a 12 hour clock, regardless of the default country information.

```
12:00 a.m
1:00 p.m.
5:00 p.m.
```

TMF_TWENTY_FOUR_HOUR—Forces the time to be displayed and interpreted using a 24 hour clock, regardless of the default country information.

```
12:00
13:00
17:00
```

TMF_UPPER_CASE—Converts the time to upper-case.

```
12:00 P.M.
1:00 A.M.
7:00 P.M.
```

TMF_ZERO_FILL—Forces the hour, minute and second values to be zero filled when their values are less than 10.

```
01:10 a.m
13:05:03
01:01 p.m.
```

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the time object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_TIME class object:

WOF_AUTO_CLEAR—Automatically clears the time buffer if the end-user tabs to the time field (from another window field) and then presses a key (without first having pressed any movement or editing keys).

WOF_BORDER—Draws a border around the time object, in graphics mode. In text mode, no border is drawn.

WOF_INVALID—Sets the initial status of the time field to be "invalid." An invalid time fits in the absolute range determined by the object type (i.e., "12:00am..11:59:59pm") but does not fulfill all the requirements specified by the program. For example, a time field may initially be set to "8:15am," but the final time, edited by the end-user, must be in the range "12:00pm..-11:59:59pm." The initial time in this example fits the absolute requirements of a UIW_TIME class object but does not fit into the specified range.

WOF_JUSTIFY_CENTER—Center-justifies the time information within the time field.

WOF_JUSTIFY_RIGHT—Right-justifies the time information within the time field.

WOF_NO_ALLOCATE_DATA—Prevents the time object from allocating a UI_TIME object to store the time information. If this flag is set, the programmer must allocate the UI_TIME object (passed as the *time* parameter) that is used by the time object.

WOF_NO_FLAGS—Does not associate any special window object flags with the time object. Setting this flag left-justifies the time information. This flag should not be used in conjunction with any other WOF flags.

WOF_NON_SELECTABLE—Prevents the time object from being selected. If this flag is set, the user will not be able to edit or position on the time information.

WOF_UNANSWERED—Sets the initial status of the time field to be "unanswered." An unanswered time field is displayed as blank space on the screen.

WOF_VIEW_ONLY—Prevents the time string from being modified. This flag will still allow the time field to become current.

- *userFunction*$_{in}$ is a programmer defined function that is called whenever:

1—the user moves onto the time field (i.e., S_CURRENT message),

2—the <ENTER> key is pressed (i.e., L_SELECT message) or

3—the user moves to a different field on the window or a different window on the screen (i.e., S_NON_CURRENT message).

The *userFunction* function (if one is provided) is called for the three cases listed above. The programmer is responsible for calling **Validate( )** to ensure that the time fits the absolute range for times or if the time is outside the default range (specified by the *range* argument passed in on the UIW_TIME constructor). The following arguments are passed to *userFunction*:

object$_{in}$—A pointer to the UIW_TIME class object or the class object derived from the UIW_TIME object base class. This argument must be typecast by the programmer.

event$_{in}$—The event that caused the user function to be called.

ccode$_{in}$—The logical or system code that caused the user function to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

L_SELECT—This message is sent if the <ENTER> key is pressed.

S_CURRENT—The time object is about to be edited. This code is sent before any editing operations are permitted.

S_NON_CURRENT—A different field or window has been selected. This code is sent after editing operations have been performed, if the time is valid for the absolute value of time field ranges and if the time is valid for the programmer-defined *range*.

The user function's *returnValue* should be 0 if the time is valid. Otherwise, the programmer should call the error system with an appropriate error message and return -1.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    UI_DATE date; // system date
```

```
    UI_TIME time; // system time

    // Create a window with a date and time field.
    UIW_WINDOW *window = UIW_WINDOW::Generic(0, 0, 45, 8, "Window");
    *window
        + new UIW_PROMPT(2, 1, "Date..")
        + new UIW_DATE(9, 1, 20, &date, "", DTF_ALPHA_MONTH | DTF_SYSTEM)
        + new UIW_PROMPT(2, 3, "Time..")
        + new UIW_TIME(9, 3, 20, &time, "", TMF_SECONDS);
        .
        .
        .

    // The time object will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_TIME::DataGet

Syntax

#include <ui_win.hpp>

UI_TIME *DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to return the UI_TIME object associated with the UIW_TIME.

- *returnValue*$_{out}$ is a pointer to a UI_TIME object containing the time information.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_TIME *timeObject)
{
    UI_TIME *time = timeObject->DataGet();
    .
    .
    .
}
```

UIW_TIME::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(UI_TIME **time*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to set the date information associated with the time object.

- $time_{in}$ is the new time information.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_TIME *time)
{
    .
    .
    .

    UI_TIME timeInfo(12, 0, 30);
    time->DataSet(&timeInfo);
}
```

UIW_TIME::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine is used to send run-time information to a time object. It is declared virtual so that any derived time class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the time object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified time object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_CREATE—Ensures that the string time (shown to the screen) reflects the internal time associated with the time object.

  S_CURRENT—When a time field receives this message, it calls the programmer-specified user function, if one exists. The user function will get a pointer to the UIW_TIME class, and the value for the user function *ccode* will be S_CURRENT.

  S_NON_CURRENT—Causes the programmer-specified user function to be called (described in the UIW_TIME constructor.) The value for *ccode* will be S_NON_CURRENT. The programmer must call **Validate( )**.

  All other events are passed by **Event( )** to **UIW_STRING::Event( )** for processing.

UIW_TIME::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_TIME, the request is passed on to UIW_STRING. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

  GET_FLAGS—Returns the TMF_FLAGS associated with the object if *objectID* is ID_TIME, STF_FLAGS if *objectID* is ID_STRING, or WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  SET_FLAGS—Sets the TMF_FLAGS, specified by *data*, associated with the object if *objectID* is ID_TIME, STF_FLAGS if *objectID* is ID_STRING, or WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  CLEAR_FLAGS—Clears the TMF_FLAGS, specified by *data*, associated with the object if *objectID* is ID_TIME, STF_FLAGS if *objectID* is ID_STRING, or WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

- *data*$_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

- *objectID*$_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, then the request will be interpreted by **UIW_STRING::-Information( )**.

Example

```
#include <ui_win.hpp>

ExampleFunction()
{
    time->Information(SET_TEXT, "12:00");
    .
    .
    .

    time->Information(SET_FLAGS, WOF_BORDER);
    .
    .
    .

}
```

UIW_TIME::Validate

Syntax

#include <ui_win.hpp>

virtual int Validate(int *processError* = TRUE);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to validate time objects. When a time object receives the S_CURRENT or S_NON_CURRENT messages, it will call **Validate( )** to check to see if the time entered was valid. However, if *userFunction* was specified by the programmer, then **Validate( )** will not be called by the library and must be called by the programmer if time validation is desired.

- *returnValue*$_{out}$ is 0 if the validation was positive or non-zero in the event of an error.

TMI_INVALID—The time was in an invalid format.

TMI_VALUE_MISSING—No time value was entered.

TMI_OUT_OF_RANGE—The time was not within the valid range for times or programmer specified range.

TMI_OK—The time was entered in a correct format and within the valid range.

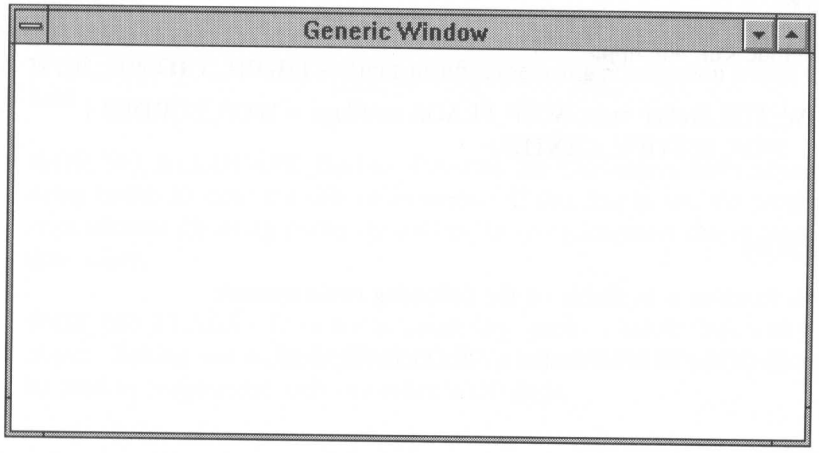
- *processError*$_{in}$ determines whether **Validate( )** should call **UI_ERROR_SYSTEM::-ReportError( )** if an error occurs.

Example

```
#include <ui_win.hpp>

EVENT_TYPE TimeUserFunction(UI_WINDOW_OBJECT *object, UI_EVENT &,
    EVENT_TYPE ccode)
{
    if (ccode != S_NON_CURRENT)
        return (ccode);

    // Do specific validation.
    UI_TIME currentTime;
    UI_TIME *time = ((UIW_TIME *)object)->DataGet();

    // Call the default Validate function to check for valid time.
    int valid = object->Validate(TRUE);

    // Call error system if the time entered is later than the system time.
    if (valid == TMI_OK && currentTime < *time)
    {
        valid = TMI_INVALID;
        char timeString[64];
        currentTime.Export(timeString, TMF_NO_FLAGS);
        object->errorSystem->ReportError(object->windowManager, WOS_NO_STATUS,
            "The time must be before %s.", timeString);
    }
    // Return error status.
    if (valid == TMI_OK)
        return (0);
    else
        return (-1);
}

void ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    UIW_WINDOW *window = UIW_WINDOW::Generic(0, 0, 45, 8, "VCR Control");
    *window
        + new UIW_PROMPT(2, 1, "Start time:")
        + new UIW_TIME(12, 1, 20, &UI_TIME(), NULL, TMF_NO_FLAGS,
            WOF_BORDER | WOF_AUTO_CLEAR, TimeUserFunction)
        + new UIW_PROMPT(2, 3, "End time:")
        + new UIW_TIME(12, 3, 20, &UI_TIME(), NULL, TMF_NO_FLAGS,
            WOF_BORDER | WOF_AUTO_CLEAR, TimeUserFunction);
    *windowManager + window;
}
```

CHAPTER 68 – UIW_TITLE

The UIW_TITLE class is used to display short textual information about the parent window and, when clicked on with a mouse, it moves the position of the parent window. When the UIW_TITLE is double clicked with the mouse, it causes the parent window to be maximized (or restored if the window is already in a maximized state.) The figure below shows graphical implementations of a window with a UIW_TITLE class object (shown with the "Generic Window" string):



The UIW_TITLE class is declared in **UI_WIN.HPP**. Its public and protected members are shown below:

```
class EXPORT UIW_TITLE : public UIW_BUTTON
{
    friend class EXPORT UIF_TITLE;
public:
    // Members described in UIW_TITLE reference chapter.
    UIW_TITLE(char *text, WOF_FLAGS woFlags = WOF_BORDER |
        WOF_JUSTIFY_CENTER);
    char *DataGet(void);
    void DataSet(char *text);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_TITLE(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_TITLE(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
};
```

- *UIF_TITLE* is for use with Zinc Designer. Programmers should not use this part of the class.

UIW_TITLE::UIW_TITLE

Syntax

#include <ui_win.hpp>

UIW_TITLE(char **text*, WOF_FLAGS *woFlags* = WOF_BORDER | WOF_JUSTIFY_CENTER);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_TITLE class object. To ensure that the title is drawn correctly, it must be added to the window after the border, maximize, minimize and system buttons. The following example shows the correct and incorrect order of title creation:

```
1)  // CORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1")
        .
        .
        .

2)  // INCORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1")
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
```

.
.
.

- $text_{in}$ is the window's title.

- $woFlags_{in}$ are flags (common to all window objects) that determine the general presentation of the title object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of a UIW_TITLE class object:

  WOF_JUSTIFY_CENTER—Center-justifies the string information within the title field.

  WOF_JUSTIFY_RIGHT—Right-justifies the string information within the title field.

  WOF_NO_ALLOCATE_DATA—Prevents the title object from allocating a string buffer to store the title information. If this flag is set, the programmer must allocate the string buffer (passed as the *text* parameter) that is used by the title object.

  WOF_NO_FLAGS—Does not associate any special window flags with the title object. Setting this flag left-justifies the title information. This flag should not be used in conjunction with any other WOF flags.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a basic window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1");
    *windowManager + window;
    .
    .
    .

    // The title object will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_TITLE::DataGet

Syntax

#include <ui_win.hpp>

char *DataGet(void);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to return the text information associated with the title object.

- *returnValue*$_{out}$ is a pointer to the text information associated with the title.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_TITLE *title)
{
    char *title = title->DataGet();
    .
    .
    .

