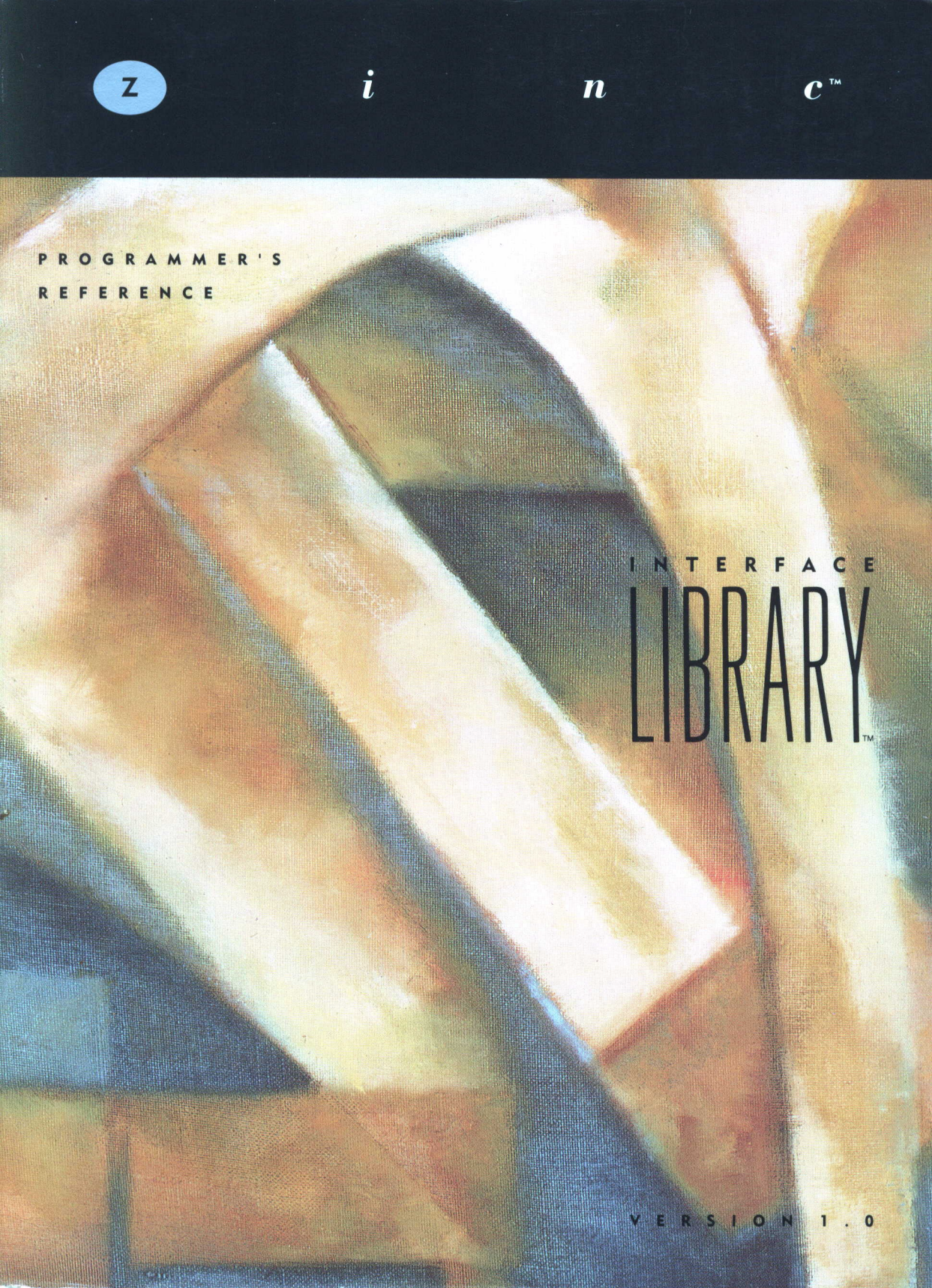

z i n c™
PROGRAMMER'S
REFERENCE
INTERFACE
LIBRARY™
VERSION 1.0

Zinc™ Interface Library™

Programmer's Reference

Version 1.0

Zinc Software Incorporated
Pleasant Grove, Utah

TABLE OF CONTENTS

INTRODUCTION

The Programmer's Reference contains descriptions of the Zinc Interface Library 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. It contains the following sections:

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

Class object references—This section (Chapters 1 through 48) contains short descriptions about the class objects (or structures), the available member variables and functions and the calling conventions used with the class object.

All other high-level information is contained in the Programmer's Guide. The programmer's guide is an overview document to the Zinc Interface Library. It contains the following sections:

Installation—This section (Chapter 1) tells how to install the Zinc Interface Library software package on your machine.

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

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

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

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

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

The remaining parts of this chapter give the programmer information about the classes, structures and global variables defined in the Zinc Interface Library.

CLASSES, STRUCTURES AND GLOBAL VARIABLES

General purpose

```
class UI_ELEMENT
class UI_DATE
class UI_LIST
class UI_TIME
```

Screen display

```
struct UI_PALETTE
struct UI_PALETTE_MAP

class UI_DISPLAY
class UI_DOS_BGI_DISPLAY
class UI_DOS_TEXT_DISPLAY

UI_PALETTE *_backgroundPalette
UI_PALETTE_MAP *_errorPaletteMapTable
UI_PALETTE_MAP *_helpPaletteMapTable
UI_PALETTE_MAP *_normalPaletteMapTable
```

Event management

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

class UI_BIOS_KEYBOARD
class UI_CURSOR
class UI_DEVICE
class UI_EVENT_MANAGER
class UI_MS_MOUSE

UI_EVENT_MAP *_eventMapTable
UI_EVENT_MAP *_hotKeyMapTable
```

Window management

```
class UI_WINDOW_MANAGER
class UI_WINDOW_OBJECT
class UIW_BORDER
class UIW_BUTTON
class UIW_DATE
class UIW_FORMATTED_STRING
class UIW_ICON
class UIW_MATRIX
class UIW_MAXIMIZE_BUTTON
class UIW_MINIMIZE_BUTTON
class UIW_NUMBER
class UIW_POP_UP_ITEM
class UIW_POP_UP_MENU
class UIW_POP_UP_WINDOW
class UIW_PROMPT
class UIW_PULL_DOWN_ITEM
class UIW_PULL_DOWN_MENU
class UIW_STRING
class UIW_SYSTEM_BUTTON
class UIW_TEXT
class UIW_TIME
class UIW_TITLE
class UIW_WINDOW
```

Error system

```
class UI_ERROR_SYSTEM
class UI_ERROR_WINDOW_SYSTEM

UI_ERROR_SYSTEM *_errorSystem;
```

Help system

```
class UI_HELP_SYSTEM
class UI_HELP_WINDOW_SYSTEM

UI_HELP_SYSTEM *_helpSystem;
```

INCLUDE FILE HIERARCHY

UI_GEN.HPP

```
#ifndef UI_GEN_HPP
#define UI_GEN_HPP

    // struct UI_POSITION
    // struct UI_REGION

    // class UI_DATE
    // class UI_ELEMENT
    // class UI_LIST
    // class UI_TIME

#endif
```

UI_DSP.HPP

```
#ifndef UI_DSP_HPP
#define UI_DSP_HPP
#include <UI_GEN.HPP>

    // struct UI_PALETTE

    // class UI_DISPLAY
    // class UI_DOS_BGI_DISPLAY
    // class UI_DOS_TEXT_DISPLAY

    // extern UI_PALETTE *_backgroundPalette;

#include <UI_EVT.HPP>
#endif
```

UI_EVT.HPP

```
#ifndef UI_EVT_HPP
#define UI_EVT_HPP
#include <UI_DSP.HPP>

    // struct UI_EVENT
    // struct UI_KEY

    // class UI_BIOS_KEYBOARD
    // class UI_CURSOR
    // class UI_DEVICE
    // class UI_EVENT_MANAGER
    // class UI_MS_MOUSE

#endif
```

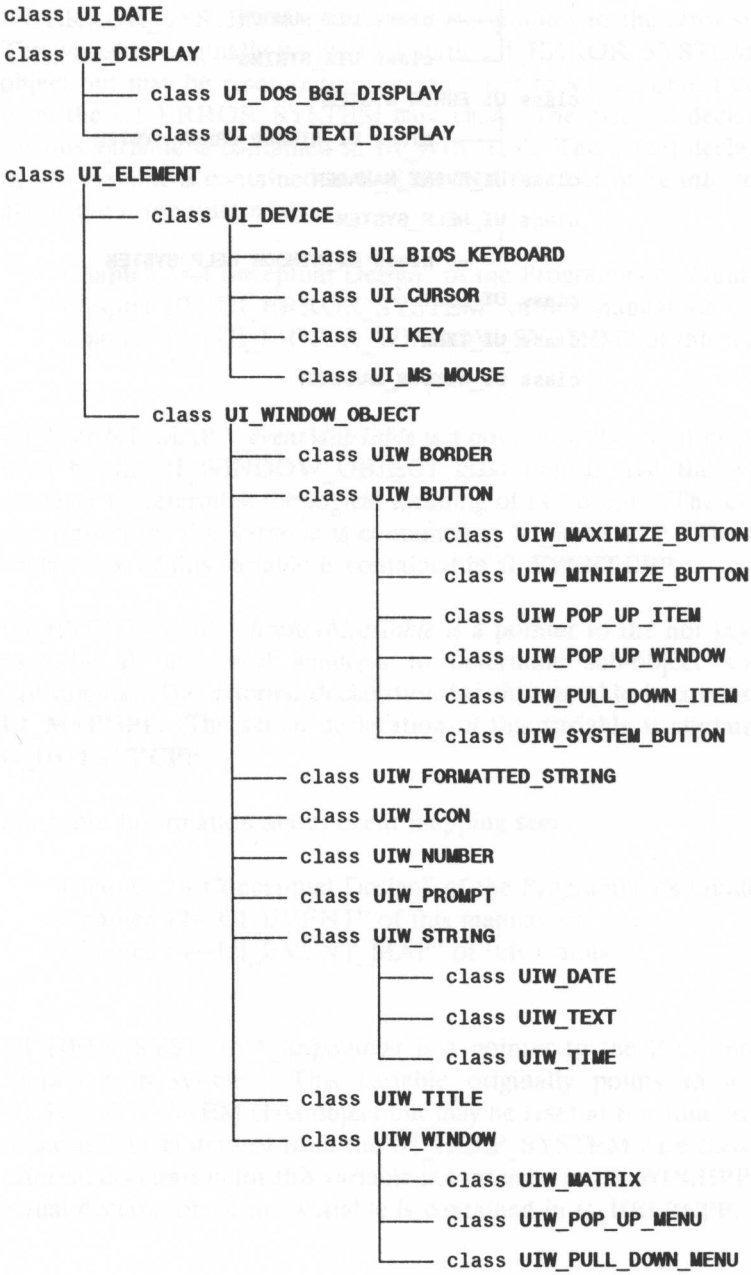
UI_MAP.HPP

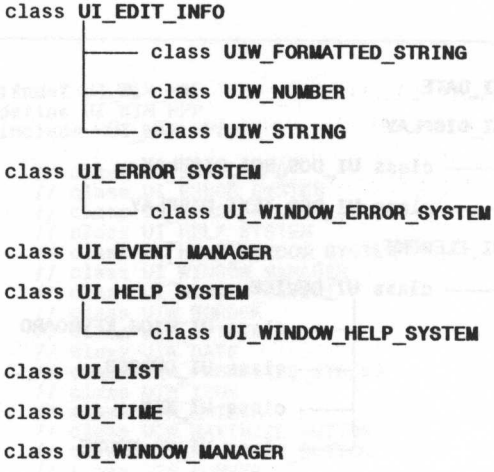
```
#ifndef UI_MAP_HPP
#define UI_MAP_HPP
#include <UI_EVT.HPP>