}
```

UIW_TITLE::DataSet

Syntax

#include <ui_win.hpp>

void DataSet(char *_text_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used to set the text information associated with the title.

- $text_{in}$ is a pointer to the new text information to be displayed on the UIW_TITLE.

Example

```
#include <ui_win.hpp>

ExampleFunction(UIW_TITLE *title)
{
    .
    .
    .
    title->DataSet("Error Window");
}
```

UIW_TITLE::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine is used to send run-time information to a title object. It is declared virtual so that any derived title class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the title object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified title object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_CREATE—This message will cause the title to compute its region within its parent window and redisplay its text.

  S_DISPLAY_ACTIVE and **S_DISPLAY_INACTIVE**—In text mode, these messages ensure that the title is displayed and updated.

  L_VIEW—If the WOAF_NO_MOVE flag is not set, this message causes the mouse pointer to change to a hand icon when the mouse enters the area of the title.

All other events are passed by **Event( )** to **UIW_BUTTON::Event( )** for processing.

UIW_TITLE::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_TITLE, then the request is passed on to **UIW_BUTTON::Information( )**. See "Chapter 47—UIW_BUTTON" for information regarding the information requests.

  NOTE: To change a title's text, the **UIW_TITLE::Information( )** function can be called with the SET_TEXT request (this request is passed to **UIW_BUTTON::-Information( )**). Alternately, the **Information( )** function of the window the title is attached to can be called with the SET_TEXT request. This request will be sent to the title.

- *data*$_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

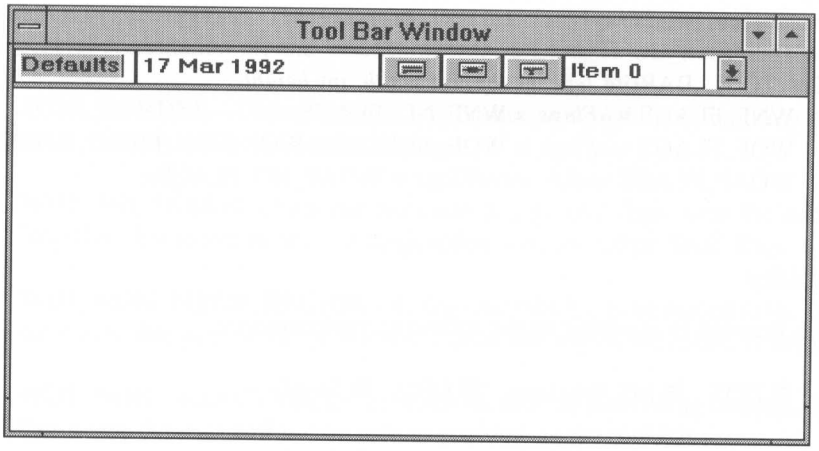
- *objectID*$_{in}$ is the identification code of the object to receive the request. If *objectID* is left blank or unrecognized, then the request will be interpreted by UIW_BUTTON.

Example

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

ExampleFunction()
{
    title->Information(SET_TEXT, "Example 2");
    .
    .
    .

}
```

CHAPTER 69 – UIW_TOOL_BAR

The UIW_TOOL_BAR class object is used as a controlling structure for a set of window objects. A UIW_TOOL_BAR differs from a pull-down menu in that it may contain a mixture of different window objects. A tool bar is positioned along the top of the window. If the window also contains a pull-down menu, the tool bar will be placed directly below the pull-down menu. Multiple tool bars may be added to a window. The figure below shows a graphical implementation of a UIW_TOOL_BAR class object with various window objects:



The public members of the UIW_TOOL_BAR class (declared in **UI_WIN.HPP**) are:

```
class EXPORT UIW_TOOL_BAR : public UIW_WINDOW
{
    friend class EXPORT UIF_TOOL_BAR;
public:
    // Members described in UIW_TOOL_BAR reference chapter.
    UIW_TOOL_BAR(int left, int top, int width, int height,
        WNF_FLAGS wnFlags = WNF_NO_FLAGS,
        WOF_FLAGS woFlags = WOF_BORDER | WOF_NON_FIELD_REGION,
        WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_TOOL_BAR(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual ~UIW_TOOL_BAR(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
    virtual void Store(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
};
```

- *UIF_TOOL_BAR* is for use with Zinc Designer. Programmers should not use this part of the class.

UIW_TOOL_BAR::UIW_TOOL_BAR

Syntax

#include <ui_win.hpp>

UIW_TOOL_BAR(int *left*, int *top*, int *width*, int *height*,
WNF_FLAGS *wnFlags* = WNF_NO_FLAGS,
WOF_FLAGS *woFlags* = WOF_BORDER | WOF_NON_FIELD_REGION,
WOAF_FLAGS *woAdvancedFlags* = WOAF_NO_FLAGS);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a UIW_TOOL_BAR class object.

- $left_{in}$ and top_{in} are the starting position of the tool bar within the window.
- $width_{in}$ and $height_{in}$ are the size of the tool bar.
- $wnFlags_{in}$ gives information on how to display the items in the tool bar. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of the tool bar:

  WNF_BITMAP_CHILDREN—Used to denote that some of the tool bar's sub-objects contain bitmaps.

  WNF_NO_FLAGS—Does not associate any special flags with the tool bar. This flag should not be used in conjunction with any other WNF flags.

WNF_NO_WRAP—Causes objects placed in the tool bar to be positioned according to their specified coordinates. By default, objects within a tool bar are automatically positioned according to the order in which they were added. If too many objects are added, the tool bar will wrap to an additional line to display all of the objects. If the **WNF_NO_WRAP** flag is set, then objects within the tool bar must have *height* and *width* values such that all of the objects may be visible.

WNF_SELECT_MULTIPLE—Allows more than one item in the tool bar to be selected.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the tool bar. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_TOOL_BAR class object:

  WOF_BORDER—Draws a single line border around the tool bar, in graphics mode. In text mode, no border is drawn.

  WOF_NO_FLAGS—Does not associate any special flags with the tool bar. This flag should not be used in conjunction with any other WOF flags.

  WOF_NON_FIELD_REGION—Causes the tool bar to be placed in the uppermost available portion of the window (under the pull-down menu, if any).

  WOF_NON_SELECTABLE—Prevents the tool bar from being selected. If this flag is set, the user will not be able to position on the tool bar.

- *woAdvancedFlags*$_{in}$ are flags that determine the advanced operation of the window object. The following flags are declared in **UI_WIN.HPP**:

  WOAF_NO_FLAGS—Does not associate any special advanced flags with the window object. This flag should not be used in conjunction with any other WOAF flags.

  WOAF_NON_CURRENT—The tool bar cannot be made current. If this flag is set, users will not be able to select the tool bar from the keyboard nor with a mouse.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    UI_DATE date();
    UI_TIME time();

    // Create a window with a tool bar.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample menus ")
        + &(*new UIW_TOOL_BAR(0,0,20,2)
            + new UIW_BUTTON(0,0,10,0,"Button")
            + new UIW_STRING(0,0,10,"String")
            + new UIW_DATE(0,0,10,&date)
            + new UIW_TIME(0,0,10,&time));
    *windowManager + window;
    .
    .
    .

    // The tool bar will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_TOOL_BAR::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT *&event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a tool bar menu object. It is declared virtual so that any derived tool bar class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the tool bar object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified tool bar object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  S_CREATE and **S_SIZE**—If the tool bar receives one of these messages, it will automatically determine the positions of the sub objects.

  All other events are passed by **Event( )** to **UIW_WINDOW::Event( )** for processing.

UIW_TOOL_BAR::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

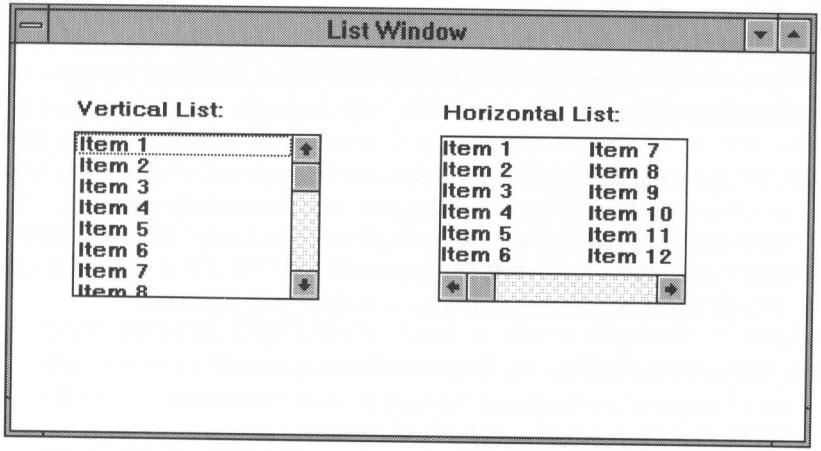
■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object. **Information( )** processes no requests, rather it passes them to **UIW_WINDOW::Information( )** for processing.

CHAPTER 70 – UIW_VT_LIST

The UIW_VT_LIST class is used to display information in a single-column fashion within a window. Any window object can be added to a vertical list (e.g., strings, dates, icons). The figure below shows the graphical implementation of a UIW_VT_LIST object with several string objects:



The UIW_VT_LIST class is declared in **UI_WIN.HPP**. Its public and protected members are:

```
class EXPORT UIW_VT_LIST : public UIW_WINDOW
{
    friend class EXPORT UIW_COMBO_BOX;
    friend class EXPORT UIF_VT_LIST;
public:
    // Members described in UIW_VT_LIST reference chapter.
    UIW_VT_LIST(int left, int top, int width, int height,
        int (*compareFunction)(void *element1, void *element2) = NULL,
        WNF_FLAGS wnFlags = WNF_NO_WRAP, WOF_FLAGS woFlags = WOF_BORDER,
        WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS);
    UIW_VT_LIST(int left, int top, int width, int height,
        int (*compareFunction)(void *element1, void *element2),
        WOF_FLAGS flagSetting, UI_ITEM *item);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);

    // Members described in UI_LIST reference chapter.
    UIW_VT_LIST(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL);
    virtual ~UIW_VT_LIST(void);
    virtual void Destroy(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
```

```
        virtual void Store(const char *name, UI_STORAGE *file = NULL,
            UI_STORAGE_OBJECT *object = NULL);
};
```

- *UIF_VT_LIST* is for use with Zinc Designer. Programmers should not use this part of the class.

UIW_VT_LIST::UIW_VT_LIST

Syntax

#include <ui_win.hpp>

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

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These overloaded constructors return a pointer to a new UIW_VT_LIST object.

The first constructor takes the following arguments:

- $left_{in}$ and top_{in} are the starting position of the list box within its parent window.
- $width_{in}$ and $height_{in}$ are the width and height of the list box field.

- *compareFunction*$_{in}$ is a programmer defined function used to determine the order of list items. New items are placed at the end of the list if this value is NULL. The following arguments are passed to *compareFunction*:

  element1$_{in}$—A pointer to the first argument to compare. This argument must be typecast by the programmer.

  element2$_{in}$—A pointer to the second argument to compare. This argument must be typecast by the programmer.

  The compare function's *returnValue* should be 0 if the two elements exactly match. If a negative value is returned, then *element1* is less than *element2*. Otherwise, a positive value indicates that *element1* is greater than *element2*. For more information about the *compare* argument see "Chapter 18—UI_LIST."

- *wnFlags*$_{in}$ are flags that determine the display pattern of the list items. The following flags (declared in **UI_WIN.HPP**) control the general presentation of the list items:

  WNF_BITMAP_CHILDREN—Used to denote that some of the list's sub objects contain bitmaps. This flag must be set if the list will contain non-string objects.

  WNF_NO_FLAGS—Does not associate any special WNF_FLAGS with the list box. This flag should not be used with any other WNF flag.

  WNF_NO_WRAP—Causes the list not to wrap when scrolling. By default, if the highlight is positioned on the last item in the list and the down key is pressed, the list will wrap and position itself on the first item in the list. The **WNF_NO_WRAP** flag disables this feature. This feature may be desirable in pull-down menus.

  WNF_AUTO_SORT—Assigns a compare function to alphabetize the strings within the list. If this flag is used, *compareFunction* should be NULL.

  WNF_SELECT_MULTIPLE—Allows multiple items within the list box to be selected. Otherwise, only one item in the list may be selected at a time.

- *woFlags*$_{in}$ are flags (general to all window objects) that determine the general operation of the list box object. The following flags (declared in **UI_WIN.HPP**) control the presentation of, or interaction with, a UIW_VT_LIST class object:

WOF_BORDER—In graphics mode, this flag draws a single line border around the list box. In text mode, no border is drawn. This is the default argument if no other argument is provided.

WOF_NO_FLAGS—Does not associate any special flags with the list box. This flag should not be used in conjunction with any other WOF flags.

WOF_NON_FIELD_REGION—The list box is not a form field. If this flag is set and the list box is attached to a higher-level window, then the *left*, *top* and *height* arguments are ignored and the list box will occupy any remaining space within the parent window.

WOF_NON_SELECTABLE—Prevents the list from being selected. If this flag is set, the user will not be able to position on or scroll through the list.

WOF_VIEW_ONLY—The list box cannot be edited. If this flag is set, the user will not be able to select list items, but will be able to scroll through the list.

- *woAdvancedFlags*$_{in}$ are flags (general to all window objects) that determine the advanced operation of the list box object.

  WOAF_NO_FLAGS—Does not associate any special advanced flags with the list box. This flag should not be used in conjunction with any other WOAF flags.

  WOAF_NON_CURRENT—Prevents the end-user from positioning on the list box at runtime.

The second constructor creates UIW_STRING objects and then adds them into the list box. This allows the programmer to add strings to a list box without having to use the **+ operator** for each one.

- *left*$_{in}$ and *top*$_{in}$ are the starting position of the list box within its parent window.

- *width*$_{in}$ and *height*$_{in}$ are the width and height of the list box field.

- *compareFunction*$_{in}$ is a programmer defined function used to determine the order of list items. New items are placed at the end of the list if this value is NULL. The following arguments are passed to *compareFunction*:

  element1$_{in}$—A pointer to the first argument to compare. This argument must be typecast by the programmer.

element2$_{in}$—A pointer to the second argument to compare. This argument must be typecast by the programmer.

The compare function's *returnValue* should be 0 if the two elements exactly match. If a negative value is returned, then *element1* is less than *element2*. Otherwise, a positive value indicates that *element1* is greater than *element2*. For more information about the *compare* argument see "Chapter 18—UI_LIST."

- *flagSetting*$_{in}$ is the variable against which the item's flag setting is compared. If the item's *value* and this *flagSetting* match, the item will be marked as having been selected.

- *item*$_{in}$ an array of UI_ITEM structures used to construct a series of UIW_STRING objects within the list box. For more information regarding the use of the UI_ITEM structure, see "Chapter 16—UI_ITEM" of this manual.

Example

```
#include <ui_win.hpp>

ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    // Create the list box field.
    UIW_VT_LIST *listBox = new UIW_VT_LIST(10, 1, 25, 6);
    *listBox
        + new UIW_STRING(0, 0, 19, "Item 1")
        + new UIW_STRING(0, 0, 19, "Item 2")
        + new UIW_STRING(0, 0, 19, "Item 3")
        + new UIW_STRING(0, 0, 19, "Item 4");

    // Attach the list box to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + UIW_BORDER
        + new UIW_TITLE(" Sample list box ")
        + new UIW_PROMPT(2, 1, "List box: ")
        + listBox;
    *windowManager + window;
    .
    .
    .

    // The list box will automatically be destroyed when the window
    // is destroyed.
}

ExampleFunction2(UI_WINDOW_MANAGER *windowManager)
{
    UI_ITEM listBoxItems[] =
    {
        { 11,   NULL,   "Item 1.1", STF_NO_FLAGS },
        { 12,   NULL,   "Item 1.2", STF_NO_FLAGS },
        { 21,   NULL,   "Item 2.1", STF_NO_FLAGS },
        { 22,   NULL,   "Item 2.2", STF_NO_FLAGS },
        { 0,    NULL,   NULL, 0 }
    };
```

```
    // Create the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10);
    *window
        + UIW_BORDER
        + new UIW_TITLE(" Sample list box ")
        + new UIW_PROMPT(2, 1, "List box:")
        + new UIW_VT_LIST(10, 1, 20, 6, NULL, WOF_NO_FLAGS, listBoxItems);
    *windowManager + window;
    .
    .
    .

    // The list box will automatically be destroyed when the window
    // is destroyed.
}
```

UIW_VT_LIST::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a list box object. It is declared virtual so that any derived list box class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the list box object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified list box object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  L_PGDN and **L_PGUP**—These similar messages move the current highlighted field one page up or down in the list box. For example, if four items of an eight item list box are visible and the user is positioned on any of the first four items,

L_PGDN causes items five through eight to replace items one through four as the visible portion of the list box.

L_UP and **L_DOWN**—These messages move the highlight either one item up or one item down. For example, if the first four items of a list box are showing and the user is positioned on item number four, L_DOWN causes the visible portion of the list box to scroll down so that items two through five are visible and the fifth item is highlighted.

L_UP works in a similar manner. If items two through five are visible and the second item is highlighted, L_UP causes the list box to scroll so that items one through four are visible and the first item is highlighted.

S_CREATE—This message causes the size of the items within the list box to be automatically calculated.

S_VSCROLL—This message causes the list box field to scroll according to the information in *event.scroll.delta*, which contains the number of items to move up or down. For example, if item number one of a list box is highlighted, an *event.scroll.delta* value of -1 causes the field to scroll down one item so that items two through five are visible. Likewise, a positive value causes the list box to scroll up. The highlight will remain in its original position as long as the item is visible on the screen. If the last item is highlighted and the *event.scroll.-delta* information is a negative number, or the first item is highlighted and the value is positive, no scrolling takes place.

All other events are passed by **Event( )** to **UIW_WINDOW::Event( )** for processing.

UIW_VT_LIST::Information

Syntax

#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST *request*, void **data*, OBJECTID *objectID*);

Portability

This function is available on the following environments:

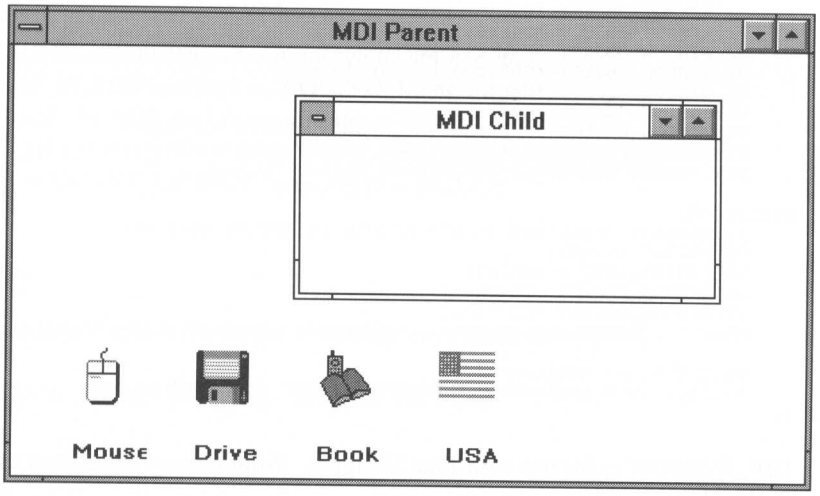
■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object. **Information( )** does not process requests, rather it passes them to **UIW_WINDOW::Information( )** for processing.

CHAPTER 71 – UIW_WINDOW

The UIW_WINDOW class is used as the controlling object for window fields that are to be displayed on the screen. Any object derived from UI_WINDOW_OBJECT may be added to a window (e.g., buttons, menus, windows). The figure below shows a graphical implementation of a UIW_WINDOW class object with an MDI child window and several minimized objects:



The UIW_WINDOW class is declared in **UI_WIN.HPP**. Its public and protected members are shown below:

```
class EXPORT UIW_WINDOW : public UI_WINDOW_OBJECT, public UI_LIST
{
    friend class EXPORT UI_WINDOW_MANAGER;
    friend class EXPORT UIF_WINDOW;
public:
    // Members described in UIW_WINDOW reference chapter.
    WNF_FLAGS wnFlags;
    UI_LIST support;

    UIW_WINDOW(int left, int top, int width, int height,
        WOF_FLAGS woFlags = WOF_NO_FLAGS,
        WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS,
        UI_HELP_CONTEXT helpContext = NO_HELP_CONTEXT,
        UI_WINDOW_OBJECT *minObject = NULL);
    virtual EVENT_TYPE Event(const UI_EVENT &event);
    static UIW_WINDOW *Generic(int left, int top, int width, int height,
        char *title, UI_WINDOW_OBJECT *minObject = NULL,
        WOF_FLAGS woFlags = WOF_NO_FLAGS,
        WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS,
        UI_HELP_CONTEXT helpContext = NO_HELP_CONTEXT);
    virtual void *Information(INFO_REQUEST request, void *data,
        OBJECTID objectID = 0);
```

```
    static int StringCompare(void *object1, void *object2);

    // Members described in UI_WINDOW_OBJECT reference chapter.
    UIW_WINDOW(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL);
    virtual ~UIW_WINDOW(void);
    static UI_WINDOW_OBJECT *New(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL);
    virtual void Load(const char *name, UI_STORAGE *file,
        UI_STORAGE_OBJECT *object);
    virtual void Store(const char *name, UI_STORAGE *file = NULL,
        UI_STORAGE_OBJECT *object = NULL);

    // Members described in UI_LIST reference chapter.
    UI_WINDOW_OBJECT *Add(UI_WINDOW_OBJECT *object);
    UI_WINDOW_OBJECT *Current(void);
    UI_WINDOW_OBJECT *First(void);
    UI_WINDOW_OBJECT *Get(const char *name);
    UI_WINDOW_OBJECT *Get(NUMBERID numberID);
    virtual void Destroy(void);
    UI_WINDOW_OBJECT *Subtract(UI_WINDOW_OBJECT *object);
    UIW_WINDOW &operator+(UI_WINDOW_OBJECT *object);
    UIW_WINDOW &operator-(UI_WINDOW_OBJECT *object);

protected:
    // Members described in UIW_WINDOW reference chapter.
    UI_REGION scroll;
    UI_REGION_LIST clipList;
    UI_WINDOW_OBJECT *vScroll;
    UI_WINDOW_OBJECT *hScroll;
    char *compareFunctionName;  // Used for stroage purposes only.

    void CheckSelection(void);
    virtual void RegionMax(UI_WINDOW_OBJECT *object);
};
```

- *UIF_WINDOW* is for use with Zinc Designer. Programmers should not use this part of the class.

- *wnFlags* contains the WNF_FLAGS associated with the window. These flags are used by windows and objects derived from UIW_WINDOW.

- *support* is a list of the items that are part of the basic window. These items include: border, title, maximize button, minimize button and system button. These items are kept in a separate list so that when the programmer adds an additional object, such as a UIW_PROMPT, it will be the first in the list when **First( )** is called.

- *scroll* is the region of the window to be scrolled.

- *clipList* is a support object used while computing clip regions.

- *vScroll* is a pointer to the vertical scroll bar attached to the window.

- *hScroll* is a pointer to the horizontal scroll bar attached to the window.

- *compareFunctionName* is the string representation of the compare function's name. It is used for storage purposes only.

UIW_WINDOW::UIW_WINDOW

Syntax

#include <ui_win.hpp>

UIW_WINDOW(int *left*, int *top*, int *width*, int *height*,
WOF_FLAGS *woFlags* = WOF_NO_FLAGS,
WOAF_FLAGS *woAdvancedFlags* = WOAF_NO_FLAGS,
UI_HELP_CONTEXT *helpContext* = NO_HELP_CONTEXT,
UI_WINDOW_OBJECT **minObject* = NULL);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This constructor returns a pointer to a new UIW_WINDOW class object.

- $left_{in}$ and top_{in} are the starting position of the window on the screen display or within the parent window.

- $width_{in}$ and $height_{in}$ are the size of the window.

- $woFlags_{in}$ are flags (common to all window objects) that determine the general operation of the window. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_WINDOW class object:

  WOF_BORDER—In graphics mode, this flag draws a single line border around the window. In text mode, no border is drawn.

  WOF_NO_FLAGS—Does not associate any special flags with the window. This flag should not be used in conjunction with any other WOF flags.

WOF_NON_FIELD_REGION—If the window is added to a parent window, this flag causes the child window to expand to fill the remaining space in the parent's window. If other objects are displayed on the parent window, but do not have this flag set, they will be hidden by the object with this flag set.

WOF_NON_SELECTABLE—Prevents the window from being selected. If this flag is set, the user will not be able to edit or move within the window.

- *woAdvancedFlags*$_{in}$ are flags that determine the advanced operation of the window object. The following flags (declared in **UI_WIN.HPP**) can only be set by advanced window objects (e.g., UIW_WINDOW, UIW_HZ_LIST, UIW_VT_LIST, UIW_-PULL_DOWN_MENU):

  WOAF_LOCKED—Prevents the Window Manager from removing the window from the screen display.

  WOAF_MDI_OBJECT—Causes the window to be an MDI window. If this flag is set on a window that is added to the Window Manager, it becomes an MDI parent (i.e., it can contain MDI child objects). An MDI parent must have a pull-down menu. In general, other than the standard support objects (i.e., system button, border, title, etc.) and the pull-down menu, MDI parent windows should only contain MDI children.

  If this flag is set on a window that is added to another MDI window, it becomes an MDI child window. MDI child windows can be moved or sized but will remain entirely within the MDI parent window.

  NOTE: MDI is not standard across environments. For example, in Windows, child windows will be clipped by their parent window, but in Motif, the child windows will not be clipped by their parent. In Motif, the child windows are still owned by the parent window, however, and so closing the parent window will cause all child windows added to the parent to close also.

  WOAF_DIALOG_OBJECT—Creates the window as a dialog box. A dialog box is a temporary window used to display or receive information from the user. Using this flag will cause a dialog style border to be displayed.

  NOTE: Some operating environments (e.g, Windows) will create a border, system button and title for a dialog window. Other environments (e.g., DOS) may not, and so a border, system button and title must be added to the dialog window by the programmer.

WOAF_MODAL—Prevents any other window from receiving event information from the Window Manager. A modal window receives all event information until it is removed from the screen display.

WOAF_NO_DESTROY—Prevents the Window Manager from calling the window's destructor. If this flag is set, the window can be removed from the screen display, but the programmer must call the destructor associated with the window object.

WOAF_NO_FLAGS—Does not associate any special advanced flags with the window object. Setting this flag allows the user to move, size and interact with the window in a normal fashion. This flag should not be used in conjunction with any other WOAF flags.

WOAF_NO_MOVE—Prevents the end-user from changing the screen location of the window at runtime.

WOAF_NO_SIZE—Prevents the end-user from changing the size of the window at runtime.

WOAF_NON_CURRENT—Prevents the window from becoming current. If this flag is set, users will not be able to select the window from the keyboard nor with a mouse.

WOAF_TEMPORARY—Causes the window to only occupy the screen temporarily. Once another window is selected from the screen, the temporary window is closed. The temporary window is also destroyed if the **WOAF_NO_DESTROY** flag is not set.

- *helpContext*$_{in}$ is the help information associated with the window. This specifies the help that will be presented by the help system when the help key is pressed. (For more information about the help system and help context information see "Chapter 14—UI_HELP_SYSTEM" of this manual.)

- *minObject*$_{in}$ is the UIW_ICON to which the window will be minimized when the S_MINIMIZE message is received.

Example 1

```
#include <ui_win.hpp>

ExampleFunction1(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1");
    *windowManager + window;
    .
    .
    .

    // The window will automatically be destroyed when the window
    // manager is destroyed.
}
```

Example 2

```
#include <ui_win.hpp>

ExampleFunction2(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window with basic window objects.
    UIW_ICON *icon = new UIW_ICON(0, 0, "iconLogo", "Zinc Logo",
        ICF_MINIMIZE_OBJECT);
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11, WOF_NO_FLAGS,
        WOAF_NO_FLAGS, NO_HELP_CONTEXT, icon);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1");
    *windowManager + window;
    .
    .
    .

    // The window will automatically be destroyed when the window
    // manager is destroyed.
}
```

UIW_WINDOW::CheckSelection

Syntax

```
#include <ui_win.hpp>

void CheckSelection(void);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This routine is used to set the selected status (and turn on the check mark, if applicable) of the current object on the window.

UIW_WINDOW::Event

Syntax

#include <ui_win.hpp>

virtual EVENT_TYPE Event(const UI_EVENT &*event*);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This advanced routine is used to send run-time information to a window object. It is declared virtual so that any derived window class can override its default operation.

- *returnValue*$_{out}$ is a response based on the type of *event*. If successful, the window object returns the logical type of event that was interpreted from the *event* reference variable. If the event could not be processed, S_UNKNOWN is returned.

- *event*$_{in}$ contains a run-time message for the specified window object. The type of operation depends on the interpreted value for the event. The following logical events are processed by **Event( )**:

  L_NEXT and **L_PREVIOUS**—These messages cause the next (L_NEXT) or the previous (L_PREVIOUS) field in the list of window objects to become current.

If the last field is current, L_NEXT causes the first field to become current. Likewise, if the first field is current, L_PREVIOUS causes the last field to become current. Any objects with the WOF_NON_SELECTABLE flag set will be ignored.

S_CREATE and **S_SIZE**—Cause the window to compute its coordinates preparatory to being displayed on the screen.

S_MAXIMIZE—Unless the WOAF_NO_SIZE and the WOAF_NO_MOVE flags are set, this message causes the window to be maximized so that it occupies the entire screen. If the window is already in a maximized state, L_MAXIMIZE causes it to return to its original size.

S_MINIMIZE—Unless the WOAF_NO_SIZE and the WOAF_NO_MOVE flags are set, this message causes the window to be minimized to an icon (if the icon's WOF_MINIMIZE_OBJECT flag was set.) If the window is already in a minimized state, L_MINIMIZE causes the window to return to its original size.

S_RESTORE—This message has effect only if the window is in either a maximized or a minimized state, in which case it causes the window to return to its original size.

Most other events are passed by **Event( )** to the current object for processing. The events L_BEGIN_SELECT, L_CONTINUE_SELECT and L_END_SELECT, however, involve a position indicator (contained in *event.position*) and are therefore routed to the window object that overlaps the specified position.

UIW_WINDOW::Generic

Syntax

```
#include <ui_win.hpp>

static UIW_WINDOW *Generic(int left, int top, int width, int height, char *title,
    UI_WINDOW_OBJECT *minObject = NULL,
    WOF_FLAGS woFlags = WOF_NO_FLAGS,
    WOAF_FLAGS woAdvancedFlags = WOAF_NO_FLAGS,
    UI_HELP_CONTEXT helpContext = NO_HELP_CONTEXT);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function returns a new window that includes a border, a maximize button, a minimize button, a system button (which is the same as that created with the **UIW_SYSTEM_BUTTON::Generic( )** function) and a title.