    // struct UI_EVENT_MAP
    // struct UI_PALETTE_MAP

    // extern UI_EVENT_MAP *_eventMapTable;
    // extern UI_EVENT_MAP *_hotKeyMapTable;
    // extern UI_PALETTE_MAP *_normalPaletteMapTable;
    // extern UI_PALETTE_MAP *_errorPaletteMapTable;
    // extern UI_PALETTE_MAP *_helpPaletteMapTable;

#endif
```

UI_WIN.HPP

```
#ifndef UI_WIN_HPP
#define UI_WIN_HPP
#include <UI_MAP.HPP>

    // class UI_EDIT_INFO
    // class UI_ERROR_SYSTEM
    // class UI_ERROR_WINDOW_SYSTEM
    // class UI_HELP_SYSTEM
    // class UI_HELP_WINDOW_SYSTEM
    // class UI_WINDOW_MANAGER
    // class UI_WINDOW_OBJECT
    // class UIW_BORDER
    // class UIW_BUTTON
    // class UIW_DATE
    // class UIW_FORMATTED_STRING
    // class UIW_ICON
    // class UIW_MATRIX
    // class UIW_MAXIMIZE_BUTTON
    // class UIW_MINIMIZE_BUTTON
    // class UIW_NUMBER
    // class UIW_POP_UP_ITEM
    // class UIW_POP_UP_MENU
    // class UIW_POP_UP_WINDOW
    // class UIW_PROMPT
    // class UIW_PULL_DOWN_ITEM
    // class UIW_PULL_DOWN_MENU
    // class UIW_STRING
    // class UIW_SYSTEM_BUTTON
    // class UIW_TEXT
    // class UIW_TIME
    // class UIW_TITLE
    // class UIW_WINDOW

    // extern UI_ERROR_SYSTEM *_errorSystem;
    // extern UI_HELP_SYSTEM *_helpSystem;

#endif
```

CLASS HIERARCHY




class UI_EDIT_INFO
class UIW_FORMATTED_STRING
class UIW_NUMBER
class UIW_STRING
class UI_ERROR_SYSTEM
class UI_WINDOW_ERROR_SYSTEM
class UI_EVENT_MANAGER
class UI_HELP_SYSTEM
class UI_WINDOW_HELP_SYSTEM
class UI_LIST
class UI_TIME
class UI_WINDOW_MANAGER

GLOBAL VARIABLES

Error system

UI_ERROR_SYSTEM *_*errorSystem*_ is a pointer to the error system. This variable originally points to a static UI_ERROR_SYSTEM class object but may be reset at run-time to point to a class object derived from the UI_ERROR_SYSTEM base class. The external declaration for this variable is contained in **UI_WIN.HPP**. The actual declaration of this variable is contained in **G_ERROR.CPP**. For more information about the error system see:

"Chapter 2—Conceptual Design" of the Programmer's Guide,
"Chapter 10—UI_ERROR_SYSTEM" of this manual, or
"Chapter 11—UI_ERROR_WINDOW_SYSTEM" of this manual.

Event mapping

UI_EVENT_MAP *_*eventMapTable*_ is a pointer to the event map table used by all UI_WINDOW_OBJECT class objects and the window manager to determine the logical meaning of raw events. The external declaration for this variable is contained in **UI_MAP.HPP**. The actual declaration of this variable is contained in **G_EVENT.CPP**.

UI_EVENT_MAP *_*hotKeyMapTable*_ is a pointer to the hot key table used by all high-level windows to determine sub-object hot key equivalents. The external declaration for this variable is contained in **UI_MAP.HPP**. The actual declaration of this variable is contained in **G_HOTKEY.CPP**.

For more information about event mapping see:

"Chapter 2—Conceptual Design" of the Programmer's Guide,
"Chapter 12—UI_EVENT" of this manual, or
"Chapter 14—UI_EVENT_MAP" of this manual.

Help system

UI_HELP_SYSTEM *_*helpSystem*_ is a pointer to the Zinc Interface Library help system. This variable originally points to a static UI_HELP_SYSTEM class object but may be reset at run-time to point to a class object derived from the UI_HELP_SYSTEM base class. The external declaration for this variable is contained in **UI_WIN.HPP**. The actual declaration of this variable is contained in **G_HELP.CPP**.

For more information about the help system see:

"Chapter 15—UI_HELP_SYSTEM" or
"Chapter 16—UI_HELP_WINDOW_SYSTEM"

of this manual.

Palette mapping

UI_PALETTE **_backgroundPalette* is a pointer to the background palette. The external declaration for this variable is contained in **UI_DSP.HPP**. The actual declaration of this variable is contained in **G_PBACK.CPP**.

UI_PALETTE_MAP **_errorPaletteMapTable* is a pointer to the error palette table. This is the palette table used by the error system. The external declaration for this variable is contained in **UI_MAP.HPP**. The actual declaration of this variable is contained in **G_PERROR.CPP**.

UI_PALETTE_MAP **_helpPaletteMapTable* is a pointer to the help palette table. This is the palette table used by the help system. The external declaration for this variable is contained in **UI_MAP.HPP**. The actual declaration of this variable is contained in **G_PHELP.CPP**.

UI_PALETTE_MAP **_normalPaletteMapTable* is a pointer to the normal palette table. This is the default palette table used by all window objects. The external declaration for this variable is contained in **UI_MAP.HPP**. The actual declaration of this variable is contained in **G_PNORM.CPP**.

For more information about palette mapping see:

"Chapter 2—Conceptual Design" of the Programmer's Guide,
"Chapter 19—UI_PALETTE" of this manual, or
"Chapter 20—UI_PALETTE_MAP" of this manual.

CHAPTER 1 – UI_BIOS_KEYBOARD

Overview

The UI_BIOS_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 the Zinc Interface Library, the keyboard is interfaced with other devices, such as a mouse, to provide smooth control of the user's input. This interface is achieved through the event manager, where the UI_BIOS_KEYBOARD class is one of many input devices that feed event information to the event manager's event queue. The public members of the UI_BIOS_KEYBOARD class (declared in **UI_EVT.HPP**) are:

```
class UI_BIOS_KEYBOARD : public UI_DEVICE
{
public:
    UI_BIOS_KEYBOARD(USHORT initialState = D_ON);
    virtual ~UI_BIOS_KEYBOARD(void);
};
```

Keyboard event information

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 UI_KEY
{
    UCHAR shiftState;       // The keyboard's shift state.
    UCHAR value;            // The key's ascii value.
};

struct UI_EVENT
{
    int type;               // The type of event (E_KEY).
    USHORT rawCode;         // The key's raw scan code.
    union
    {
        UI_KEY key;         // The key information.
        UI_REGION region;
        UI_POSITION position;
        void *data;
    };
};
```

- *type* is the event type. The 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/constant equivalents is provided in **UI_MAP.HPP**. For example, the function keys shown above have the following constant declarations:

```
const USHORT ENTER          =   0x1C0D;
const USHORT ESCAPE         =   0x011B;
const USHORT F1             =   0x3B00;
const USHORT GRAY_UP_ARROW  =   0x48E0;
```

- *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 *key.scanCode*.

See also The example file **XBIOSKEY.CPP**, which gives a complete example of the UI_BIOS_KEYBOARD class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the operation of device classes within the event manager.

"Chapter 2—UI_CURSOR" of this manual, which describes an additional device derived from the UI_DEVICE class.

"Chapter 4—UI_DEVICE" of this manual, which describes the base class from which the UI_BIOS_KEYBOARD class is derived.

"Chapter 13—UI_EVENT_MANAGER" of this manual, which describes the operation (e.g., addition, subtraction, state change) of device classes within the event manager.

"Chapter 18—UI_MS_MOUSE" of this manual, which describes an additional device derived from the UI_DEVICE class.

UI_BIOS_KEYBOARD::UI_BIOS_KEYBOARD

Syntax

```
#include <ui_evt.h>

UI_BIOS_KEYBOARD::UI_BIOS_KEYBOARD(
    USHORT initialState = D_ON);
```

Remarks This constructor returns a pointer to a new UI_BIOS_KEYBOARD class object. It should be called after the following class constructors have been called:

1—UI_DOS_BGI_DISPLAY *or* UI_DOS_TEXT_DISPLAY, then

2—UI_EVENT_MANAGER

NOTE: If the keyboard device is attached to the event manager, it will automatically be destroyed when the event manager is destroyed.

- *initialState*$_{in}$ is the initial state of the keyboard device. The keyboard device may be set 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 not placed in the event queue.

  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 UI_BIOS_KEYBOARD device can be changed at run-time using the **UI_EVENT_MANAGER::DeviceState** function call.

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    UI_EVENT_MANAGER eventManager(100, &display);

    // Default to the ON state.
    UI_BIOS_KEYBOARD *keyboard = new UI_BIOS_KEYBOARD;
    eventManager + keyboard;
    .
    .
    .
}

ExampleFunction2()
{
    // Attach the keyboard directly to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager + new UI_BIOS_KEYBOARD;
    .
    .
    .
}
```

UI_BIOS_KEYBOARD::~UI_BIOS_KEYBOARD

Syntax

```
#include <ui_evt.hpp>

virtual UI_BIOS_KEYBOARD::~UI_BIOS_KEYBOARD(void);
```

Remarks This virtual destructor destroys the class information associated with the UI_BIOS_KEYBOARD object and closes the BIOS interface to the keyboard. Care should be taken to only destroy a keyboard device that is not attached to the event manager.

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    UI_EVENT_MANAGER eventManager(100, &display);

    // Default to the ON state.
    UI_BIOS_KEYBOARD *keyboard = new UI_BIOS_KEYBOARD;
    eventManager + keyboard;
    .
    .
    .
    // Remove the keyboard device from the event manager manually.
    // This operation is not necessary since it would automatically
    // be destroyed by the event manager (see ExampleFunction2).
    eventManager - keyboard;
    delete keyboard;
}

ExampleFunction2()
{
    // Attach the keyboard directly to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager + new UI_BIOS_KEYBOARD;
    .
    .
    .
    // This automatically calls the keyboard destructor.
    delete eventManager;
    delete display;
}
```

CHAPTER 2 – UI_CURSOR

Overview

The UI_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 graphics mode, this class paints a blinking cursor on the screen. In text mode, this class uses BIOS calls to enable or disable the blinking hardware cursor. The public members of the UI_CURSOR class (declared in **UI_EVT.HPP**) are:

```
class UI_CURSOR : public UI_DEVICE
{
public:
    UI_CURSOR(USHORT initialState = D_OFF);
    virtual ~UI_CURSOR(void);
};
```

See also

The example file **XCURSOR.CPP**, which gives a complete example of the UI_CURSOR class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the operation of device classes within the event manager.

"Chapter 1—UI_BIOS_KEYBOARD" of this manual, which describes an additional device derived from the UI_DEVICE class.

"Chapter 4—UI_DEVICE class" of this manual, which describes the base class from which the UI_CURSOR class is derived.

"Chapter 13—UI_EVENT_MANAGER" of this manual, which describes the operation (e.g., addition, subtraction, state change) of device classes within the event manager.

"Chapter 18—UI_MS_MOUSE" of this manual, which describes an additional device derived from the UI_DEVICE class.

UI_CURSOR::UI_CURSOR

Syntax #include <ui_evt.hpp>

UI_CURSOR::UI_CURSOR(USHORT *initialState* = D_OFF);

Remarks This constructor returns a pointer to a new UI_CURSOR class object. It should be called after the following class constructors have been called:

1—UI_DOS_BGI_DISPLAY *or* UI_DOS_TEXT_DISPLAY, then

2—UI_EVENT_MANAGER

NOTE: If the cursor device is attached to the event manager, it will automatically be destroyed when the event manager is destroyed.

- *initialState*$_{in}$ is the initial state of the cursor device. The cursor device may be set 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 state of the UI_CURSOR device can be changed at run-time using the **UI_EVENT_MANAGER::DeviceState** function call.

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    UI_EVENT_MANAGER eventManager(100, &display);

    // Initialize the cursor.
    UI_CURSOR *cursor = new UI_CURSOR;
    eventManager + cursor;
    .
    .
    .
}
```

```
ExampleFunction2()
{
    // Attach the devices directly to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager
        + new UI_BIOS_KEYBOARD
        + new UI_MS_MOUSE
        + new UI_CURSOR;
    .
    .
    .
}
```

UI_CURSOR::~UI_CURSOR

Syntax #include <ui_evt.hpp>

virtual UI_CURSOR::~UI_CURSOR(void);

Remarks This virtual destructor destroys the class information associated with the UI_CURSOR object. Care should be taken to only destroy a cursor device that is not attached to the event manager.

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    UI_EVENT_MANAGER eventManager(100, display);

    // Initialize the cursor.
    UI_CURSOR *cursor = new UI_CURSOR;
    eventManager + cursor;
    .
    .
    .
    // Remove the cursor from the event manager, then call
    // its destructor.
    eventManager - cursor;
    delete cursor;
}
```

```
ExampleFunction2()
{
    // Attach the devices directly to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager
        + new UI_BIOS_KEYBOARD
        + new UI_MS_MOUSE
        + new UI_CURSOR;
    .
    .
    .
    // This automatically calls the cursor destructor.
    delete eventManager;
    delete display;
}
```

CHAPTER 3 – UI_DATE

Overview 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 28—UIW_DATE" of this manual for information about the date window object.) The public members of the UI_DATE class (declared in **UI_GEN.HPP**) are:

```
class UI_DATE
{
public:
    UI_DATE(void);
    UI_DATE(int packedDate);
    UI_DATE(int year, int month, int day);
    UI_DATE(const char *string, USHORT dtFlags = DTF_NO_FLAGS);
    virtual ~UI_DATE(void);

    void Export(void);
    void Export(int packedDate);
    void Export(int *year, int *month, int *day,
        int *dayOfWeek = NULL);
    void Export(char *string, int maxLength,
        USHORT dtFlags = DTF_NO_FLAGS);
    void Import(void);
    int Import(int packedDate);
    int Import(int year, int month, int day);
    int Import(const char *string,
        USHORT dtFlags = DTF_NO_FLAGS);
    static void NameTablesSet(DATE_TEXT *monthTablePtr = NULL,
        DATE_TEXT *dayTablePtr = NULL);

    int operator > (UI_DATE& rightOperand);
    int operator < (UI_DATE& rightOperand);
    int operator == (UI_DATE& rightOperand);
};
```

See also The example file **XDATE.CPP**, which gives a complete example of the UI_DATE class.

"Chapter 23—UI_TIME" of this manual, which describes a similar low-level class that maintains time information.

"Chapter 28—UIW_DATE" of this manual, which describes a high-level window object that uses the UI_DATE class to store display information.

UI_DATE::UI_DATE

Syntax

```
#include <ui_gen.hpp>

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

Remarks

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

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

The second 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 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).

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 and 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 all DOS based systems):

  DTF_ALPHA_MONTH—Forces the month value to interpreted in alphabetic form. For example, if the DTF_ALPHA_-MONTH flag were set, an ascii date such as "12-4-1900" would be invalid, since "12" is not an alphabetic month.

  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 class object. In this case, the ascii date will be interpreted using the default country information. 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 ascii date were specified by the programmer and the DTF_SYSTEM flag were set, then 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.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_DATE date1;              // System date initialization.
    UI_DATE date2(1990, 1, 1);  // Integer initialization.
    UI_DATE *date3 =            // String initialization.
        new UI_DATE("Jan. 1, 1990");
    .
    .
    .
}
```

UI_DATE::~UI_DATE

Syntax

#include <ui_gen.hpp>

virtual UI_DATE::~UI_DATE(void);

Remarks

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

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_DATE date1;              // System date initialization.
    UI_DATE date2(1990, 1, 1); // Integer initialization.
    UI_DATE *date3 =            // String initialization.
        new UI_DATE("Jan. 1, 1990");
    .
    .
    .
    delete date3;
    // The destructors for date1 and date2 are automatically called
    // when the scope of this function ends.
}
```

UI_DATE::Export

Syntax

#include <ui_gen.hpp>

void UI_DATE::Export(void);

or

void UI_DATE::Export(int **packedDate*);

or

void UI_DATE::Export(int **year*, int **month*, int **day*,
int **dayOfWeek* = 0);

or

void UI_DATE::Export(char **string*, int *maxLength*,
USHORT *dtFlags* = DTF_NO_FLAGS);

Remarks

The first overloaded function sets the system date according to the UI_DATE object's date information. This function only works on environments where the system date can be set.

The second overloaded function returns date information through a single packed integer argument.

- *packedDate*$_{in/out}$ is a packed representation of the date (whose format is the same as 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).

The third 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 fourth 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 at least *maxLength* in size.

- *maxLength*$_{in}$ is the maximum character length the ascii formatted date may occupy. For example, if *maxLength* were set to 15 and a long date of "December 4, 1980" were formatted, the final formatted date would be converted to "Dec. 4, 1980" since the original string was too long to fit in the specified buffer.

- *dtFlags*$_{in}$ gives formatting information about the return ascii date. The following flags (declared in **UI_GEN.HPP**) override the country dependant information (supplied by all DOS based systems):

  DTF_ALPHA_MONTH—Formats the month as an ascii string value. Some example dates with the DTF_ALPHA_MONTH

flag set are: "March 28, 1990," "December 4, 1980" and "January 3, 2003."

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

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

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

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

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

DTF_NO_FLAGS—Does not associate any special flags with the format function. In this case, the ascii date is formatted using the default country information. For example, the U.S. date "DEC 4 1989" would be shown as "4 DEC 1989" if default country information specified a European format, or "1989 DEC 4" if the default country information specified a Japanese format. This flag should not be used in conjunction with any other DTF flag.

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

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

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

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

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

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

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

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_DATE date; // Initialize a system date.

    // Print out the date in various forms.
    int packedDate;
    date.Export(&packedDate);
    printf("Packed date value: %x\n", packedDate);
```

```
    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, 128, 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>

void Import(void);
or
int Import(int *packedDate*);
or
int Import(int *year*, int *month*, int *day*);
or
int Import(const char **string*,
USHORT *dtFlags* = DTF_NO_FLAGS);

Remarks

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

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

- *returnValue*$_{out}$ is 0 (DTI_OK) if the packed date value was successfully imported. Otherwise, the return value is DTI_INVALID.
- *packedDate*$_{in}$ is a packed representation of the date (whose format is the same as 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).

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

- *returnValue*$_{out}$ is 0 (DTI_OK) if the date values were successfully imported. Otherwise, the return value is DTI_INVALID.

- *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 and 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 decide the order of date values.

- *returnValue*$_{out}$ is 0 (DTI_OK) if the ascii string parsed to a valid date. Otherwise, one of the following error codes (defined in **UI_GEN.HPP**) is returned:

  DTI_INVALID—The date was specified in an invalid format. For example, the date "Feb. 33, 1990" is invalid since there are not 33 days in February.

DTI_AMBIGUOUS—The month value was ambiguous. For example, the date "j 20 1990" has an ambiguous month since 'j' could apply to "January," "June," or "July."

DTI_VALUE_MISSING—A date value was not present in the ascii string and the DTF_SYSTEM flag has not been set.

DTI_INVALID_NAME—The ascii month or day-of-week name did not match any of the names contained in the day or month name tables (see the **UI_DATE::NameTablesSet** section below).

- *string*$_{in}$ is a pointer to the ascii date.
- *dtFlags*$_{in}$ gives information on how to interpret the date string. The following flags (declared in **UI_GEN.HPP**) override the country dependant information (supplied by all DOS based systems):

  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_FORCE_ALPHA_MONTH—Forces the date to contain the month value in alpha form. For example, if the DTF_ALPHA_MONTH flag were set, an ascii date such as "12-4-1900" would be invalid, since "12" is not an alphabetic month.

  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 class object. In this case, the ascii date will be interpreted using the default country information. 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 DT_SYSTEM flag were set, then 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.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_DATE date; // Initialize a system date.

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

    date.Import("1-1-1990");
    date.Export(asciiDate, 128, 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::NameTablesSet

Syntax

```
#include <ui_gen.hpp>

static void UI_DATE::NameTablesSet(
    const char **monthTable = NULL,
    const char **dayTable = NULL);
```

Remarks

This static function re-defines the month and day name tables used to give the ascii representation of a date. The default month and day tables are:

```
static char *_monthNames[] =
{
    "Jan.",     "January",
    "Feb.",     "February",
    "Mar.",     "March",
    "Apr.",     "April",
    "May.",     "May",
    "Jun.",     "June",
    "Jul.",     "July",
    "Aug.",     "August",
    "Sept.",    "September",
```

```
    "Oct.",     "October",
    "Nov.",     "November",
    "Dec.",     "December",
};

static char *_dayNames[] =
{
    "Sun.",     "Sunday",
    "Mon.",     "Monday",
    "Tues.",    "Tuesday",
    "Wed.",     "Wednesday",
    "Thurs.",   "Thursday",
    "Fri.",     "Friday",
    "Sat.",     "Saturday"
};
```

- *monthTable*$_{in}$ is a pointer to the new month table. This table must be allocated by the programmer and remain active throughout program execution. If this argument is NULL, *monthTable* is reset to point to the default month table (shown above).

- *dayTable*$_{in}$ is a pointer to the new day table. This table must be allocated by the programmer and remain active throughout program execution. If this argument is NULL, *dayTable* is reset to point to the default day table (shown above).

Example

```
#include <ui_gen.hpp>

static char *_spanishMonthNames[] =
{
    "Ene.",     "Enero",
    "Feb.",     "Febrero",
    "Mar.",     "Marzo",
    "Abr.",     "Abril",
    "May.",     "Mayo",
    "Jun.",     "Junio",
    "Jul.",     "Julio",
    "Ago.",     "Agosto",
    "Sept.",    "Septiembre",
    "Oct.",     "Octubre",
    "Nov.",     "Noviembre",
    "Dic.",     "Diciembre",
};

static char *_spanishDayNames[] =
{
    "Dom.",     "Domingo",
    "Lun.",     "Lunes",
    "Mar.",     "Martes",
    "Mier.",    "Miercoles",
    "Juev.",    "Jueves",
    "Vier.",    "Viernes",
    "Sab.",     "Sabado"
};
```