- $left_{in}$, top_{in}, $width_{in}$ and $height_{in}$ determine the position and size of the window.
- $title_{in}$ is a pointer to the text that is used for the window's title.
- $minObject_{in}$ is the UIW_ICON to which the window will be minimized when the S_MINIMIZE message is received.
- $woFlags_{in}$ are flags that determine the general operation of the window. They are described in the constructor section of this chapter.
- $woAdvancedFlags_{in}$ are flags that determine the advanced operation of the window object. They are described in the constructor.
- $helpContext_{in}$ is the help information associated with the window. It is described in the constructor.

Example

```
#include <ui_win.hpp>

ExampleFunction(UI_WINDOW_MANAGER *windowManager)
{
    // Create a window with basic window objects.
    UIW_WINDOW *window = UIW_WINDOW::Generic(0, 1, 67, 11, "Window1");
    *windowManager + window;
    .
    .
    .

    // The window will automatically be destroyed when the window
    // manager is destroyed.
}
```

UIW_WINDOW::Information

Syntax

```
#include <ui_win.hpp>

virtual void *Information(INFO_REQUEST request, void *data,
    OBJECTID objectID = 0);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function allows Zinc Application Framework objects and programmer functions to get or modify specified information about an object.

- *returnValue*$_{out}$ is a modified pointer to return data, based on the type of request. If the request did not require the return of pointer information, this value is the *data* pointer that was passed in.

- *request*$_{in}$ is a request to get or set information associated with the object. If a particular request is not handled by UIW_WINDOW, then the request is passed on to UI_WINDOW_OBJECT. The following requests (defined in **UI_WIN.HPP**) are recognized within Zinc Application Framework:

  CLEAR_FLAGS—Clears the WNF_FLAGS, specified by *data*, associated with the window if the value in *objectID* is ID_WINDOW or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

  COPY_TEXT—Returns the *text* associated with the window's title. If this message is used, *data* is the address of a buffer where the text will be copied. This buffer must be large enough to contain all of the characters associated with the window's title and the terminating NULL character.

GET_FLAGS—Returns the WNF_FLAGS associated with the window if the value in *objectID* is ID_WINDOW or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

GET_NUMBERID_OBJECT—Returns a pointer to the object in the window whose numberID matches the numberID passed in through the *data* argument. If this message is sent, *data* must be a pointer to a programmer defined unsigned short.

GET_STRINGID_OBJECT—Returns a pointer to the object in the window whose stringID matches the stringID passed in through the *data* argument. If this message is sent, *data* must be a pointer to a string.

GET_TEXT—Returns the *text* associated with the window's title. If this message is used, *data* must be the address of a character pointer that will point to the window's title string.

SET_FLAGS—Sets the WNF_FLAGS, specified by *data*, associated with the window if the value in *objectID* is ID_WINDOW or the WOF_FLAGS if *objectID* is ID_WINDOW_OBJECT. On success, *data* is returned. Otherwise, NULL is returned. If this message is sent, *data* must be a pointer to a programmer defined UIF_FLAGS.

SET_HSCROLL—Sets *hScroll* to the value passed by *event.data*.

SET_TEXT—Sets the text associated with the window title. If this message is sent, *data* must be a character pointer to the title's new text.

SET_VSCROLL—Sets *vScroll* to the value passed by *event.data*.

- $data_{in/out}$ is a pointer to the information needed to process the request. In general, this argument must be space allocated and initialized by the programmer.

- $objectID_{in}$ is the identification code of the object to receive the request. If *objectID* is unrecognized, the request will be interpreted by UI_WINDOW_OBJECT.

Example

```
#include <ui_win.hpp>

ExampleFunction()
{
    // Update the window's title.
    window->Information(SET_TEXT, "New Window");
    .
    .
    .

}
```

UIW_WINDOW::RegionMax

Syntax

#include <ui_win.hpp>

virtual void RegionMax(UI_WINDOW_OBJECT *_object_);

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function calculates how much space *object* can occupy within the window. Objects within the window that have the WOF_NON_FIELD_REGION flag set, such as the title and system button of a window, have their regions reserved and are therefore not included in the total available maximum region. The regions of any other objects, however, are included in the total available region, since these objects can overlap with others.

- $object_{in/out}$ is a pointer to the object that is requesting the maximum region of the window.

UIW_WINDOW::StringCompare

Syntax

```
#include <ui_win.hpp>

static int StringCompare(void *object1, void *object2);
```

Portability

This function is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This function is used as the compare function if the WNF_AUTO_SORT flag is set on the window. This function causes the objects to be sorted in ascending alphabetical order.

- *returnValue*$_{out}$ is negative if the text associated with *object1* is alphabetically less than the text associated with *object2*, 0 if the text associated with both objects is the same, or positive if the text associated with *object1* is alphabetically greater than the text associated with *object2*.

- *object1*$_{in}$ is a pointer to the first object to be compared.

- *object2*$_{in}$ is a pointer to the second object to be compared.

APPENDIX A – SUPPORT DEFINITIONS

This appendix describes various support items of Zinc Application Framework. These items, along with a short summary of their purpose, are given below. A more detailed description of each item follows this summary.

attrib—Macro that combines text foreground and background color attributes.

FlagSet—Macro that determines if any flags from one argument are contained in a second argument.

FlagsSet—Macro that determines if all flags from one argument are contained in a second argument.

Max—Macro that returns the larger of two arguments.

Min—Macro that returns the smaller of two arguments.

NULL—Variable used to indicate a null argument or function.

OBJECTID—Type declaration for object identifications.

SCREENID—Type declaration for screen identifications.

TRUE and **FALSE**—Boolean operators that are opposites.

UCHAR—Type declaration for an unsigned char.

ULONG—Type declaration for an unsigned long.

UIF_FLAGS—Base type declaration for library flags.

USHORT—Type declaration for an unsigned short.

VOIDF—Macro type typecasts a function pointer.

VOIDP—Macro that typecasts a pointer to data.

attrib

Syntax

#include <ui_dsp.hpp>

#define attrib(*foreground, background*) (((background) << 4) + (foreground))

Portability

This macro is available on the following environments:

■ DOS ☐ MS Windows ☐ OS/2 ☐ Motif

Remarks

This macro combines text foreground and background color values to make one UCHAR value. **attrib** is used by the UI_PALETTE structure to describe color and monochrome text attributes.

Example

```
#include <ui_dsp.hpp>

static UI_PALETTE backgroundPalette =
{
    '\260', attrib(BLUE, BLACK), attrib(MONO_DIM, MONO_BLACK),
    PTN_INTERLEAVE_FILL, BLUE, BLUE, BW_WHITE, BW_WHITE, GS_GRAY, GS_GRAY
};

static UI_PALETTE xorPalette =
{
    '\260', attrib(BLUE, BLACK), attrib(MONO_DIM, MONO_BLACK),
    PTN_SOLID_FILL, LIGHTGRAY, LIGHTGRAY, BW_WHITE, BW_WHITE, GS_GRAY, GS_GRAY
};
```

FlagSet

Syntax

include <ui_gen.hpp>

#define FlagSet(*flag1, flag2*) ((flag1) & (flag2))

Portability

This macro is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This macro AND's two arguments together. For example, if one argument were a 0 and the other were a 1, the result would be FALSE, since there are no bits in the arguments that overlap. On the other hand, if one argument were 0x0100 and the other were 0x0F00, the result would be 0x0100 (TRUE).

Example

```
#include <ui_win.hpp>

void UIW_BUTTON::DataSet(char *_text)
{
    // Reset the string.
    if (text && text != _text && !FlagSet(woFlags, WOF_NO_ALLOCATE_DATA))
        delete text;
    if (text == _text || FlagSet(woFlags, WOF_NO_ALLOCATE_DATA))
        text = _text;
    else
        text = _text ? strdup(_text) : NULL;

    // Check for a hotkey.
    if (text)
    {
        extern void strrepc(char *string, int c, int repc);
        strrepc(text, '~', '&');
        char *tHotKey = strchr(text, '&');
        if (tHotKey)
            hotKey = tolower(tHotKey[1]);
    }
```

FlagsSet

Syntax

include <ui_gen.hpp>

#define FlagsSet(*flag1*, *flag2*) (((flag1) & (flag2)) == (flag2))

Portability

This macro is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This macro is similar to **FlagSet**, except that it checks to see if the two arguments contain the same flags, or if the second argument's flags are all contained in the first argument. For example, if one flag were a 0 and the other were a 1, the result would be 0 (FALSE), since there are no flags that overlap. On the other hand, if one flag were 0x0100 and the other were 0x0F00, the result would be 0x0100 (FALSE), while these flags reversed (i.e., 0x0F00, 0x0100) would result in TRUE.

Both **FlagSet** and **FlagsSet** are used extensively throughout the library when comparing window flags (i.e., WOF_FLAGS and WOAF_FLAGS) or when comparing status flags (i.e., WOS_STATUS and WOAS_STATUS). They are also used in window object derived classes when flags are compared (e.g., BTF_ flags in the button class). The Event Manager also uses **FlagSet** and **FlagsSet** with the **UI_EVENT_MANAGER::Get( )** function to determine the point of the queue from which the event will be retrieved.

Example

```
#include <ui_win.hpp>

EVENT_TYPE UIW_BORDER::Event(const UI_EVENT &event)
{
    // Switch on the event type.
    EVENT_TYPE ccode = LogicalEvent(event);
    switch (ccode)
    {
    .
    .
    .

    case L_VIEW:
    case L_BEGIN_SELECT:
        {
        .
        .
        .

        if (ccode == L_BEGIN_SELECT)
        {
            if (!FlagSet(parent->woAdvancedFlags, WOAF_NO_SIZE) &&
                !FlagSet(parent->woStatus, WOS_MAXIMIZED | WOS_MINIMIZED))
                parent->Event(UI_EVENT(S_SIZE, sizeFlags, event.position));
            break;
        }

        DEVICE_IMAGE image = DM_VIEW;
        if (FlagsSet(sizeFlags, M_LEFT_CHANGE | M_TOP_CHANGE) ||
```

```
            FlagsSet(sizeFlags, M_RIGHT_CHANGE | M_BOTTOM_CHANGE))
            image = DM_DIAGONAL_ULLR;
        else if (FlagsSet(sizeFlags, M_LEFT_CHANGE | M_BOTTOM_CHANGE) ||
            FlagsSet(sizeFlags, M_RIGHT_CHANGE | M_TOP_CHANGE))
            image = DM_DIAGONAL_LLUR;
        else if (FlagSet(sizeFlags, M_LEFT_CHANGE | M_RIGHT_CHANGE))
            image = DM_HORIZONTAL;
        else if (FlagSet(sizeFlags, M_TOP_CHANGE | M_BOTTOM_CHANGE))
            image = DM_VERTICAL;
        eventManager->DeviceState(E_MOUSE, image);
        }
        break;

    default:
        ccode = UI_WINDOW_OBJECT::Event(event);
        break;
    }
    // Return the control code.
    return (ccode);
}
```

Max

Syntax

include <ui_gen.hpp>

#define Max(*arg1*, *arg2*) (((arg1) > (arg2)) ? (arg1) : (arg2))

Portability

This macro is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This macro returns the larger of two arguments.

Example

```
#include <ui_dsp.hpp>
int UI_REGION_ELEMENT::Overlap(const UI_REGION &tRegion)
{
    return (Max(tRegion.left, region.left) <= Min(tRegion.right, region.right)
        && Max(tRegion.top, region.top) <= Min(tRegion.bottom, region.bottom));
}
```

Min

Syntax

include <ui_gen.hpp>

#define Min(*arg1*, *arg2*) (((arg1) < (arg2)) ? (arg1) : (arg2))

Portability

This macro is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This macro returns the smaller of two arguments.

Example

```
#include <ui_dsp.hpp>

int UI_REGION_ELEMENT::Overlap(const UI_REGION &tRegion)
{
    return (Max(tRegion.left, region.left) <= Min(tRegion.right, region.right)
        && Max(tRegion.top, region.top) <= Min(tRegion.bottom, region.bottom));
}
```

NULL

Syntax

include <ui_gen.hpp>

```
#if !defined(NULL)
#   if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
#       define  NULL  0
#   else
#       define  NULL  0L
#   endif
#endif
```

Portability

This definition is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

This variable is used to indicate a null function, variable or argument.

Example

```
class UI_ELEMENT
{
    friend class UI_LIST;

public:
    // Public members described in UI_ELEMENT reference chapter.
    UI_ELEMENT *previous, *next;

    UI_ELEMENT(void) : previous(NULL), next(NULL) { }
    virtual ~UI_ELEMENT(void) { }
    UI_ELEMENT *Next(void) { return(next); }
    UI_ELEMENT *Previous(void) { return(previous); }
};
```

OBJECTID

Syntax

#include <ui_dsp.hpp>

typedef unsigned int OBJECTID;

Portability

This type is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The OBJECTID variable is used to associate an identification value with each object (e.g., ID_BORDER, ID_WINDOW, etc.).

SCREENID

Syntax