```
ExampleFunction1()
{
    // Redefine the date name tables.
    UI_DATE::NameTablesSet(_spanishMonthNames,
        _spanishDayNames);
    .
    .
    .
    // Restore the date name tables.
    UI_DATE::NameTablesSet(NULL, NULL);
}
```

UI_DATE::operator >

Syntax

#include <ui_gen.hpp>

int UI_DATE::operator > (UI_DATE& *rightOperand*);

Remarks

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

- *returnValue*$_{out}$ is TRUE if the UI_DATE object is chronologically greater than *rightOperand*. Otherwise, this value is FALSE.
- *rightOperand*$_{in}$ is the UI_DATE object against which to compare.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_DATE currentDate; // Initialize a system date.
    UI_DATE newYears1990("Jan. 1, 1990");
    UI_DATE twentyFirstCentury("Jan. 1, 2001");

    // Check the dates.
    if (currentDate == newYears1990)
        printf("It's new years day 1990.\n")
    else if (currentDate > twentyFirstCentury ||
        currentDate == twentyFirstCentury)
        printf("It's the twenty first century.\n");
    else
        printf("It's not the twenty first century.\n");
}
```

UI_DATE::operator <

Syntax

#include <ui_gen.hpp>

int UI_DATE::operator < (UI_DATE& *rightOperand*);

Remarks

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

- *returnValue*$_{out}$ is TRUE if the UI_DATE object is chronologically less than *rightOperand*. Otherwise, this value is FALSE.
- *rightOperand*$_{in}$ is the UI_DATE object against which to compare.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_DATE currentDate; // Initialize a system date.
    UI_DATE newYears1990("Jan. 1, 1990");
    UI_DATE twentyFirstCentury("Jan. 1, 2001");

    // Check the dates.
    if (currentDate == newYears1990)
        printf("It's new years day 1990.\n")
    else if (currentDate < twentyFirstCentury)
        printf("It's not the twenty first century.\n");
    else
        printf("It's the twenty first century.\n");
}
```

UI_DATE::operator ==

Syntax

#include <ui_gen.hpp>

int UI_DATE::operator == (UI_DATE& *rightOperand*);

Remarks

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

- *returnValue*$_{out}$ is TRUE if the UI_DATE object is chronologically equal to *rightOperand*. Otherwise, this value is FALSE.
- *rightOperand*$_{in}$ is the UI_DATE object against which to compare.

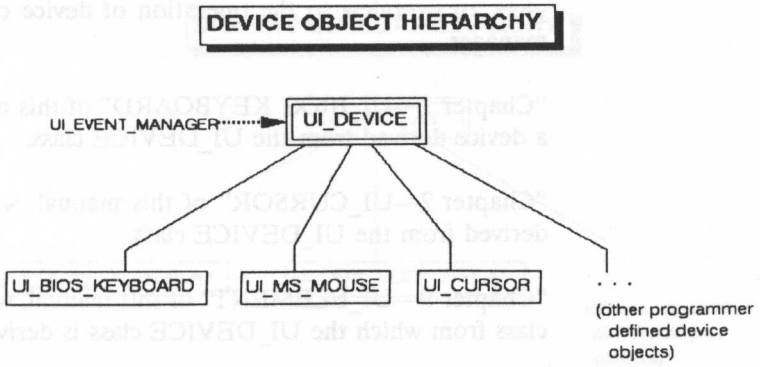
Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_DATE currentDate; // Initialize a system date.
    UI_DATE newYears1990("Jan. 1, 1990");
    UI_DATE twentyFirstCentury("Jan. 1, 2001");

    // Check the dates.
    if (currentDate == newYears1990)
        printf("It's new years day 1990.\n")
    else if (currentDate < twentyFirstCentury)
        printf("It's not the twenty first century.\n");
    else
        printf("It's the twenty first century.\n");
}
```

CHAPTER 4 – UI_DEVICE

Overview

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 UI_BIOS_KEYBOARD, UI_CURSOR, or UI_MOUSE, must be used. The figure below shows the device object hierarchy:



Classes derived from the UI_DEVICE base class include:

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

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

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

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

Input devices are attached to the event manager at run-time by the programmer. Once a device is attached, it feeds input information to

the event queue when polled by the event manager, or it feeds directly to the event queue if it is an interrupt device.

Other chapters in this manual contain more information about the classes derived from the UI_DEVICE base class as well as their construction, destruction and use within the event manager (see the references below).

See also

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the operation of device classes within the event manager.

"Chapter 1—UI_BIOS_KEYBOARD" of this manual, which describes a device derived from the UI_DEVICE class.

"Chapter 2—UI_CURSOR" of this manual, which describes a device derived from the UI_DEVICE class.

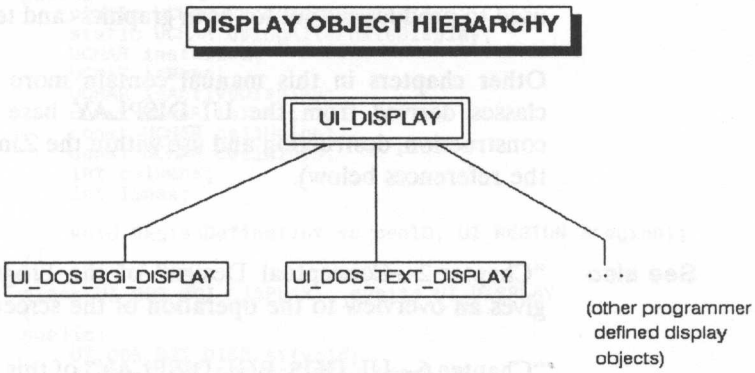
"Chapter 9—UI_ELEMENT" of this manual, which describes the base class from which the UI_DEVICE class is derived.

"Chapter 13—UI_EVENT_MANAGER" of this manual, which describes the operation (e.g., addition, subtraction, state change) of device classes within the event manager.

"Chapter 18—UI_MS_MOUSE" of this manual, which describes a device derived from the UI_DEVICE class.

CHAPTER 5 – UI_DISPLAY

Overview The UI_DISPLAY class is an abstract class that defines basic information associated with the screen display. Since the UI_DISPLAY class is abstract, it cannot be used as a constructed class. Rather, derived classes, such as UI_DOS_TEXT_DISPLAY or UI_DOS_BGI_-DISPLAY must be used. The graphic image below shows the display object hierarchy:



Classes derived from the UI_DISPLAY base class include:

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

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

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

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

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

The 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.

Other chapters in this manual contain more information about the classes derived from the UI_DISPLAY base class as well as their construction, destruction and use within the Zinc Interface Library (see the references below).

See also "Chapter 2—Conceptual Design" of the Programmer's Guide which gives an overview to the operation of the screen display.

"Chapter 6—UI_DOS_BGI_DISPLAY" of this manual which describes a graphics display derived from the UI_DISPLAY class.

"Chapter 7—UI_DOS_TEXT_DISPLAY" of this manual which describes a text display derived from the UI_DISPLAY class.

CHAPTER 6 – UI_DOS_BGI_DISPLAY

Overview The UI_DOS_BGI_DISPLAY class implements a graphics display that uses the Turbo C++ BGI graphics library package to display all information to the screen. The public members of the UI_DISPLAY and UI_DOS_BGI_DISPLAY classes (declared in **UI_DSP.HPP**) are:

```
class UI_DISPLAY
{
public:
    static UCHAR usingActiveDisplay;
    static UCHAR usingAlternateDisplay;
    UCHAR installed;
    UCHAR isMono;
    UCHAR isActiveDisplay;
    const UCHAR isText;
    const UCHAR cellHeight;
    const UCHAR cellWidth;
    int columns;
    int lines;

    void RegionDefine(int screenID, UI_REGION &region);
};

class UI_DOS_BGI_DISPLAY : public UI_DISPLAY
{
public:
    UI_DOS_BGI_DISPLAY(void);
    virtual ~UI_DOS_BGI_DISPLAY(void);

    void Bitmap(int screenID, const UI_REGION &region,
        const USHORT *bitmap, const UI_PALETTE *palette,
        int fillBackground = TRUE);
    void Fill(int screenID, const UI_REGION &region,
        const UI_PALETTE *palette);
    void FillXOR(const UI_REGION &region);
    void Line(int screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1);
    void Rectangle(int screenID, const UI_REGION &region,
        const UI_PALETTE *palette, int width = 1);
    void RectangleXOR(const UI_REGION &region);
    void RegionConvert(UI_REGION &region);
    void Text(int screenID, int left, int top,
        const char *text, const UI_PALETTE *palette,
        int length = -1, int fillBackground = TRUE);
    int TextHeight(const char *string);
    int TextWidth(const char *string);
};
```

- *cellHeight* is the pixel height of a cell. The cell height is determined by the default font setting.
- *cellWidth* is the pixel width of a cell. The cell width is determined by the default font setting.

- *columns* gives the total pixel width of the screen display. For example, if the constructed display were 480 lines x 640 columns, the value of *columns* would be 640.
- *installed* is TRUE if the display is correctly installed. Otherwise, this value is FALSE.
- *isActiveDisplay* is TRUE if the specified display class object is the active display. If it is the alternate display, this value is FALSE.
- *isMono* is not used by the graphics display class.
- *isText* is always FALSE for UI_DOS_BGI_DISPLAY class objects.
- *lines* gives the total pixel height of the screen display. For example, if the constructed display were 480 lines x 640 columns, the value of *lines* would be 480.
- *usingActiveDisplay* is TRUE if the Zinc Interface Library has constructed an active display. The first display created in an application is the active (primary) display.
- *usingAlternateDisplay* is TRUE if the Zinc Interface Library has constructed an alternate display. The second display created in an application is the alternate (secondary) display.

See also The example file **XBGIDISP.CPP**, which gives a complete example of the UI_DOS_BGI_DISPLAY class.

"Chapter 2—Conceptual Design" of the Programmer's Guide which gives an overview to the operation of the screen display.

"Chapter 5—UI_DISPLAY" of this manual which describes the base class from which the UI_DOS_BGI_DISPLAY class is derived.

"Chapter 7—UI_DOS_TEXT_DISPLAY" of this manual which describes a text display derived from the UI_DISPLAY class.

UI_DOS_BGI_DISPLAY::UI_DOS_BGI_DISPLAY

Syntax #include <ui_dsp.hpp>

UI_DOS_BGI_DISPLAY::UI_DOS_BGI_DISPLAY(void);

Remarks This constructor returns a pointer to a new UI_DOS_BGI_DISPLAY object. When a new UI_DOS_BGI_DISPLAY class is constructed, the system finds the associated BGI device driver, then clears the screen display to the background color and pattern specified by the global palette variable *_backgroundPalette* (See "Chapter 19—UI_PALETTE" of this manual).