```
#include <ui_dsp.hpp>

#if defined(ZIL_MSDOS)
    typedef long SCREENID;
#elif defined(ZIL_MSWINDOWS)
    typedef HWND SCREENID;
#elif defined(ZIL_OS2)
    typedef HWND SCREENID;
#elif defined(ZIL_MOTIF)
    typedef Widget SCREENID;
```

Portability

This type is available on the following environments:

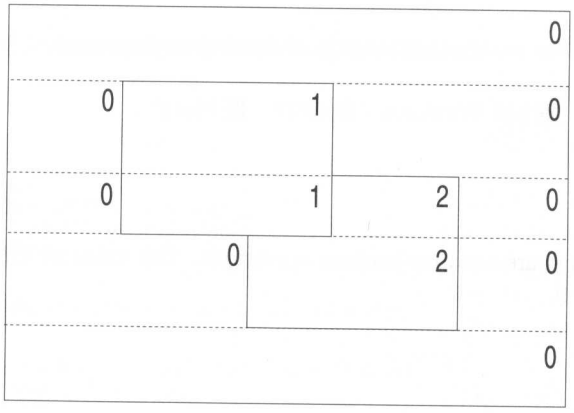
■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The SCREENID variable is used to associate an identification or window handle with the low-level display. There are several definitions for SCREENID—one for each platform. Under DOS, SCREENID values are determined by the Window Manager when a window is attached to the screen. Under Microsoft Windows, Windows NT and IBM OS/2, SCREENID is determined by the native system when the window is attached to the screen.

NOTE: Under Motif, SCREENID is used slightly differently. Under Motif, SCREENID is a pointer to the type of Xt widget that the particular object is.

The display uses screen identifications to reserve regions of the screen. For example, if the programmer defined two windows and attached them sequentially to the Window Manager (e.g., window1 first and then window2), the Window Manager would register unique screen identifications with each window and with the low-level display. The following figure shows how the display may register these screen regions (where 0 represents the screen background, and 1 and 2 represent the overlapping windows):



Once a window displayed its information to the screen, the display would match the window's identification with its list of region identifications before any information could be painted to the screen. Only those regions that matched the windows identification would be updated to the screen.

Example

```
class UI_REGION_ELEMENT : public UI_ELEMENT
{
public:
    // Members described in UI_REGION_ELEMENT reference chapter.
    SCREENID screenID;

    UI_REGION region;
    .
    .
    .

};
```

TRUE and FALSE

Syntax

include <ui_gen.hpp>

#ifndef TRUE
    const int *TRUE* = 1;
    const int *FALSE* = 0;
#endif

Portability

These definitions are available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

These constants are used for boolean operations. The value of TRUE is 1, and the value of FALSE is 0.

Example

```
class UI_DISPLAY: public UI_REGION_LIST
{
public:
    .
    .
    .
    virtual void Text(int screenID, int left, int top, const char *text,
        const UI_PALETTE *palette, int length = -1, int fill = TRUE,
        int xor = FALSE, const UI_REGION *clipRegion = NULL,
        LOGICAL_FONT font = FNT_DIALOG_FONT) = 0;
    .
    .
    .

};
```

UCHAR

Syntax

include <ui_gen.hpp>

typedef unsigned char *UCHAR*;

Portability

This type is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The *UCHAR* variable is an unsigned character that is one byte.

Example

```
// --- key.shiftState ---
const UCHAR S_RIGHT_SHIFT    = 0x0001;
const UCHAR S_LEFT_SHIFT     = 0x0002;
const UCHAR S_CTRL           = 0x0004;
const UCHAR S_ALT            = 0x0008;

// --- key,value is the low 8 bits of the scan code ---
struct UI_KEY
{
    // Fields described in UI_KEY reference chapter.
    UCHAR shiftState, value;
};
```

UIF_FLAGS

Syntax

include <ui_gen.hpp>

typedef USHORT *UIF_FLAGS*;

Portability

This type is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The *UIF_FLAGS* type is used as the base type for all library flags. Using a base type allows for portability to environments that have different byte sizes, etc. The following flags are defined by the library: **BTF_FLAGS** (button flags), **DTF_FLAGS** (date flags), **ICF_FLAGS** (icon flags), **MNIF_FLAGS** (menu item flags), **NMF_FLAGS** (number flags), **SBF_FLAGS** (scroll bar flags), **TMF_FLAGS** (time flags), **UIS_FLAGS** (status flags), **WNF_FLAGS** (window flags), **WOF_FLAGS** (window object flags), **WOAF_-FLAGS** (advanced window object flags).

ULONG

Syntax

include <ui_gen.hpp>

typedef unsigned long *ULONG*;

Portability

This type is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The *ULONG* variable is an unsigned long that is four bytes.

USHORT

Syntax

include <ui_gen.hpp>

typedef unsigned short *USHORT*;

Portability

This type is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The *USHORT* variable is an unsigned short that is two bytes.

VOIDF

Syntax

include <ui_env.hpp>

```
#if defined(__BCPLUSPLUS__) || defined(__TCPLUSPLUS__)
#    define VOIDF(function) ((void *)(function))
```

Portability

This macro is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The *VOIDF* macro is used to typecast a function pointer according to the requirements of the environment. The definition of *VOIDF* varies depending on the environment. This macro is typically used in UI_ITEM arrays.

VOIDP

Syntax

include <ui_env.hpp>