If a primary display class has already been constructed, this function constructs a secondary graphics display using an alternate display (if a second display exists and it is the graphics display). This provides the low-level implementation of dual-monitor support.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    // Call the BGI display constructor.
    UI_DOS_BGI_DISPLAY display;
    .
    .
    .
}

ExampleFunction2(int graphics)
{
    // Put a level of abstraction into the call.
    UI_DISPLAY *display;
    display = (graphics) ?
        (UI_DISPLAY *)new UI_DOS_BGI_DISPLAY :
        (UI_DISPLAY *)new UI_DOS_TEXT_DISPLAY(LINES_AUTO);
    .
    .
    .
}

ExampleFunction3()
{
    // Create two displays (dual screen monitor support).
    UI_DOS_BGI_DISPLAY display1;
    UI_DOS_TEXT_DISPLAY *display2 = new UI_DOS_TEXT_DISPLAY;
    .
    .
    .
}
```

UI_DOS_BGI_DISPLAY::~UI_DOS_BGI_DISPLAY

Syntax #include <ui_dsp.hpp>

virtual UI_DOS_BGI_DISPLAY::~UI_DOS_BGI_DISPLAY(void);

Remarks This virtual destructor destroys the class information associated with the UI_DOS_BGI_DISPLAY object and clears the screen display.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_BGI_DISPLAY display;
    .
    .
    .
    // The display destructor is called automatically when
    // the scope of this function ends.
}

ExampleFunction2(int graphics)
{
    UI_DISPLAY *display;
    display = (graphics) ?
        (UI_DISPLAY *)new UI_DOS_BGI_DISPLAY :
        (UI_DISPLAY *)new UI_DOS_TEXT_DISPLAY(LINES_AUTO);
    .
    .
    .
    // Call the display destructor.
    delete display;
}

ExampleFunction3()
{
    UI_DOS_BGI_DISPLAY display1;
    UI_DOS_TEXT_DISPLAY *display2 = new UI_DOS_TEXT_DISPLAY;
    .
    .
    .
    // Delete display2 since it is a UI_DOS_BGI_DISPLAY class
    // pointer.
    delete display2;
    // Display1 is destroyed when the scope of this function
    // ends.
}
```

UI_DOS_BGI_DISPLAY::Bitmap

Syntax #include <ui_dsp.hpp>

void UI_DOS_BGI_DISPLAY::Bitmap(int *screenID*,
const UI_REGION *&region*, const USHORT **bitmap*,
const UI_PALETTE **palette*, int *fillBackground*);

Remarks This function draws a bitmap image to the screen at the coordinates specified by *region*. A bitmap is defined by an array of USHORT (unsigned short) values. Each bitmap must be constructed in the following manner:

1—The first USHORT value must contain the width of the bitmap in pixels.

2—The second USHORT value must contain the height of the bitmap in pixels.

3—all remaining USHORT values contain the actual bitmap pattern. Each of these values is evaluated from high to low bit.

For example, a 16 x 4 bitmap (16 columns, 4 lines) that draws a rectangle could be represented in the following manner:

```
// This is the bitmap for a rectangle.
USHORT rectangleBitmap[] =
{
    16,          // Width of the bitmap pattern.
    4,           // Height of the bitmap pattern.
    0xFFFF,      // ................
    0x8001,      // .              .
    0x8001,      // .              .
    0xFFFF       // ................
};
```

There are no restrictions on the height and width of a bitmap image. The width however, is aligned along 16 bit boundaries. For example, if a 4 x 4 bitmap were defined and contained a similar pattern to the bitmap shown above, the following declaration could be made:

```
// This is the bitmap for a box.
USHORT boxBitmap[] =
{
    4,           // Width of the bitmap pattern.
    4,           // Height of the bitmap pattern.
```

```
    0xF000,     // ••••     Only the top four bits
    0x9000,     // •  •     of these values are
    0x9000,     // •  •     evaluated.
    0xF000      // ••••
};
```

Similarly, a 32 x 4 rectangle bitmap would be represented in the following manner:

```
// This is the bitmap for 32 pixel wide rectangle.
USHORT rectangleBitmap[] =
{
    32,                 // Width of the bitmap pattern.
    4,                  // Height of the bitmap pattern.
                        // Each line is represented by
                        // 2 USHORTS.
    0xFFFF, 0xFFFF,     // ••••••••••••••••••••••••••••••••
    0x8000, 0x0001,     // •                              •
    0x8000, 0x0001,     // •                              •
    0xFFFF, 0xFFFF      // ••••••••••••••••••••••••••••••••
};
```

The highlight bits (i.e., those bits whose values are 1) of the bitmap are drawn with the foreground color of the palette argument. The non-highlight bits (i.e., those bits whose values are 0) of the bitmap are drawn with the background color of the palette argument if *fillBackground* is set to TRUE. Otherwise, non-highlight bits are ignored.

The constants BITMAP_HEIGHT and BITMAP_WIDTH are declared to give array access to the height and width values of a specified bitmap.

Bitmaps do not have text screen equivalents. There is a **UI_DOS_TEXT_DISPLAY::Bitmap** function specified for the UI_DOS_TEXT_DISPLAY class, but it is a stub. Thus the **UI_DOS_BGI_DISPLAY::Bitmap** function should be used with caution.

- *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. (See "Chapter 5—UI_DISPLAY" of this manual for more information about screen identifications.)
- *region*$_{in}$ is a reference pointer to the desired display region. *Region.left* and *region.top* are pixel coordinate values that specify the left-top corner position of the bitmap. *Region.right* and *region.bottom* are pixel coordinate values that specify the right-bottom corner position of the bitmap. Typically, the right and

bottom values will be the left-top values plus the bitmap's respective width and height.

- *bitmap*$_{in}$ is the bitmap pattern to be displayed. The bitmap pattern must follow the format discussed above.

- *palette*$_{in}$ is a pointer to the palette used when displaying the bitmap pattern. The palette's foreground color is used for all 1 specified bits. The palette's background color is used for all 0 specified bits if *fillBackground* is set to TRUE.

- *fillBackground*$_{in}$ is a flag indicating whether to fill the background (i.e., all 0 bits) with the defined palette's background color. If *fillBackground* is set to TRUE, the background will be filled with the palette's background color. Otherwise, 0 bits are ignored.

Example

```
#include <graphics.h>   // include for graphics colors
#include <ui_dsp.hpp>

USHORT rectangleBitmap[] =
{
    16,                                 // Width
    4,                                  // Height
    0xFFFF, 0x8001, 0x8001, 0xFFFF  // Bitmap pattern
};
UI_PALETTE rectanglePalette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(BLUE, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};

ExampleFunction1()
{
    // Draw a blue on white box to the screen.
    UI_DOS_BGI_DISPLAY display;

    // Draw a box to the left-top corner of the screen.
    region.left = region.top = 0;
    region.right = region.left + bitmap[BITMAP_WIDTH] - 1;
    region.bottom = region.top + bitmap[BITMAP_HEIGHT] - 1;
    display.Bitmap(ID_SCREEN, region, rectangleBitmap,
        rectanglePalette, TRUE};
    .
    .
    .
}

USHORT boxBitmap[] =
{
    8,          // Width
    8,          // Height
    0xFF00,     // ........
    0x8100,     // .      .
    0x8100,     // .      .
    0x8100,     // .      .
```

```
    0x8100,     // •      •
    0x8100,     // •      •
    0x8100,     // •      •
    0xFF00      // ••••••••
};
UI_PALETTE boxPalette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(BLUE, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};
USHORT xBitmap[] =
{
    8,          // Width
    8,          // Height
    0x0000,     //
    0x4200,     //  •    •
    0x2400,     //   •  •
    0x1800,     //    ••
    0x1800,     //    ••
    0x2400,     //   •  •
    0x4200,     //  •    •
    0xFF00      //
};
UI_PALETTE xPalette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(BLUE, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};

ExampleFunction2()
{
    // Construct the graphics display then draw a black box with
    // a red 'X' to the screen.  The first Bitmap() call fills
    // the background and draws the box.  The second Bitmap() call
    // draws the 'X'.
    UI_DOS_BGI_DISPLAY display;

    // Define a unique region for the bitmaps.
    display.RegionDefine(-1, region);
    region.left = region.top = 0;
    region.right = region.left + boxBitmap[BITMAP_WIDTH] - 1;
    region.bottom = region.top + boxBitmap[BITMAP_HEIGHT] - 1;

    display.Bitmap(-1, region, boxBitmap, boxPalette, TRUE};
    display.Bitmap(-1, region, xBitmap, xPalette, FALSE};
}
```

UI_DOS_BGI_DISPLAY::Fill

Syntax #include <ui_dsp.hpp>

void UI_DOS_BGI_DISPLAY::Fill(int *screenID*,
const UI_REGION *&region*, const UI_PALETTE **palette*);

Remarks This function fills a rectangular, two dimensional region on the screen with the background color and pattern specified by *palette*. This function uses the Turbo C++ **bar()** function with the following modifications:

1—The screen display for all devices (e.g., the mouse and cursor) are shut off before the specified region is filled. This prevents devices from overwriting the screen display.

2—The region is clipped according to the screen identification. This prevents writing to overlapping window regions.

3—The fill color is specified by the background portion of *palette*.

- *screenID*$_{in}$ is a screen object identification used to determine the parts of the fill region that can be updated to the screen. Only those screen locations that match *screenID* are updated. (See "Chapter 5—UI_DISPLAY" of this manual for more information about screen identifications.)
- *region*$_{in}$ is a reference pointer to the desired fill region. *Region.left* and *region.top* are pixel coordinate values that specify the left-top corner position of the fill region. *Region.right* and *region.bottom* values specify the right-bottom corner position of the fill region.
- *palette*$_{in}$ is a pointer to the palette argument used when filling the screen region. The palette's background color is used for the fill.

Example

```
#include <graphics.h>   // include for graphic colors.
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_BGI_DISPLAY display;
    .
    .
    .
    // Define the fill region.
    UI_REGION region = { 100, 100, 200, 200 };
    display.RegionDefine(ID_SCREEN, region);
    display.Fill(ID_SCREEN, region, _backgroundPalette);
    .
    .
    .
}
```

```
UI_PALETTE palette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(BLUE, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};

ExampleFunction2()
{
    UI_DOS_BGI_DISPLAY display;
    .
    .
    .
    // Fill the specified region.  Notice the region is defined to
    // be the whole screen, but only the region with the
    // identification -1 will actually be updated.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display.columns / 4;
    region.bottom = display.lines / 4;
    display.RegionDefine(-1, region);
    region.right = display.columns - 1;
    region.bottom = display.lines - 1;
    display.Fill(-1, region, palette);
}

ExampleFunction3()
{
    UI_DOS_BGI_DISPLAY *display = new UI_DOS_BGI_DISPLAY;
    .
    .
    .
    // Fill the whole screen with the background palette.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display->columns - 1;
    region.bottom = display->lines - 1;
    // Make sure the full screen is defined.
    display->RegionDefine(ID_SCREEN, region);
    display->Fill(ID_SCREEN, region, _backgroundPalette);
    .
    .
    .
}
```

UI_DOS_BGI_DISPLAY::FillXOR

Syntax #include <ui_dsp.hpp>

void UI_DOS_BGI_DISPLAY::FillXOR(
    const UI_REGION *&region*);

Remarks This function XOR fills a rectangular, two dimensional region of the screen. This function differs from the **UI_DOS_BGI_DISPLAY::Fill** function in the following respects:

1—No screen identification is required. The **UI_DOS_BGI_-DISPLAY::FillXOR** function automatically XOR fills the region, independent of the screen identification for the region.

2—No palette is required. The XOR region is the opposite color of the current screen color on the specified pixel locations. To remove an XOR region, simply XOR the same region again.

This function differs from the **UI_DOS_BGI_DISPLAY::RectangleXOR** function (described below), because it XOR fills the whole region, not just the edge of the region.