```
#if defined(__BCPLUSPLUS__) || defined(__TCPLUSPLUS__)
#    define VOIDP(pointer) ((void *)(pointer))
```

Portability

This macro is available on the following environments:

■ DOS ■ MS Windows ■ OS/2 ■ Motif

Remarks

The *VOIDP* macro is used to typecast a data pointer according to the requirements of the environment. The definition of *VOIDP* varies depending on the environment.

APPENDIX B – SYSTEM EVENTS

This appendix describes the system events that can be generated in Zinc Application Framework. System messages are passed using the UI_EVENT structure, where the system message is contained in *event.type* and any related information is contained in the union portion of the UI_EVENT structure. (For additional information about system event mapping, see the **Event( )** member functions associated with window objects.) The following messages (declared in **UI_EVT.HPP**) are recognized within Zinc Application Framework:

S_ADD_OBJECT—Is used to add an object (contained in *event.data*) to the receiving object. This message is interpreted only by those objects that contain a list (e.g., windows, horizontal and vertical lists, combo boxes, etc.).

S_ALT_KEY—Is sent if the <Alt> key is pressed and released without pressing any other keys. This message switches focus between the pull-down menu and the current object on the window and is specific to DOS.

S_CLOSE—Tells each window object to uninitialize its internal information. This message is sent by the Window Manager whenever a window is removed from the screen display (such as when the close option is selected by the end-user). This message can also be sent directly to the Window Manager, telling it to remove the current window from its list of windows and to destroy it. As a result, the window is removed from the screen. If a temporary window (a window with the WOAF-_TEMPORARY flag set, such as a pull-down menu) is present on the screen, it will remove it and the next window in the Window Manager's list of windows. If the WOAF_NO_DESTROY flag is set on the current window, the S_CLOSE message will cause that window to be subtracted from the Window Manager, which will cause it to disappear from the screen as well, but it will not be destroyed.

S_CLOSE_TEMPORARY—Is the same as S_CLOSE, except that it always removes only one window from the screen. Thus, if the top window is a temporary window (as determined by the WOAF_TEMPORARY flag), it is the only window removed from the screen. For example, if an end user presses <Esc> (the default key associated with S_CLOSE_TEMPORARY) while a pull-down menu is the current window, only the pull-down menu will be removed. Any other windows will remain on the screen. In all other instances, the current window is removed from the screen.

S_CONTINUE—This message is sent by the programmer to the Event Manager, allowing it time to poll its devices (e.g., keyboard and cursor). The message is sent to the current object but does not have any effect on program execution other than

to allow the system time to poll the devices. For example, this event is necessary if a continuous clock device has been implemented, because it ensures that the clock will change every second even if another time-consuming event is in progress.

S_CREATE—Tells each window object to initialize its internal information, such as its size and position within the parent window. The S_CREATE message is always succeeded by an S_CURRENT, S_DISPLAY_ACTIVE or S_DISPLAY_INACTIVE message. This message is sent to all of the window objects associated with a window whenever the window is attached to the Window Manager.

S_CURRENT—Is a message, sent by the Window Manager, which tells the receiving window object that it is the current (or highlighted) window object on the screen. This message is sent instead of the S_DISPLAY_ACTIVE message, which applies to all other objects within the window. The S_CURRENT message uses the UI_REGION portion of the UI_EVENT structure to indicate the coordinates of any region that overlaps a part of the previously current window object. This region will need to be refreshed, or repainted, to the foreground of the display. If no region is overlapping, the object will simply be shown in a current mode, without any repainting. Once the S_CURRENT message is sent, the window object will receive all relevant event information passed through the Window Manager, since event messages are most often intended for the current object.

S_DEINITIALIZE—This message is sent by the Window Manager to a window when it is being subtracted from the Window Manager. This message is used by the window to deinitialize any information.

S_DISPLAY_ACTIVE—Is sent through the parent window to a window object, telling the object to re-display itself according to an active state. In other words, one of the other objects within the parent window has become current, so the rest of the objects must be re-displayed according to an active state. This message is passed with the affected region (contained in the UI_REGION portion of the UI_EVENT structure). The object only needs to re-display its screen information when the region passed by the event overlaps the region of the object.

S_DISPLAY_INACTIVE—Is sent through a parent window to a window object, telling the object to re-display itself according to an inactive state. In other words, the parent window is no longer active (meaning that none of its objects are current), and all objects within it must be re-displayed as inactive. This message is passed with the affected region (contained in the UI_REGION portion of the UI_EVENT structure). The object only needs to re-display its screen information when the region passed by the event overlaps the region of the object.

S_HSCROLL—Tells the receiving window object to scroll its information the total number of columns specified by *event.scroll.delta*. For example, a value of -5 tells the receiving object to scroll the information **left** five lines.

S_HSCROLL_CHECK—Tells the receiving window object to set its scroll information. This message sends no scroll information, rather, it serves as a method informing a window object that may need to scroll. For example, when an item is deleted from a horizontal list, this message will redisplay list items and move over the items that were to the right of the deleted item.

S_HSCROLL_SET—Tells the receiving window object to set its scroll information. If this message is sent *event.scroll.maximum* should contain the total number of columns the object occupies (i.e., the total number of columns required to show all the object's information); *event.scroll.current* should contain the position of the cursor within the scroll field; *event.scroll.showing* should contain the total number of columns the object currently occupies on the screen.

S_INITIALIZE—This message is sent by the Window Manager to a window when it is being added to the Window Manager. This message is used to initialize the window and objects attached to it.

S_MAXIMIZE—Is sent through the Window Manager to the parent window, telling it to either enlarge the current window on the screen to its maximum size (normally the size of the screen), or to restore the window to its normal state if it is already maximized.

S_MDICHILD_EVENT—This message is added to another system event to cause that system event to act on an MDI child, as opposed to a window attached directly to the Window Manager. For example, to close a child window, an event of type S_CLOSE + S_MDICHILD_EVENT would be placed on the event queue. Only select system events are supported by MDI children.

S_MINIMIZE—Is sent through the Window Manager to the parent window, telling it to either reduce the current window on the screen to its minimum size.

S_MOVE—Is sent from the Window Manager to the window object, telling it that the parent window has been moved. The delta position of the object is contained in the *position* field of UI_EVENT, which is passed within this message. For example, an *event.position.line* of -10 and an *event.position.column* of 15 says that the window object coordinates need to be moved 10 lines up and 15 columns to the right.

S_NO_OBJECT—Is sent back to the programmer from the Window Manager, indicating that no window object is attached to it. An event was passed that was intended for a particular window object, but since no object is attached to the Window Manager, no processing can occur.

S_NON_CURRENT—Tells the receiving window object that it is no longer the current window object. The only exception to this is when the receiving object sends back an S_ERROR message (e.g., a date field with an out-of-range date). In this case, the current window object does not change its status until the error is resolved.

S_POSITION—This message is used to reset the current position of the mouse pointer or the text cursor. *event.position* must contain the new position of the cursor.

S_REDISPLAY—Is handled by the Window Manager. It tells the system to re-display the background and all windows attached to the screen. This message is converted to S_CURRENT, S_DISPLAY_ACTIVE and S_DISPLAY_INACTIVE messages by the Window Manager and is sent as such to each window that is attached to the Window Manager.

S_REGION_DEFINE—Is sent to a window object when it has the WOAF_-MULTIPLE_REGIONS flag set. This allows the window object to define its own sub-regions on the screen (using the **UI_DISPLAY::RegionDefine( )** member function).

S_RESET_DISPLAY—If *event.type* is S_RESET_DISPLAY and *event.data* is NULL, then the Event Manager will send the D_RESTORE message to all input devices. This tells the Event Manager that the display is about to be reset. If *event.type* is S_RESET_DISPLAY and *event.data* points to a new display, then the Event Manager will re-initialize the *display* pointer associated with the Event Manager and send the S_INITIALIZE message to all input devices. In addition, the Window Manager will re-initialize its display pointer and the pointers of all the window objects attached to the Window Manager.

S_RESTORE—Restores the window object to its original size.

S_SIZE—Is passed by window objects to the Window Manager in order to initiate a size operation. In this case, the Window Manager allows the end-user to size the window according the size operations permitted by the application. The UI_REGION portion of UI_EVENT structure is used to indicate which sides of the window can be modified. The available selections (M_RIGHT_CHANGE, M_TOP_CHANGE, M_BOTTOM_CHANGE, M_LEFT_CHANGE) are OR'ed together to give the allowed size operations. Once a window is sized, the S_SIZE message is passed on

to the affected window to indicate the new size of the window. The new window size is once again contained in the UI_REGION portion of the UI_EVENT structure.

S_SYSTEM_FIRST and **S_SYSTEM_LAST**—Contain the low and high system event values. The purpose of these messages is so an event function can look at *event.type* and check to see if it is a system event (i.e., if it is between **S_SYSTEM_-FIRST** and **S_SYSTEM_LAST**).

S_SUBTRACT_OBJECT—Is used to delete an object (pointed to by *event.data*) from the receiving object. This message is interpreted only by those objects that contain a list (e.g., windows, horizontal and vertical lists, combo boxes, etc.).

S_UNKNOWN—The event (keyboard, mouse or system) was not recognized by the receiving window object. If this message is passed back, no action was taken by the current window object.

S_VSCROLL—Tells the receiving window object to scroll its information the total number of lines specified by *event.scroll.delta*. For example, a value of -5 tells the receiving object to scroll the information **up** five lines.

S_VSCROLL_CHECK—Tells the receiving window object to set its scroll information. This message sends no scroll information, rather, it serves as a method informing a window object that may need to scroll. For example, when an item is deleted from a vertical list, this message will redisplay list items and move up the items that were below the deleted item.

S_VSCROLL_SET—Tells the receiving window object to set its scroll information. If this message is sent *event.scroll.maximum* should contain the total number of lines the object occupies (i.e., the total number of lines required to show all the object's information); *event.scroll.current* contains the position of the cursor within the scroll field; *event.scroll.showing* contains the total number of lines the object currently occupies on the screen.

APPENDIX C – LOGICAL EVENTS

This appendix describes the logical events that can be interpreted or generated in Zinc Application Framework. Logical events are passed using the UI_EVENT structure, where the logical message can either be contained directly in *event.type* or interpreted from *event.rawCode* using **MapEvent( )**. Any related information is contained in the union portion of the UI_EVENT structure. (For additional information about logical event mapping, see the **Event( )** member functions associated with window objects.)

General

The following logical messages (declared in **UI_EVT.HPP**) are general logical messages:

L_ALT_KEY—Indicates that the keyboard <Alt> key was pressed and then released without another key being pressed. This message is only sent by the keyboard device. L_ALT_KEY, which is specific to DOS, causes the window's system menu (if any) to become current.

L_BEGIN_ESCAPE—Begins the escape process of a window or a window object. (This message is normally sent by the mouse driver.) When the mouse button is released (L_END_ESCAPE), the escape process will be completed.

L_BEGIN_SELECT—Begins the selection process of a window or window object. (This message is normally sent by the mouse driver.) For example, if the end-user presses the left mouse button, the selection of the object is initiated. When the mouse button is released (L_END_SELECT), the selection will be completed.

L_CANCEL—Can be processed by the programmer to cancel operations done to the current window. Unless this event is implemented by the programmer, it is ignored by the Window Manager, since complete backup copies of a window's field data are not available within the library itself.

L_CONTINUE_ESCAPE—Is a drag operation sent by the mouse driver, indicating that L_BEGIN_ESCAPE was already sent and the mouse is presently being dragged as part of the escape process.

L_CONTINUE_SELECT—Is a drag operation sent by the mouse driver, indicating that L_BEGIN_SELECT was already sent and the mouse is presently being dragged as part of the selection process.

L_END_ESCAPE—Indicates that the escape process, initiated with the L_BEGIN_-ESCAPE message, is complete. For example, the end-user has pressed and released the right mouse button.

L_END_SELECT—Indicates that the selection process, initiated with the L_-BEGIN_SELECT message, is complete. For example, the end-user has pressed and released the left mouse button.

L_EXIT—Tells the programmer that all program flow should discontinue between the Event Manager and the Window Manager. The Window Manager treats this message as a no-op; that is, no windows are removed from the Window Manager until its destructor is called.

L_EXIT_FUNCTION—Tells the programmer that the user wishes to exit the program.

L_HELP—If the message is interpreted by the Window Manager, it requests general help associated with the application. If this message is interpreted by a particular window, it requests the context-sensitive help associated with the window.

L_MDICHILD_EVENT—This message is added to another logical event to cause that logical event to act on an MDI child, as opposed to a window attached directly to the Window Manager. Only select logical events are supported by MDI children.

L_SYSTEM_FIRST and **L_SYSTEM_LAST**—Contain the low and high system event values. The purpose of these messages is so an event function can look at *event.type* and check to see if it is a system event (i.e., if it is between **L_SYSTEM_-FIRST** and **L_SYSTEM_LAST**).

L_SELECT—Selects the current object or completes a selection process. For example, if a user is positioned on a menu item and presses <Enter>, the message is converted to an L_SELECT logical event, the item is selected, and any associated action procedure is called.

Window Manager

The following messages are interpreted by the Window Manager:

L_NEXT—Makes the next window object the current window object on the screen.

L_VIEW—Is an interpreted event which indicates that no mouse buttons are currently pressed but the mouse is being moved across the screen.

Window Objects

All of the following messages are movement for window objects which are attached to a window:

L_DOWN—If the field is a multi-line field and the cursor is not positioned on the bottom line, L_DOWN moves the cursor *down* one line on the display.

L_LEFT—Moves the cursor to the left in the current field.

L_NEXT—Moves from the current window field to the *next* selectable window field (i.e., the next window field that is attached to the window). If the last window field is currently selected, L_NEXT cycles to the *first* selectable window field.

L_PREVIOUS—Moves from the current window field to the *previous* selectable window field (i.e., the window field attached immediately before the current field). If the first window field is currently selected, L_PREVIOUS cycles to the *last* selectable window field.

L_RIGHT—Moves the cursor to the right in the current field.

L_UP—If the field is a multi-line field and the cursor is not positioned on the top line, L_UP moves the cursor *up* one line on the display.

Menu Items

All of the following are movement for menu items:

L_DOWN—Moves to the menu item immediately *below* the current menu item.

L_FIRST—Moves to the first item of the current menu, or, if the first item is already current, it moves to the first item of the previous selectable menu.

L_LAST—Moves to the last item of the current menu, or, if the last item is already current, it moves to the last item of the next selectable menu.

L_LEFT—If the current menu has more than one column of items, L_LEFT moves the cursor (highlight) to the item immediately to the left of the current item.

L_NEXT—Moves from the current menu item to the next selectable menu item.

L_PREVIOUS—Moves from the current menu item to the previous selectable menu item.

L_RIGHT—If the current menu has more than one column of items, L_RIGHT moves the cursor (highlight) to the item immediately to the right of the current item.

L_UP —Moves to the menu item immediately above the current menu item.

Edit Objects

The remaining logical events are processed by the edit objects (UIW_STRING, UIW_TEXT, UIW_NUMBER, UIW_DATE, UIW_TIME).

L_BEGIN_MARK—Determines the beginning position in marking a region.

L_CONTINUE_MARK—Indicates that the marking procedure (initiated by L_BEGIN_MARK) is in progress and that the marked area is being enlarged or reduced.

L_COPY_MARK—Copies the marked region and places it in the global paste buffer.

L_CUT—Removes the marked region and places it in the global paste buffer.

L_CUT_PASTE—If a region is marked, this message cuts the marked region and places it in the global paste buffer. If a region is not marked, this message pastes the contents of the global paste buffer at the current cursor position.

L_DELETE—Deletes the character at the cursor position. The cursor position remains the same after the operation.

L_DELETE_EOL—Deletes from the cursor position to the end of the line.

L_DELETE_WORD—Deletes the word at the current cursor position.

L_END_MARK—Completes the mark operation (initiated by L_BEGIN_MARK) of a specific region within the field.

L_INSERT_TOGGLE—Toggles between *insert* and *overstrike* edit modes in a field.

L_MARK—Marks *all* of the information contained in the field. This message is generally only interpreted by one-line edit objects.

L_BOL—Moves the cursor to the beginning of the current line within the current object.

L_BOTTOM—Moves the cursor to the bottom of the current field.

L_DOWN—If the field occupies a single line of the screen, or if the cursor is positioned on the bottom line of a multi-line field, L_DOWN moves from the current window field to the window field immediately below the current field. The left or right edge of the field above must be aligned vertically with the boundary of the current field. If the field is a multi-line field and the cursor is not positioned on the bottom line, L_DOWN moves the cursor down one line on the display.

L_EOL—Moves the cursor to the end of the current line within the current object.

L_LEFT—Moves the cursor to the previous character.

L_PGDN—If the field occupies a single line on the screen, or if the cursor is positioned on the bottom line of a multi-line field, L_PGDN moves the cursor from the current 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, L_PGDN moves the cursor down one page in the current field.

L_PGUP—If the field occupies a single line on the screen, or if the cursor is positioned on the top line of a multi-line field, L_PGUP moves the cursor from the current 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, L_PGUP moves the cursor up one page in the current field.

L_RIGHT—Moves the cursor to the next character.

L_TOP—Moves the cursor to the top of the current field.

L_UP—If the field occupies a single line of the screen, or if the cursor is positioned on the top line of a multi-line field, L_UP moves form the current window field to the window field immediately above the current field. The left or right edge of the field above must be aligned vertically with the boundary of the current field. If the field is a multi-line field and the cursor is not positioned on the top line, L_UP moves the cursor up one line on the display.

L_PASTE—Copies the contents of the global paste buffer (placed there by L_CUT or L_COPY_MARK procedures) into the current field. The copy will only occur if the data matches the type of information that the field can receive.

L_WORD_LEFT—Moves the cursor to the beginning of the previous word.

L_WORD_RIGHT—Moves the cursor to the beginning of the next word.

APPENDIX D – CLASS IDENTIFIERS

This appendix contains a list of fixed constant values that are used as class identifiers. The definition of these constants is contained in **UI_GEN.HPP**.

General

The following identifications are general to Zinc Application Framework:

ID_DISPLAY—Identification for the UI_DISPLAY class.

ID_EVENT_MANAGER—Identification for the UI_EVENT_MANAGER class.

ID_WINDOW_MANAGER—Identification for the UI_WINDOW_MANAGER class.

ID_END—Identification that indicates the end of an array (used by **MapEvent( )** and **MapPalette( )**).

ID_SCREEN—Identification for the screen background.

ID_DIRECT—Identification for the screen directly, including any objects on the screen.

Window Objects

The following identifications apply to window objects:

ID_BIGNUM	UIW_BIGNUM class
ID_BORDER	UIW_BORDER class
ID_BUTTON	UIW_BUTTON class
ID_COMBO_BOX	UIW_COMBO_BOX class
ID_DATE	UIW_DATE class
ID_FORMATTED_STRING	UIW_FORMATTED_STRING class

ID_GROUP	UIW_GROUP class
ID_HZ_LIST	UIW_HZ_LIST class
ID_ICON	UIW_ICON class
ID_INTEGER	UIW_INTEGER class
ID_LIST	Used to tie the list classes
ID_MAXIMIZE_BUTTON	UIW_MAXIMIZE_BUTTON class
ID_MENU	Used to tie the menu classes
ID_MENU_ITEM	Used to tie the menu item classes
ID_MINIMIZE_BUTTON	UIW_MINIMIZE_BUTTON class
ID_NUMBER	Used to tie the number classes
ID_POP_UP_MENU	UIW_POP_UP_MENU class
ID_POP_UP_ITEM	UIW_POP_UP_ITEM class
ID_PROMPT	UIW_PROMPT class
ID_PULL_DOWN_MENU	UIW_PULL_DOWN_MENU class
ID_PULL_DOWN_ITEM	UIW_PULL_DOWN_ITEM class
ID_REAL	UIW_REAL class
ID_SCROLL_BAR	UIW_SCROLL_BAR class
ID_STRING	UIW_STRING class
ID_SYSTEM_BUTTON	UIW_SYSTEM_BUTTON class
ID_TEXT	UIW_TEXT class
ID_TIME	UIW_TIME class
ID_TITLE	UIW_TITLE class

ID_TOOL_BAR	UIW_TOOL_BAR class
ID_VT_LIST	UIW_VT_LIST class
ID_WINDOW	UIW_WINDOW class
ID_WINDOW_OBJECT	UI_WINDOW_OBJECT class

Shadowing

The following identifications are used to determine the color of a shaded object (e.g., border, button):

ID_OUTLINE	Outline of the object
ID_WHITE_SHADOW	Top-left shadow (normal)
ID_LIGHT_SHADOW	Bottom-right shadow (normal)
ID_DARK_SHADOW	Top-left shadow (depressed)
ID_BLACK_SHADOW	Bottom-right shadow (depressed)

APPENDIX E – ZINC OBJECT STORAGE

This appendix describes the file layout for *<file>*.**DAT** files. These files are created by the Interactive Design Tool whenever the "File, Save" or "File, save As" option is selected.

File Information

Each **.DAT** file contains all objects created and saved by Zinc Designer. Each file is organized in the following manner:

Zinc Signature
Revision Number
UIW_WINDOW directory
- contains definitions for UIW_WINDOW as well as the window's sub-objects.

UI_BITMAP directory
- contains bitmap data (i.e., name, height, width, bitmap array).

UI_ICON directory
- contains icon data (i.e., name, text, icon array).

UI_HELP directory
- contains help contexts (i.e., help context, title, message).

UI_HPP directory
- contains information used to create the **.HPP** file.

UI_CPP directory
- contains entries to connect window objects with their corresponding userFunction (specified in Zinc Designer).

Zinc signature is stored by the ZINC_SIGNATURE structure (defined in **STORE.CPP**):

```
struct ZINC_SIGNATURE
{
    char copyrightNotice[64];
    UCHAR majorVersion;
    UCHAR minorVersion;
    USHORT magicNumber;
};
```

UI_WINDOW_OBJECT

The UI_WINDOW_OBJECT class stores the following member variables:

numberID
stringID
woFlags
woAdvancedFlags
left
top
right
bottom
helpContext
userFlags
userStatus
userObjectName
userFunctionName

UIW_BIGNUM

The bignum class stores the following member variables, after calling **UIW_STRING::Store( )**:

nmFlags
range

NOTE: The bignum value is saved by storing the string representation of the bignum when **UIW_STRING::Store( )** is called.

UIW_BORDER

The border is stored as an attribute of UIW_WINDOW.

UIW_BUTTON

The UIW_BUTTON class stores the following member variables, after calling **UI_-WINDOW_OBJECT::Store( )**:

btFlags
value
depth

text
bitmapName

NOTE: If a bitmap is associated with the button, it is stored in the UI_BITMAP directory.

UIW_DATE

The UIW_DATE class stores the following member variables, after calling **UIW_-STRING::Store( )**:

dtFlags
range

NOTE: The date value is saved by storing the string representation of the date when **UIW_STRING::Store( )** is called.

UIW_FORMATTED_STRING

The UIW_FORMATTED_STRING class stores the following member variables, after calling **UIW_STRING::Store( )**:

compressedText
editText
deleteText

UIW_GROUP

The UIW_GROUP class stores the following member variable, after calling **UIW_-WINDOW::Store( )**:

text

UIW_HZ_LIST

The UIW_HZ_LIST class stores the following member variables, after calling **UIW_-WINDOW::Store( )**:

cellWidth
cellHeight

UIW_ICON

The UIW_ICON class stores the following member variables, after calling **UI_-WINDOW_OBJECT::Store( )**:

icFlags
title
iconName
iconWidth
iconHeight
iconArray

UIW_INTEGER

The UIW_INTEGER class stores the following member variables, after calling **UIW_-STRING::Store( )**:

nmFlags
range

NOTE: The integer value is saved by storing the string representation of the integer when **UIW_STRING::Store( )** is called.

UIW_MAXIMIZE_BUTTON

The UIW_MAXIMIZE_BUTTON class only stores its *search.type*, which is ID_-MAXIMIZE_BUTTON.

UIW_MINIMIZE_BUTTON

The UIW_MINIMIZE_BUTTON class only stores its *search.type*, which is ID_-MINIMIZE_BUTTON.

UIW_POP_UP_ITEM

The UIW_POP_UP_ITEM class stores the following member variable, after calling **UIW_BUTTON::Store( )**:

mniFlags

NOTE: The pop-up item also stores its associated menu (if any) by calling the menu's store function.

UIW_POP_UP_MENU

The UIW_POP_UP_MENU class does not store any member variables, it only calls **UIW_WINDOW::Store( )**

UIW_PROMPT

The UIW_PROMPT class stores the following member variable, after calling **UI_-WINDOW_OBJECT::Store( )**:

text

UIW_PULL_DOWN_ITEM

The UIW_PULL_DOWN_ITEM class calls **UIW_BUTTON::Store( )** then *menu*.**Store( )** (i.e., the store function associated with the menu member variable). It does not store any information of its own.

UIW_PULL_DOWN_MENU

The UIW_POP_UP_MENU class stores the following member variable, after calling **UIW_WINDOW::Store( )**:

indentation

UIW_REAL

The UIW_REAL class stores the following member variables, after calling **UIW_-STRING::Store( )**:

nmFlags
range

NOTE: The real value is saved by storing the string representation of the real when **UIW_STRING::Store( )** is called.

UIW_SCROLL_BAR

The UIW_SCROLL_BAR class stores the following member variables, after calling **UIW_WINDOW::Store( )**:

sbFlags
minimum
maximum
current

UIW_STRING

The UIW_STRING class stores the following member variables, after calling **UI_-WINDOW_OBJECT::Store( )**:

stFlags
maxLength
text

UIW_SYSTEM_BUTTON

The UIW_SYSTEM_BUTTON class calls **UIW_BUTTON::Store( )** then stores the following member variables:

syFlags
menu (menu calls the its associated **Store( )** function.

UIW_TEXT

The UIW_TEXT class stores the following member variables, after calling **UIW_-STRING::Store( )**:

maxLength
text
noOfObjects
_value (This value is stored for each line of the text field.)
object (This value is stored for each line of the text field.)
wnFlags

UIW_TIME

The UIW_TIME class stores the following member variables, after calling **UIW_-STRING::Store( )**:

tmFlags
range

NOTE: The time value is saved by storing the string representation of the time when **UIW_STRING::Store( )** is called.

UIW_TITLE

The UIW_TITLE class stores the following member variable:

text

UIW_VT_LIST

The UIW_VT_LIST class calls **UIW_WINDOW::Store( )**, it does not store any variables of its own.

UIW_WINDOW

The UIW_WINDOW class:

1—Checks for a valid directory (i.e., in the file) and disk file. If the file does not exist (i.e., a new file), the file is created and the following variables are stored:

miniNumeratorX
miniDenominatorX
miniNumeratorY
miniDenominatorY

2—The window and each of the sub-objects are stored. First **UI_-WINDOW_OBJECT::Store( )** is called to store the window and then the following member variables are stored:

noOfObjects (i.e., number of objects attached to the window)

Support Objects (The *object->searchID* is stored for each object and then the **object->Store( )** is called. This is done for all of the window's support objects.)

Regular Objects (The *object->searchID* is stored for each object and then the **object->Store( )** is called. This is done for all of the window's objects that are not in the support list.)

wnFlags
compareFunctionName

3—Write out the header information. Header information (used to re-construct the object) is stored for each object in the window.

4—User and compare function names are stored together with an logical link to the objects to which they are attached.

APPENDIX F – RAW SCAN CODES

This appendix contains a listing of the raw DOS, MS Windows, OS/2 and Motif scan codes used to map physical keyboard input into logical Zinc events. You must define *USE_RAW_KEYS* in order to get access to the scan codes (their definition is protected by #ifdef USE_RAW_KEYS). These raw scan codes are defined in **UI_EVT.HPP**:

```
#if defined(USE_RAW_KEYS)
#if defined(ZIL_MSDOS) || defined(ZIL_OS2)
const RAW_CODE ESCAPE                   = 0x011B;    // Miscellaneous keys
const RAW_CODE ENTER                    = 0x1C0D;
const RAW_CODE SPACE                    = 0x3920;
const RAW_CODE BACKSPACE                = 0x0E08;
const RAW_CODE CTRL_BACKSPACE           = 0x0E7F;
const RAW_CODE TAB                      = 0x0F09;
const RAW_CODE CTRL_TAB                 = 0x9400;
const RAW_CODE BACKTAB                  = 0x0F00;

const RAW_CODE SHIFT_GRAY_DELETE        = 0x53E0;
const RAW_CODE SHIFT_GRAY_INSERT        = 0x52E0;
const RAW_CODE CTRL_BREAK               = 0x0000;
const RAW_CODE CTRL_C                   = 0x2E03;

const RAW_CODE ALT_ESCAPE               = 0x0100;
const RAW_CODE ALT_PERIOD               = 0x3400;
const RAW_CODE ALT_SPACE                = 0x3920;
const RAW_CODE ALT_WHITE_MINUS          = 0x8200;
const RAW_CODE ALT_WHITE_PLUS           = 0x8300;

const RAW_CODE ALT_A                    = 0x1E00;
const RAW_CODE ALT_B                    = 0x3000;
const RAW_CODE ALT_C                    = 0x2E00;
const RAW_CODE ALT_D                    = 0x2000;
const RAW_CODE ALT_E                    = 0x1200;
const RAW_CODE ALT_F                    = 0x2100;
const RAW_CODE ALT_G                    = 0x2200;
const RAW_CODE ALT_H                    = 0x2300;
const RAW_CODE ALT_I                    = 0x1700;
const RAW_CODE ALT_J                    = 0x2400;
const RAW_CODE ALT_K                    = 0x2500;
const RAW_CODE ALT_L                    = 0x2600;
const RAW_CODE ALT_M                    = 0x3200;
const RAW_CODE ALT_N                    = 0x3100;
const RAW_CODE ALT_O                    = 0x1800;
const RAW_CODE ALT_P                    = 0x1900;
const RAW_CODE ALT_Q                    = 0x1000;
const RAW_CODE ALT_R                    = 0x1300;
const RAW_CODE ALT_S                    = 0x1F00;
const RAW_CODE ALT_T                    = 0x1400;
const RAW_CODE ALT_U                    = 0x1600;
const RAW_CODE ALT_V                    = 0x2F00;
const RAW_CODE ALT_W                    = 0x1100;
const RAW_CODE ALT_X                    = 0x2D00;
const RAW_CODE ALT_Y                    = 0x1500;
const RAW_CODE ALT_Z                    = 0x2C00;

const RAW_CODE ALT_1                    = 0x7800;
const RAW_CODE ALT_2                    = 0x7900;
const RAW_CODE ALT_3                    = 0x7A00;
const RAW_CODE ALT_4                    = 0x7B00;
const RAW_CODE ALT_5                    = 0x7C00;
const RAW_CODE ALT_6                    = 0x7D00;
```

```
const RAW_CODE ALT_7                    = 0x7E00;
const RAW_CODE ALT_8                    = 0x7F00;
const RAW_CODE ALT_9                    = 0x8000;
const RAW_CODE ALT_0                    = 0x8100;

const RAW_CODE GRAY_ENTER               = 0xE00D;   // Grey keys
const RAW_CODE GRAY_UP_ARROW            = 0x48E0;
const RAW_CODE GRAY_DOWN_ARROW          = 0x50E0;
const RAW_CODE GRAY_LEFT_ARROW          = 0x4BE0;
const RAW_CODE GRAY_RIGHT_ARROW         = 0x4DE0;
const RAW_CODE GRAY_INSERT              = 0x52E0;
const RAW_CODE GRAY_DELETE              = 0x53E0;
const RAW_CODE GRAY_HOME                = 0x47E0;
const RAW_CODE GRAY_END                 = 0x4FE0;
const RAW_CODE GRAY_PGUP                = 0x49E0;
const RAW_CODE GRAY_PGDN                = 0x51E0;
const RAW_CODE GRAY_DIVIDE              = 0xE02F;
const RAW_CODE GRAY_MULTIPLY            = 0x372A;
const RAW_CODE GRAY_PLUS                = 0x4E2B;
const RAW_CODE GRAY_MINUS               = 0x4A2D;

const RAW_CODE CTRL_GRAY_UP_ARROW       = 0x8DE0;
const RAW_CODE CTRL_GRAY_DOWN_ARROW     = 0x91E0;
const RAW_CODE CTRL_GRAY_LEFT_ARROW     = 0x73E0;
const RAW_CODE CTRL_GRAY_RIGHT_ARROW    = 0x74E0;
const RAW_CODE CTRL_GRAY_INSERT         = 0x92E0;
const RAW_CODE CTRL_GRAY_DELETE         = 0x93E0;
const RAW_CODE CTRL_GRAY_HOME           = 0x77E0;
const RAW_CODE CTRL_GRAY_END            = 0x75E0;
const RAW_CODE CTRL_GRAY_PGUP           = 0x84E0;
const RAW_CODE CTRL_GRAY_PGDN           = 0x76E0;
const RAW_CODE CTRL_GRAY_DIVIDE         = 0x9500;
const RAW_CODE CTRL_GRAY_MULTIPLY       = 0x9600;
const RAW_CODE CTRL_GRAY_PLUS           = 0x9000;
const RAW_CODE CTRL_GRAY_MINUS          = 0x8E00;

const RAW_CODE ALT_GRAY_UP_ARROW        = 0x9800;
const RAW_CODE ALT_GRAY_DOWN_ARROW      = 0xA000;
const RAW_CODE ALT_GRAY_LEFT_ARROW      = 0x9B00;
const RAW_CODE ALT_GRAY_RIGHT_ARROW     = 0x9D00;
const RAW_CODE ALT_GRAY_INSERT          = 0xA200;
const RAW_CODE ALT_GRAY_DELETE          = 0xA300;
const RAW_CODE ALT_GRAY_HOME            = 0x9700;
const RAW_CODE ALT_GRAY_END             = 0x9F00;
const RAW_CODE ALT_GRAY_PGUP            = 0x9900;
const RAW_CODE ALT_GRAY_PGDN            = 0xA100;
const RAW_CODE ALT_GRAY_DIVIDE          = 0xA400;
const RAW_CODE ALT_GRAY_MULTIPLY        = 0x3700;
const RAW_CODE ALT_GRAY_PLUS            = 0x4E00;
const RAW_CODE ALT_GRAY_MINUS           = 0x4A00;

const RAW_CODE WHITE_UP_ARROW           = 0x4800;   // White keys
const RAW_CODE WHITE_DOWN_ARROW         = 0x5000;
const RAW_CODE WHITE_LEFT_ARROW         = 0x4B00;
const RAW_CODE WHITE_RIGHT_ARROW        = 0x4D00;
const RAW_CODE WHITE_INSERT             = 0x5200;
const RAW_CODE WHITE_DELETE             = 0x5300;
const RAW_CODE WHITE_HOME               = 0x4700;
const RAW_CODE WHITE_END                = 0x4F00;
const RAW_CODE WHITE_PGUP               = 0x4900;
const RAW_CODE WHITE_PGDN               = 0x5100;
const RAW_CODE WHITE_CENTER             = 0x4C00;

const RAW_CODE CTRL_WHITE_UP_ARROW      = 0x8D00;
const RAW_CODE CTRL_WHITE_DOWN_ARROW    = 0x9100;
const RAW_CODE CTRL_WHITE_LEFT_ARROW    = 0x7300;
const RAW_CODE CTRL_WHITE_RIGHT_ARROW   = 0x7400;
const RAW_CODE CTRL_WHITE_INSERT        = 0x9200;
```