- *region*$_{in}$ is a reference pointer to the desired XOR region. *Region.left* and *region.top* are pixel coordinate values that specify the left-top corner position of the XOR region. *Region.right* and *region.bottom* are pixel coordinate values that specify the right-bottom corner position of the XOR region.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_BGI_DISPLAY display;
    .
    .
    .
    // Fill an XOR region on the screen.
    UI_REGION region = { 100, 100, 150, 150 };
    display.FillXOR(region);
    .
    .
    .
    // Remove the XOR region.
    display.FillXOR(region);
    .
    .
    .
}
```

UI_DOS_BGI_DISPLAY::Line

Syntax #include <ui_dsp.hpp>

void UI_DOS_BGI_DISPLAY::Line(int *screenID*, int *left*, int *top*, int *right*, int *bottom*, const UI_PALETTE **palette*, int *width* = 1);

Remarks This function draws a graphics line between two points specified by top-left and bottom-right in the color specified by the background portion of the palette argument. This function uses the Turbo C++ **line()** function with the following modifications:

1—The screen display for all devices (e.g., the mouse and cursor) are shut off before the specified line is drawn. This prevents devices from overwriting the screen display.

2—The region is clipped according to the screen identification. This prevents writing to overlapping window regions.

3—The line color is specified by the background portion of *palette*.

- $screenID_{in}$ 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. (See "Chapter 5—UI_DISPLAY" of this manual for more information about screen identifications.)

- $left_{in}$ is the beginning left position of the line on the screen (in pixel coordinates). This argument, combined with *top*, forms the beginning point of the line.

- top_{in} is the beginning top position of the line on the screen (in pixel coordinates). This argument, combined with *left*, forms the beginning point of the line.

- $right_{in}$ is the ending right position of the line on the screen (in pixel coordinates). This argument, combined with *bottom*, forms the ending point of the line.

- *bottom*$_{in}$ is the ending bottom position of the line on the screen (in pixel coordinates). This argument, along with *right*, forms the ending point of the line.

- *palette*$_{in}$ is a pointer to a palette used when drawing the graphics line. The palette's background color is used to draw the line.

- *width*$_{in}$ is ignored by the graphics display. In text mode, this value specifies the line width.

Example

```
#include <graphics.h>  // include for graphic colors.
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_BGI_DISPLAY display;
    .
    .
    .
    // Draw a diagonal line on the background display.
    display.Line(-1, 100, 100, 200, 200, _backgroundPalette);
    .
    .
    .
}
UI_PALETTE palette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(RED, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};

ExampleFunction2()
{
    UI_DOS_BGI_DISPLAY display;
    .
    .
    .
    // Draw a line to a pre-defined region of the screen.
    // Notice the coordinates are defined to draw a line on the
    // whole screen, but only the region with the identification
    // -1 will actually be updated.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display.columns / 4;
    region.bottom = display.lines / 4;
    display.RegionDefine(-1, region);
    display.Line(-1, 0, 0, display.columns - 1,
        display.lines - 1, palette);
}
ExampleFunction3()
{
    UI_DOS_BGI_DISPLAY *display = new UI_DOS_BGI_DISPLAY;
    .
    .
    .
    // Draw a diagonal line across the screen.
    UI_REGION region;
```

```
    region.left = region.top = 0;
    region.right = display->columns - 1;
    region.bottom = display->lines - 1;
    // Make sure the full screen is defined.
    display->RegionDefine(ID_SCREEN, region);
    display->Fill(ID_SCREEN, 0, 0, display->columns - 1,
        display->lines - 1, _backgroundPalette);
    .
    .
    .
}
```

UI_DOS_BGI_DISPLAY::Rectangle

Syntax #include <ui_dsp.hpp>

void UI_DOS_BGI_DISPLAY::Rectangle(int *screenID*, const UI_REGION *&region*, const UI_PALETTE **palette*, int *width* = 1);

Remarks This function draws a rectangle on the screen. The background color of the color palette is the color used when drawing the rectangle. This function uses the Turbo C++ **rectangle()** function with the following additions:

1—The screen display for all devices (e.g., the mouse and cursor) are shut off before the specified rectangle is drawn. This prevents devices from overwriting the screen display.

2—The rectangle is clipped according to the screen identification. This prevents writing to overlapping window regions.

3—The rectangle color is specified by the background portion of *palette*.

- *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.
- *region*$_{in}$ is a reference pointer to the desired rectangle region. *Region.left* and *region.top* are pixel coordinate values that specify the left-top corner position of the rectangle. *Region.right* and *region.bottom* are pixel coordinate values that specify the right-bottom corner position of the rectangle.

- *palette*$_{in}$ is a pointer to the palette used when drawing the rectangle. The palette's background color is used to draw the rectangle.

- *width*$_{in}$ is ignored by the graphics display. In text mode, this value specifies the rectangle width.

Example

```
#include <graphics.h>   // include file for the graphics colors.
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_BGI_DISPLAY display;
    .
    .
    .
    // Draw a rectangle to the screen's background.
    UI_REGION region = { 100, 100, 200, 200 };
    display.RegionDefine(ID_SCREEN, region);
    display.Rectangle(ID_SCREEN, region, _backgroundPalette);
    .
    .
    .
}

UI_PALETTE palette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(RED, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};

ExampleFunction2()
{
    UI_DOS_BGI_DISPLAY *display = new UI_DOS_BGI_DISPLAY;
    .
    .
    .
    // Draw a rectangle.  Notice the region is defined to be the
    // whole screen, but only the region with the identification -1
    // will actually be updated.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display->columns / 4;
    region.bottom = display->lines / 4;
    display->RegionDefine(-1, region);

    region.right = display->columns - 1;
    region.bottom = display->lines - 1;
    display->Rectangle(-1, region, palette);
    .
    .
    .
}
```

```
ExampleFunction3()
{
    UI_DOS_BGI_DISPLAY *display = new UI_DOS_BGI_DISPLAY;
    .
    .
    .
    // Draw a rectangle around the whole screen with the
    // background palette.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display->columns - 1;
    region.bottom = display->lines - 1;
    // Make sure the full screen is defined.
    display->RegionDefine(ID_SCREEN, region);
    display->Rectangle(ID_SCREEN, region, _backgroundPalette);
    .
    .
    .
}
```

UI_DOS_BGI_DISPLAY::RectangleXOR

Syntax #include <ui_dsp.hpp>

void UI_DOS_BGI_DISPLAY::RectangleXOR(
const UI_REGION *&region*);

Remarks This function draws an XOR rectangle on the screen. This function differs from the **UI_DOS_BGI_DISPLAY::Rectangle** function in the following respects:

1—No screen identification is required. The **UI_DOS_BGI_-DISPLAY::RectangleXOR** function automatically XORs the rectangle, independent of the screen identification for the rectangle.

2—No palette is required. The XOR rectangle is the opposite color of the current screen color on the specified pixel locations. To remove an XOR rectangle, simply XOR the same rectangle again.

This function differs from the **UI_DOS_BGI_DISPLAY::FillXOR** function (described above), because it does not fill the region, but rather, draws a box at the edge of the region.

- *region*$_{in}$ is a reference pointer to the desired XOR rectangle. *Region.left* and *region.top* are pixel coordinate values that specify the left-top corner position of the XOR rectangle. *Region.right* and

region.bottom are pixel coordinate values that specify the right-bottom corner position of the XOR rectangle.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_BGI_DISPLAY display;
    .
    .
    .
    // Draw an XOR rectangle on the screen.
    UI_REGION region = { 100, 100, 150, 150 };
    display.RectangleXOR(region);
    .
    .
    .
    // Remove the XOR rectangle.
    display.RectangleXOR(region);
    .
    .
    .
}
```

UI_DOS_BGI_DISPLAY::RegionConvert

Syntax

#include <ui_dsp.hpp>

void UI_DOS_BGI_DISPLAY::RegionConvert(
    UI_REGION *&region*);

Remarks

This function converts a cell (or text) region to a graphics (or pixel) region. The purpose of this function is to allow portability of higher-level display code. This function does not check for regions that have already been converted from text to graphics coordinates.

- $region_{in}$ is a reference pointer to the region whose cell coordinates are to be converted.

Example

```
#include <graphics.h>  // include for the palette colors.
#include <ui_dsp.hpp>

UI_PALETTE palette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(RED, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};
```

```
ExampleFunction1()
{
    UI_DOS_BGI_DISPLAY display;
    .
    .
    .
    // Convert cell coordinates to draw a box.
    // (Display independent)
    UI_REGION region = { 0, 0, 10, 10 };    // Cell coordinates.
    display.RegionConvert(region);

    display.Fill(ID_SCREEN, region, palette);
}

ExampleFunction2()
{
    UI_DOS_BGI_DISPLAY *display =
        new UI_DOS_BGI_DISPLAY;
    .
    .
    .
    // Convert cell coordinates to draw a box.
    // (Display independent)
    UI_REGION *region = { 0, 0, 10, 10 };   // Cell coordinates.
    display->RegionConvert(region);
    display->RegionDefine(-1, region);
    display->Rectangle(-1, *region, palette);
}
```

UI_DOS_BGI_DISPLAY::Text

Syntax #include <ui_dsp.hpp>

void UI_DOS_BGI_DISPLAY::Text(int *screenID*, int *left*, int *top*, const char **text*, const UI_PALETTE **palette*, int *length* = -1, int *fillBackground* = TRUE);

Remarks This function draws a text string to the screen. This function uses the Turbo C++ **outtextxy()** function with the following additions:

1—The screen display for all devices (e.g., the mouse and cursor) are shut off before the specified string is drawn. This prevents devices from overwriting the screen display.

2—The string is clipped according to the screen identification. This prevents writing to overlapping window regions.

3—The string color is specified by the background portion of *palette*.

- *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. (See "Chapter 5—UI_DISPLAY" of this manual for more information about screen identifications).

- *left*$_{in}$ is the beginning left position of the text on the screen (in pixel coordinates). This argument, combined with *top*, forms the beginning point where the text is to be displayed.

- *top*$_{in}$ is the beginning top position of the text on the screen (in pixel coordinates). This argument, combined with *left*, forms the beginning point where the text is to be displayed.

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

- *palette*$_{in}$ is a pointer to the palette used when drawing the text. The palette's foreground color is used to draw the characters of the text. If *fillBackground* is set to TRUE, the palette's background color is used to fill the cell region behind the character.

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

- *fillBackground*$_{in}$ is a flag indicating whether to fill the background (i.e., the cell region underneath the text characters) with the defined palette's background color. If *fillBackground* is set to TRUE, the text's background will be filled with the palette's background color. Otherwise, only the foreground color is used to draw the text characters.

Example

```
#include <graphics.h>   // include file for the graphics colors.
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_BGI_DISPLAY display;
    .
    .
    .
    // Display text to the top-left corner of the screen.
    UI_REGION region = { 100, 100, 200, 200 };
    display.RegionDefine(ID_SCREEN, region);
    display.Text(ID_SCREEN, 0, 0, "This is sample text",
        _backgroundPalette);
    .
    .
    .
}

UI_PALETTE palette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(BLUE, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};

ExampleFunction2()
{
    UI_DOS_BGI_DISPLAY *display = new UI_DOS_BGI_DISPLAY;
    .
    .
    .
    // Display the text.  Notice the text fills a large portion
    // of the whole screen, but only the region with the
    // identification -1 will actually be updated.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display->columns / 4;
    region.bottom = display->lines / 4;
    display->RegionDefine(-1, region);
    display->Text(-1, 0, 0,
        "This is sample text that doesn't all fit in the region",
        palette);
    .
    .
    .
}
```

UI_DOS_BGI_DISPLAY::TextHeight

Syntax #include <ui_dsp.hpp>

int UI_DOS_BGI_DISPLAY::TextHeight(const char *_string_);

Remarks This function returns the height of a specified string in screen pixels. This function is equivalent to the Turbo C++ **textheight**() function.

- *returnValue*$_{out}$ is the height of the string in screen pixels.
- *string*$_{in}$ is a pointer to the string whose height is to be determined.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_BGI_DISPLAY display;

    // Get the display independent size of text.
    int height = display.TextHeight("This is sample text");
    .
    .
    .
}
```

UI_DOS_BGI_DISPLAY::TextWidth

Syntax

#include <ui_dsp.hpp>

int UI_DOS_BGI_DISPLAY::TextWidth(const char **string*);

Remarks

This function returns the width of a specified string in pixel size. This function is equivalent to the Turbo C++ **textwidth**() function.

- *returnValue*$_{out}$ is the width of the string in screen pixels.
- *string*$_{in}$ is a pointer to the string whose width is to be determined.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_BGI_DISPLAY display;

    // Get the display independent size of text.
    int width = display.TextWidth("This is sample text");
    .
    .
    .
}
```

CHAPTER 7 – UI_DOS_TEXT_DISPLAY

Overview The UI_DOS_TEXT_DISPLAY class implements a text display that writes to screen memory. The public members of the UI_DISPLAY and UI_DOS_TEXT_DISPLAY classes (declared in **UI_DSP.HPP**) are:

```
class UI_DISPLAY
{
public:
    static UCHAR usingActiveDisplay;
    static UCHAR usingAlternateDisplay;
    UCHAR installed;
    UCHAR isMono;
    UCHAR isActiveDisplay;
    const UCHAR isText;
    const UCHAR cellHeight;
    const UCHAR cellWidth;
    int columns;
    int lines;

    void RegionDefine(int screenID, UI_REGION &region);
};

class UI_DOS_TEXT_DISPLAY : public UI_DISPLAY
{
public:
    UI_DOS_TEXT_DISPLAY(LINE_MODE lineMode = LINES_AUTO);
    virtual ~UI_DOS_TEXT_DISPLAY(void);

    void Bitmap(int screenID, const UI_REGION &region,
        const USHORT *bitmap, const UI_PALETTE *palette,
        int fillBackground = TRUE);
    void Fill(int screenID, const UI_REGION &region,
        const UI_PALETTE *palette);
    void FillXOR(const UI_REGION &region);
    void Line(int screenID, int left, int top, int right,
        int bottom, const UI_PALETTE *palette, int width = 1);
    void Rectangle(int screenID, const UI_REGION &region,
        const UI_PALETTE *palette, int width = 1);
    void RectangleXOR(const UI_REGION &region);
    void RectangleXORDiff(const UI_REGION &oldRegion,
        const UI_REGION &newRegion);
    void RegionConvert(UI_REGION &region);
    void Text(int screenID, int left, int top,
        const char *text, const UI_PALETTE *palette,
        int length = -1, int fillBackground = TRUE);
    int TextHeight(const char *string);
    int TextWidth(const char *string);
};
```

- *cellHeight* is always 1 for a UI_DOS_TEXT_DISPLAY class object.
- *cellWidth* is always 1 for a UI_DOS_TEXT_DISPLAY class object.
- *columns* gives the total character width of the screen display. For example, if the constructed display were 25 lines x 80 columns, the value of *columns* would be 80.

- *installed* is TRUE if the display installed correctly. Otherwise, this value is FALSE.

- *isActiveDisplay* is TRUE if the specified display class object is the active display. If it is the alternate display, this value is FALSE.

- *isMono* is TRUE if the monitor is in monochrome (i.e., black and white) mode. Otherwise, this value is FALSE (i.e., the monitor is in color mode). This variable is only used when the application is running in text mode.

- *isText* is always TRUE for UI_DOS_TEXT_DISPLAY class objects.

- *lines* gives the total character height of the screen display. For example, if the constructed display were 25 lines x 80 columns, the value of *lines* would be 25.

- *usingActiveDisplay* is TRUE if the Zinc Interface library has constructed an active display. The first display created in an application is the active (i.e., primary) display.

- *usingAlternateDisplay* is TRUE if the Zinc Interface library has constructed an alternate display. The second display created in an application is the alternate (i.e., secondary) display.

See also The example file **XTXTDISP.CPP**, which gives a complete example of the UI_DOS_TEXT_DISPLAY class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the operation of the screen display.

"Chapter 5—UI_DISPLAY" of this manual which describes the base class from which the UI_DOS_TEXT_DISPLAY class is derived.

"Chapter 6—UI_DOS_BGI_DISPLAY" of this manual which describes a graphics display derived from the UI_DISPLAY class.

UI_DOS_TEXT_DISPLAY::UI_DOS_TEXT_DISPLAY

Syntax

```
#include <ui_dsp.hpp>

UI_DOS_TEXT_DISPLAY::UI_DOS_TEXT_DISPLAY(
    TEXT_DISPLAY_MODE displayMode = TDM_AUTO);
```

Remarks

This constructor returns a pointer to a new UI_DOS_TEXT_DISPLAY object. When a new UI_DOS_TEXT_DISPLAY class is constructed, the system clears the screen to the background color and pattern specified by the global palette variable *_backgroundPalette* (See "Chapter 20—UI_PALETTE" of this manual).

If a primary display class has already been constructed, this function constructs a secondary text display using an alternate display (if any). This provides the low-level implementation of dual monitor support.

- $displayMode_{in}$ 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—Splits the screen display into 25 line by 40 column character cells.

  TDM_25x80—Splits 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.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    // Call the TEXT display constructor.
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
}
```

```
ExampleFunction2(int graphics)
{
    // Put a level of abstraction into the call.
    UI_DISPLAY *display;
    display = (graphics) ?
        (UI_DISPLAY *)new UI_DOS_BGI_DISPLAY :
        (UI_DISPLAY *)new UI_DOS_TEXT_DISPLAY(TDM_43x80);
    .
    .
    .
}

ExampleFunction3()
{
    // Create two displays (dual screen monitor support).
    UI_DOS_BGI_DISPLAY display1;
    UI_DOS_TEXT_DISPLAY *display2 = new UI_DOS_TEXT_DISPLAY;
    .
    .
    .
}
```

UI_DOS_TEXT_DISPLAY::~UI_DOS_TEXT_DISPLAY

Syntax

```
#include <ui_dsp.hpp>

virtual UI_DOS_TEXT_DISPLAY::~UI_DOS_TEXT_DISPLAY(
    void);
```

Remarks

This virtual destructor destroys the class information associated with the UI_DOS_TEXT_DISPLAY object and clears the screen display.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    // The display destructor is called automatically when
    // the scope of this function ends.
}

ExampleFunction2(int graphics)
{
    UI_DISPLAY *display;
    display = (graphics) ?
        (UI_DISPLAY *)new UI_DOS_BGI_DISPLAY :
        (UI_DISPLAY *)new UI_DOS_TEXT_DISPLAY(TDM_AUTO);
    .
    .
    .
    // Call the display destructor.
    delete display;
}
```

```
ExampleFunction3()
{
    UI_DOS_BGI_DISPLAY display1;
    UI_DOS_TEXT_DISPLAY *display2 = new UI_DOS_TEXT_DISPLAY;
    .
    .
    .
    // Delete display2 since it is a UI_DOS_BGI_DISPLAY
    // class pointer.
    delete display2;
    // Display1 is destroyed when the scope of the display ends.
}
```

UI_DOS_TEXT_DISPLAY::Bitmap

Syntax #include <ui_dsp.hpp>

void UI_DOS_TEXT_DISPLAY::Bitmap(int *screenID*,
const UI_REGION *&region*, const USHORT **bitmap*,
const UI_PALETTE **palette*, int *fillBackground*);

Remarks This function is a stub. (See "Chapter 6—UI_DOS_BGI_DISPLAY" of this manual for the graphics equivalent of this function.

UI_DOS_TEXT_DISPLAY::Fill

Syntax #include <ui_dsp.hpp>

void UI_DOS_TEXT_DISPLAY::Fill(int *screenID*,
const UI_REGION *&region*, const UI_PALETTE **palette*);

Remarks This function fills a rectangular, two-dimensional region on the screen with the background color and pattern specified by *palette*.

1—The screen display for all devices (e.g., the mouse and cursor) are shut off before the specified region is filled. This prevents devices from overwriting the screen display.

2—The region is clipped according to the screen identification. This prevents writing to overlapping window regions.

3—The fill color is specified by the background portion of *palette*.

- *screenID*$_{in}$ is a screen object identification used to determine the parts of the fill region that can be updated to the screen. Only those screen locations that match *screenID* are updated. (See "Chapter 5—UI_DISPLAY" of this manual for more information about screen identifications.)

- *region*$_{in}$ is a reference pointer to the desired fill region. *Region.left* and *region.top* are pixel coordinate values that specify the left-top corner position of the fill region. *Region.right* and *region.bottom* values specify the right-bottom corner position of the fill region.

- *palette*$_{in}$ is a pointer to the palette used when filling the screen region. The palette's background color is used for the fill.

Example

```
#include <graphics.h>  // include for graphic colors.
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    UI_REGION region = { 10, 10, 40, 20 };
    // Make sure ID_SCREEN gets the whole screen.
    display->RegionDefine(ID_SCREEN, region);
    display->Fill(ID_SCREEN, region, _backgroundPalette);
    .
    .
    .
}

UI_PALETTE palette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(BLUE, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};

ExampleFunction2()
{
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    .
    .
    .
    // Fill the specified region.  Notice the region is defined to
    // be the whole screen, but only the region with the
    // identification -1 will actually be updated.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display->columns / 4;
    region.bottom = display->lines / 4;
    display->RegionDefine(-1, region);
    region.right = display->columns - 1;
    region.bottom = display->lines - 1;
    display->Fill(-1, region, palette);
}
```

```
ExampleFunction3()
{
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    .
    .
    .
    // Fill the whole screen with the background palette.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display->columns - 1;
    region.bottom = display->lines - 1;
    // Make sure the full screen is defined.
    display->RegionDefine(ID_SCREEN, region);
    display->Fill(ID_SCREEN, region, _backgroundPalette);
}
```

UI_DOS_TEXT_DISPLAY::FillXOR

Syntax #include <ui_dsp.hpp>

void UI_DOS_TEXT_DISPLAY::FillXOR(
const UI_REGION *&region*);

Remarks This function XOR fills a rectangular, two dimensional region of the screen. This function differs from the **UI_DOS_TEXT_DISPLAY::Fill** function in the following respects:

1—No screen identification is required. The **UI_DOS_TEXT_-DISPLAY::FillXOR** function automatically XOR fills the region, independent of the screen identification for the region.

2—No palette is required. The XOR region is the opposite color of the current screen color on the specified pixel locations. To remove an XOR region, simply XOR the same region again.

This function differs from the **UI_DOS_TEXT_DISPLAY::RectangleXOR** function (described below), because it XORs the whole region, not just the edge of the region.