```
const RAW_CODE CTRL_WHITE_DELETE        = 0x9300;
const RAW_CODE CTRL_WHITE_HOME          = 0x7700;
const RAW_CODE CTRL_WHITE_END           = 0x7500;
const RAW_CODE CTRL_WHITE_PGUP          = 0x8400;
const RAW_CODE CTRL_WHITE_PGDN          = 0x7600;
const RAW_CODE CTRL_WHITE_CENTER        = 0x8F00;

const RAW_CODE F1                       = 0x3B00;   // Function keys
const RAW_CODE F2                       = 0x3C00;
const RAW_CODE F3                       = 0x3D00;
const RAW_CODE F4                       = 0x3E00;
const RAW_CODE F5                       = 0x3F00;
const RAW_CODE F6                       = 0x4000;
const RAW_CODE F7                       = 0x4100;
const RAW_CODE F8                       = 0x4200;
const RAW_CODE F9                       = 0x4300;
const RAW_CODE F10                      = 0x4400;
const RAW_CODE F11                      = 0x8500;
const RAW_CODE F12                      = 0x8600;

const RAW_CODE SHIFT_F1                 = 0x5400;
const RAW_CODE SHIFT_F2                 = 0x5500;
const RAW_CODE SHIFT_F3                 = 0x5600;
const RAW_CODE SHIFT_F4                 = 0x5700;
const RAW_CODE SHIFT_F5                 = 0x5800;
const RAW_CODE SHIFT_F6                 = 0x5900;
const RAW_CODE SHIFT_F7                 = 0x5A00;
const RAW_CODE SHIFT_F8                 = 0x5B00;
const RAW_CODE SHIFT_F9                 = 0x5C00;
const RAW_CODE SHIFT_F10                = 0x5D00;
const RAW_CODE SHIFT_F11                = 0x8700;
const RAW_CODE SHIFT_F12                = 0x8800;

const RAW_CODE CTRL_F1                  = 0x5E00;
const RAW_CODE CTRL_F2                  = 0x5F00;
const RAW_CODE CTRL_F3                  = 0x6000;
const RAW_CODE CTRL_F4                  = 0x6100;
const RAW_CODE CTRL_F5                  = 0x6200;
const RAW_CODE CTRL_F6                  = 0x6300;
const RAW_CODE CTRL_F7                  = 0x6400;
const RAW_CODE CTRL_F8                  = 0x6500;
const RAW_CODE CTRL_F9                  = 0x6600;
const RAW_CODE CTRL_F10                 = 0x6700;
const RAW_CODE CTRL_F11                 = 0x8900;
const RAW_CODE CTRL_F12                 = 0x8A00;

const RAW_CODE ALT_F1                   = 0x6800;
const RAW_CODE ALT_F2                   = 0x6900;
const RAW_CODE ALT_F3                   = 0x6A00;
const RAW_CODE ALT_F4                   = 0x6B00;
const RAW_CODE ALT_F5                   = 0x6C00;
const RAW_CODE ALT_F6                   = 0x6D00;
const RAW_CODE ALT_F7                   = 0x6E00;
const RAW_CODE ALT_F8                   = 0x6F00;
const RAW_CODE ALT_F9                   = 0x7000;
const RAW_CODE ALT_F10                  = 0x7100;
const RAW_CODE ALT_F11                  = 0x8B00;
const RAW_CODE ALT_F12                  = 0x8C00;
#elif defined(ZIL_MSWINDOWS)
const RAW_CODE ESCAPE                   = 0x001B;   // Miscellaneous keys
const RAW_CODE ENTER                    = 0x000D;
const RAW_CODE BACKSPACE                = 0x0008;
const RAW_CODE CTRL_BACKSPACE           = 0x0408;
const RAW_CODE TAB                      = 0x0009;
const RAW_CODE CTRL_TAB                 = 0x0409;
const RAW_CODE BACKTAB                  = 0x0309;
const RAW_CODE CTRL_BREAK               = 0x0403;
const RAW_CODE CTRL_C                   = 0x0443;
```

```
const RAW_CODE ALT_ESCAPE               = 0x081B;
const RAW_CODE ALT_PERIOD               = 0x08BE;
const RAW_CODE ALT_SPACE                = 0x0820;
const RAW_CODE ALT_WHITE_MINUS          = 0x08BD;
const RAW_CODE ALT_WHITE_PLUS           = 0x08BB;

const RAW_CODE ALT_A                    = 0x0861;
const RAW_CODE ALT_B                    = 0x0862;
const RAW_CODE ALT_C                    = 0x0863;
const RAW_CODE ALT_D                    = 0x0864;
const RAW_CODE ALT_E                    = 0x0865;
const RAW_CODE ALT_F                    = 0x0866;
const RAW_CODE ALT_G                    = 0x0867;
const RAW_CODE ALT_H                    = 0x0868;
const RAW_CODE ALT_I                    = 0x0869;
const RAW_CODE ALT_J                    = 0x086A;
const RAW_CODE ALT_K                    = 0x086B;
const RAW_CODE ALT_L                    = 0x086C;
const RAW_CODE ALT_M                    = 0x086D;
const RAW_CODE ALT_N                    = 0x086E;
const RAW_CODE ALT_O                    = 0x086F;
const RAW_CODE ALT_P                    = 0x0870;
const RAW_CODE ALT_Q                    = 0x0871;
const RAW_CODE ALT_R                    = 0x0872;
const RAW_CODE ALT_S                    = 0x0873;
const RAW_CODE ALT_T                    = 0x0874;
const RAW_CODE ALT_U                    = 0x0875;
const RAW_CODE ALT_V                    = 0x0876;
const RAW_CODE ALT_W                    = 0x0877;
const RAW_CODE ALT_X                    = 0x0878;
const RAW_CODE ALT_Y                    = 0x0879;
const RAW_CODE ALT_Z                    = 0x087A;

const RAW_CODE ALT_1                    = 0x0831;
const RAW_CODE ALT_2                    = 0x0832;
const RAW_CODE ALT_3                    = 0x0833;
const RAW_CODE ALT_4                    = 0x0834;
const RAW_CODE ALT_5                    = 0x0835;
const RAW_CODE ALT_6                    = 0x0836;
const RAW_CODE ALT_7                    = 0x0837;
const RAW_CODE ALT_8                    = 0x0838;
const RAW_CODE ALT_9                    = 0x0839;
const RAW_CODE ALT_0                    = 0x0840;

const RAW_CODE GRAY_ENTER               = 0x000D;   // Grey keys
const RAW_CODE GRAY_UP_ARROW            = 0x0026;
const RAW_CODE GRAY_DOWN_ARROW          = 0x0028;
const RAW_CODE GRAY_LEFT_ARROW          = 0x0025;
const RAW_CODE GRAY_RIGHT_ARROW         = 0x0027;
const RAW_CODE GRAY_INSERT              = 0x002D;
const RAW_CODE GRAY_DELETE              = 0x002E;
const RAW_CODE GRAY_HOME                = 0x0024;
const RAW_CODE GRAY_END                 = 0x0023;
const RAW_CODE GRAY_PGUP                = 0x0021;
const RAW_CODE GRAY_PGDN                = 0x0022;
const RAW_CODE GRAY_DIVIDE              = 0x006F;
const RAW_CODE GRAY_MULTIPLY            = 0x006A;
const RAW_CODE GRAY_PLUS                = 0x006B;
const RAW_CODE GRAY_MINUS               = 0x006D;

const RAW_CODE CTRL_GRAY_UP_ARROW       = 0x0426;
const RAW_CODE CTRL_GRAY_DOWN_ARROW     = 0x0428;
const RAW_CODE CTRL_GRAY_LEFT_ARROW     = 0x0425;
const RAW_CODE CTRL_GRAY_RIGHT_ARROW    = 0x0427;
const RAW_CODE CTRL_GRAY_INSERT         = 0x042D;
const RAW_CODE CTRL_GRAY_DELETE         = 0x042E;
const RAW_CODE CTRL_GRAY_HOME           = 0x0424;
const RAW_CODE CTRL_GRAY_END            = 0x0423;
```

```
const RAW_CODE CTRL_GRAY_PGUP           = 0x0421;
const RAW_CODE CTRL_GRAY_PGDN           = 0x0422;
const RAW_CODE CTRL_GRAY_DIVIDE         = 0x046F;
const RAW_CODE CTRL_GRAY_MULTIPLY       = 0x046A;
const RAW_CODE CTRL_GRAY_PLUS           = 0x046B;
const RAW_CODE CTRL_GRAY_MINUS          = 0x046D;

const RAW_CODE ALT_GRAY_UP_ARROW        = 0x0826;
const RAW_CODE ALT_GRAY_DOWN_ARROW      = 0x0828;
const RAW_CODE ALT_GRAY_LEFT_ARROW      = 0x0825;
const RAW_CODE ALT_GRAY_RIGHT_ARROW     = 0x0827;
const RAW_CODE ALT_GRAY_INSERT          = 0x082D;
const RAW_CODE ALT_GRAY_DELETE          = 0x082E;
const RAW_CODE ALT_GRAY_HOME            = 0x0824;
const RAW_CODE ALT_GRAY_END             = 0x0823;
const RAW_CODE ALT_GRAY_PGUP            = 0x0821;
const RAW_CODE ALT_GRAY_PGDN            = 0x0822;
const RAW_CODE ALT_GRAY_DIVIDE          = 0x086F;
const RAW_CODE ALT_GRAY_MULTIPLY        = 0x086A;
const RAW_CODE ALT_GRAY_PLUS            = 0x086B;
const RAW_CODE ALT_GRAY_MINUS           = 0x086D;

const RAW_CODE WHITE_UP_ARROW           = 0x0026;   // White keys
const RAW_CODE WHITE_DOWN_ARROW         = 0x0028;
const RAW_CODE WHITE_LEFT_ARROW         = 0x0025;
const RAW_CODE WHITE_RIGHT_ARROW        = 0x0027;
const RAW_CODE WHITE_INSERT             = 0x002D;
const RAW_CODE WHITE_DELETE             = 0x002E;
const RAW_CODE WHITE_HOME               = 0x0024;
const RAW_CODE WHITE_END                = 0x0023;
const RAW_CODE WHITE_PGUP               = 0x0021;
const RAW_CODE WHITE_PGDN               = 0x0022;
const RAW_CODE WHITE_CENTER             = 0x002A;

const RAW_CODE CTRL_WHITE_UP_ARROW      = 0x0426;
const RAW_CODE CTRL_WHITE_DOWN_ARROW    = 0x0428;
const RAW_CODE CTRL_WHITE_LEFT_ARROW    = 0x0425;
const RAW_CODE CTRL_WHITE_RIGHT_ARROW   = 0x0427;
const RAW_CODE CTRL_WHITE_INSERT        = 0x042D;
const RAW_CODE CTRL_WHITE_DELETE        = 0x042E;
const RAW_CODE CTRL_WHITE_HOME          = 0x0424;
const RAW_CODE CTRL_WHITE_END           = 0x0423;
const RAW_CODE CTRL_WHITE_PGUP          = 0x0421;
const RAW_CODE CTRL_WHITE_PGDN          = 0x0422;
const RAW_CODE CTRL_WHITE_CENTER        = 0x042A;

const RAW_CODE F1                       = 0x0070;   // Function keys
const RAW_CODE F2                       = 0x0071;
const RAW_CODE F3                       = 0x0072;
const RAW_CODE F4                       = 0x0073;
const RAW_CODE F5                       = 0x0074;
const RAW_CODE F6                       = 0x0075;
const RAW_CODE F7                       = 0x0076;
const RAW_CODE F8                       = 0x0077;
const RAW_CODE F9                       = 0x0078;
const RAW_CODE F10                      = 0x0079;
const RAW_CODE F11                      = 0x007A;
const RAW_CODE F12                      = 0x007B;

const RAW_CODE SHIFT_F1                 = 0x0370;
const RAW_CODE SHIFT_F2                 = 0x0371;
const RAW_CODE SHIFT_F3                 = 0x0372;
const RAW_CODE SHIFT_F4                 = 0x0373;
const RAW_CODE SHIFT_F5                 = 0x0374;
const RAW_CODE SHIFT_F6                 = 0x0375;
const RAW_CODE SHIFT_F7                 = 0x0376;
const RAW_CODE SHIFT_F8                 = 0x0377;
const RAW_CODE SHIFT_F9                 = 0x0378;
```

```
const RAW_CODE SHIFT_F10                = 0x0379;
const RAW_CODE SHIFT_F11                = 0x037A;
const RAW_CODE SHIFT_F12                = 0x037B;

const RAW_CODE CTRL_F1                  = 0x0470;
const RAW_CODE CTRL_F2                  = 0x0471;
const RAW_CODE CTRL_F3                  = 0x0472;
const RAW_CODE CTRL_F4                  = 0x0473;
const RAW_CODE CTRL_F5                  = 0x0474;
const RAW_CODE CTRL_F6                  = 0x0475;
const RAW_CODE CTRL_F7                  = 0x0476;
const RAW_CODE CTRL_F8                  = 0x0477;
const RAW_CODE CTRL_F9                  = 0x0478;
const RAW_CODE CTRL_F10                 = 0x0479;
const RAW_CODE CTRL_F11                 = 0x047A;
const RAW_CODE CTRL_F12                 = 0x047B;

const RAW_CODE ALT_F1                   = 0x0870;
const RAW_CODE ALT_F2                   = 0x0871;
const RAW_CODE ALT_F3                   = 0x0872;
const RAW_CODE ALT_F4                   = 0x0873;
const RAW_CODE ALT_F5                   = 0x0874;
const RAW_CODE ALT_F6                   = 0x0875;
const RAW_CODE ALT_F7                   = 0x0876;
const RAW_CODE ALT_F8                   = 0x0877;
const RAW_CODE ALT_F9                   = 0x0878;
const RAW_CODE ALT_F10                  = 0x0879;
const RAW_CODE ALT_F11                  = 0x087A;
const RAW_CODE ALT_F12                  = 0x087B;
#elif defined (ZIL_MOTIF)
const RAW_CODE ESCAPE                   = XK_Escape;    // Misc. keys
const RAW_CODE ENTER                    = XK_Return;
const RAW_CODE BACKSPACE                = XK_BackSpace;
const RAW_CODE TAB                      = XK_Tab;

const RAW_CODE GRAY_ENTER               = XK_KP_Enter;  // Grey keys
const RAW_CODE GRAY_UP_ARROW            = XK_Up;
const RAW_CODE GRAY_DOWN_ARROW          = XK_Down;
const RAW_CODE GRAY_LEFT_ARROW          = XK_Left;
const RAW_CODE GRAY_RIGHT_ARROW         = XK_Right;
const RAW_CODE GRAY_DIVIDE              = XK_KP_Divide;
const RAW_CODE GRAY_MULTIPLY            = XK_KP_Multiply;
const RAW_CODE GRAY_PLUS                = XK_KP_Add;
const RAW_CODE GRAY_MINUS               = XK_KP_Subtract;

const RAW_CODE WHITE_UP_ARROW           = XK_Up;    // White keys
const RAW_CODE WHITE_DOWN_ARROW         = XK_Down;
const RAW_CODE WHITE_LEFT_ARROW         = XK_Left;
const RAW_CODE WHITE_RIGHT_ARROW        = XK_Right;

const RAW_CODE F1                       = XK_F1;    // Function keys
const RAW_CODE F2                       = XK_F2;
const RAW_CODE F3                       = XK_F3;
const RAW_CODE F4                       = XK_F4;
const RAW_CODE F5                       = XK_F5;
const RAW_CODE F6                       = XK_F6;
const RAW_CODE F7                       = XK_F7;
const RAW_CODE F8                       = XK_F8;
const RAW_CODE F9                       = XK_F9;
const RAW_CODE F10                      = XK_F10;
const RAW_CODE F11                      = XK_F11;
const RAW_CODE F12                      = XK_F12;
#endif
#endif
```

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z