- *region*$_{in}$ is a reference pointer to the desired XOR region. *Region.left* and *region.top* are pixel coordinate values that specify the left-top corner position of the XOR region. *Region.right* and *region.bottom* are pixel coordinate values that specify the right-bottom corner position of the XOR region.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    // Fill an XOR region on the screen.
    UI_REGION region = { 100, 100, 150, 150 };
    display.FillXOR(region);
    .
    .
    .
    // Remove the XOR region.
    display.FillXOR(region);
    .
    .
    .
}
```

UI_DOS_TEXT_DISPLAY::Line

Syntax

#include <ui_dsp.hpp>

void UI_DOS_TEXT_DISPLAY::Line(int *screenID*, int *left*, int *top*, int *right*, int *bottom*, const UI_PALETTE **palette*, int *width* = 1);

Remarks

This function draws a graphics line between two points specified by top-left and bottom-right in the color specified by the background portion of *palette*.

1—The screen display for all devices (e.g., the mouse and cursor) are shut off before the specified line is drawn. This prevents devices from overwriting the screen display.

2—The region is clipped according to the screen identification. This prevents writing to overlapping window regions.

3—The line color is specified by the background portion of *palette*.

- $screenID_{in}$ 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. (See "Chapter 5—UI_DISPLAY" of this manual for more information about screen identifications.)

- *left*$_{in}$ is the beginning left position of the line on the screen (in character coordinates). This argument, combined with *top*, forms the beginning point of the line.
- *top*$_{in}$ is the beginning top position of the line on the screen (in character coordinates). This argument, combined with *left*, forms the beginning point of the line.
- *right*$_{in}$ is the ending right position of the line on the screen (in character coordinates). This argument, combined with *bottom*, forms the ending point of the line.
- *bottom*$_{in}$ is the ending bottom position of the line on the screen (in character coordinates). This argument, along with *right*, forms the ending point of the line.
- *palette*$_{in}$ is a pointer to the palette used when drawing the graphics line. The palette's foreground color is used to draw the line.
- *width*$_{in}$ specifies the line width. If *width* is 1, a single line is drawn (i.e., the line is drawn with the following text characters: "– |"). If *width* is 2, a double width line is drawn (i.e., the line is drawn with the following text characters: "= ‖").

Example

```
#include <graphics.h>   // include for graphic colors.
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    // Draw a diagonal line on the background display.
    display.Line(-1, 10, 10, 20, 20, _backgroundPalette);
    .
    .
    .
}
```

```
UI_PALETTE palette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(BLUE, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};

ExampleFunction2()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    // Draw a line to a pre-defined region of the screen.
    // Notice the coordinates are defined to draw a line on the
    // whole screen, but only the region with the identification
    // -1 will actually be updated.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display->columns / 4;
    region.bottom = display->lines / 4;
    display.RegionDefine(-1, region);
    display.Line(-1, 0, 0, display->columns - 1,
        display->lines - 1, palette);
}

ExampleFunction3()
{
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    .
    .
    .
    // Draw a diagonal line across the screen.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display->columns - 1;
    region.bottom = display->lines - 1;
    // Make sure the full screen is defined.
    display->RegionDefine(ID_SCREEN, region);
    display->Fill(ID_SCREEN, 0, 0, display->columns - 1,
        display->lines - 1, _backgroundPalette);
    .
    .
    .
}
```

UI_DOS_TEXT_DISPLAY::Rectangle

Syntax #include <ui_dsp.hpp>

void UI_DOS_TEXT_DISPLAY::Rectangle(int *screenID*, const UI_REGION *&region*, const UI_PALETTE **palette*, int *width* = 1);

Remarks This function draws a rectangle on the screen. The background color of the color palette is the color used when drawing the rectangle.

1—The screen display for all devices (e.g., the mouse and cursor) are shut off before the specified rectangle is drawn. This prevents devices from overwriting the screen display.

2—The rectangle is clipped according to the screen identification. This prevents writing to overlapping window regions.

3—The rectangle color is specified by the background portion of *palette*.

- *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. (See "Chapter 5—UI_DISPLAY"of this manual for more information about screen identifications.)

- *region*$_{in}$ is a reference pointer to the desired rectangle region. *Region.left* and *region.top* are pixel coordinate values that specify the left-top corner position of the rectangle. *Region.right* and *region.bottom* are pixel coordinate values that specify the right-bottom corner position of the rectangle.

- *palette*$_{in}$ is a pointer to the palette used when drawing the rectangle. The palette background color is used to draw the rectangle.

- *width*$_{in}$ specifies the rectangle width. If *width* is 1, a single width rectangle is drawn (i.e., the rectangle is drawn with the following text characters: "┌ ┐ ┘ └ – |"). If *width* is 2, a double width rectangle is drawn (i.e., the rectangle is drawn with the following text characters: "┌ ┐ ┘ └ – |").

Example

```
#include <graphics.h>   // include file for the graphics colors.
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    // Draw a rectangle to the screen's background.
    UI_REGION region = { 10, 10, 20, 20 };
    // Make sure ID_SCREEN gets the whole screen.
```

```
    display.RegionDefine(ID_SCREEN, region);
    display.Rectangle(ID_SCREEN, region, _backgroundPalette);
    .
    .
    .
}

UI_PALETTE palette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(BLUE, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};

ExampleFunction2()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    // Draw a rectangle.  Notice the region is defined to be
    // the whole screen, but only the region with the
    // identification -1 will actually be updated.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display->columns / 4;
    region.bottom = display->lines / 4;
    display.RegionDefine(-1, region);

    region.right = display->columns - 1;
    region.bottom = display->lines - 1;
    display.Rectangle(-1, region, palette);
}

ExampleFunction3()
{
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    .
    .
    .

    // Draw a rectangle around the whole screen with the
    // background palette.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display->columns - 1;
    region.bottom = display->lines - 1;
    // Make sure the full screen is defined.
    display->RegionDefine(ID_SCREEN, region);
    display->Rectangle(ID_SCREEN, region, _backgroundPalette);
    .
    .
    .
}
```

UI_DOS_TEXT_DISPLAY::RectangleXOR

Syntax

```
#include <ui_dsp.hpp>

void UI_DOS_BGI_DISPLAY::RectangleXOR(
    const UI_REGION &region);
```

Remarks

This function draws an XOR rectangle on the screen. This function differs from the **UI_DOS_TEXT_DISPLAY::Rectangle** function in the following respects:

1—No screen identification is required. The **UI_DOS_TEXT_DISPLAY::RectangleXOR** function automatically XOR fills the rectangle, independent of the screen identification for the rectangle.

2—No palette is required. The XOR rectangle is the opposite color of the current screen color on the specified pixel locations. To remove an XOR rectangle, simply XOR the same rectangle again.

This function differs from the **UI_DOS_TEXT_DISPLAY::FillXOR** function (described above), because it does not fill the region, but rather, draws a box at the edge of the region.

- $region_{in}$ is a reference to the desired XOR rectangle. *Region.left* and *region.top* are pixel coordinate values that specify the left-top corner position of the XOR rectangle. *Region.right* and *region.bottom* are pixel coordinate values that specify the right-bottom corner position of the XOR rectangle.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    // Draw an XOR rectangle on the screen.
    UI_REGION region = { 10, 10, 15, 15 };
    display.RectangleXOR(region);
    .
    .
    .
    // Remove the XOR rectangle.
    display.RectangleXOR(region);
    .
    .
    .
}
```

UI_DOS_TEXT_DISPLAY::RegionConvert

Syntax

```
#include <ui_dsp.hpp>

void UI_DOS_TEXT_DISPLAY::RegionConvert(
    UI_REGION &region);
```

Remarks

This function converts a graphics (or pixel) region to a cell (or text) region. The purpose of this function is to allow portability of higher-level display code.

NOTE: This function does not check for regions that have already been converted from graphics to text coordinates.

- *region*$_{in}$ is a reference to the cell region to be converted.

Example

```
#include <graphics.h>   // include for the palette colors.
#include <ui_dsp.hpp>

UI_PALETTE palette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, RED, WHITE
};
```

```
ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    // Convert cell coordinates to draw a box.
    // (Display independent)
    UI_REGION region = { 0, 0, 10, 10 };    // Cell coordinates.
    display.RegionConvert(region);

    display.Fill(ID_SCREEN, region, palette);
}
ExampleFunction2()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    // Convert cell coordinates to draw a box.
    // (Display independent)
    UI_REGION *region = { 0, 0, 10, 10 };   // Cell coordinates.
    display.RegionConvert(region);
    display.RegionDefine(-1, region);
    display.Rectangle(-1, *region, palette);
}
```

UI_DOS_TEXT_DISPLAY::Text

Syntax #include <ui_dsp.hpp>

void UI_DOS_TEXT_DISPLAY::Text(int *screenID*, int *left*, int *top*, const char **text*, const UI_PALETTE **palette*, int *length* = -1, int *fillBackground* = TRUE);

Remarks This 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. (See "Chapter 5—UI_DISPLAY" of this manual for more information about screen identifications.)
- $left_{in}$ is the beginning left position of the text on the screen (in character coordinates). This argument, combined with *top*, forms the beginning point where the text is to be displayed.

- *top*$_{in}$ is the beginning top position of the text on the screen (in character coordinates). This argument, combined with *left*, forms the beginning point where the text is to be displayed.

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

- *palette*$_{in}$ is a pointer to the palette used when drawing the text.

- *length*$_{in}$ is the number of characters to display on the screen. If this argument value is set to -1, then the string is displayed until a NULL character '\0' is found.

- *fillBackground*$_{in}$ is ignored.

Example

```
#include <graphics.h>   // include file for the graphics colors.
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    // Display text to the top-left corner of the screen.
    UI_REGION region = { 100, 100, 200, 200 };
    // Make sure ID_SCREEN gets the whole screen.
    display.RegionDefine(ID_SCREEN, region);
    display.Text(ID_SCREEN, 0, 0, "This is sample text",
        _backgroundPalette);
    .
    .
    .
}

UI_PALETTE palette =
{
    '\260', attrib(BLUE, LIGHTGRAY),
    attrib(MONO_NORMAL, MONO_BLACK),
    SOLID_FILL, attrib(BLUE, WHITE),
    attrib(BW_BLACK, BW_WHITE), attrib(GS_BLACK, GS_WHITE)
};

ExampleFunction2()
{
    UI_DOS_TEXT_DISPLAY display;
    .
    .
    .
    // Display the text.  Notice the text fills a large portion
    // of the whole screen, but only the region with the
    // identification -1 will actually be updated.
    UI_REGION region;
    region.left = region.top = 0;
    region.right = display->columns / 4;
    region.bottom = display->lines / 4;
    display.RegionDefine(-1, region);
    display.Text(-1, 0, 0,
        "This is sample text that doesn't all fit in the region",
```

```
            palette);
}
```

UI_DOS_TEXT_DISPLAY::TextHeight

Syntax

#include <ui_dsp.hpp>

int UI_DOS_TEXT_DISPLAY::TextHeight(const char *_string_);

Remarks

This function returns the height of a specified string in character size.

- $returnValue_{out}$ is always 1.
- $string_{in}$ is ignored.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;

    // Get the display independent size of text.
    int height = display.TextHeight("This is sample text");
    .
    .
    .
}
```

UI_DOS_TEXT_DISPLAY::TextWidth

Syntax

#include <ui_dsp.hpp>

int UI_DOS_TEXT_DISPLAY::TextWidth(const char *_string_);

Remarks

This function returns the width of a specified string in characters.

- $returnValue_{out}$ is the width of the string in characters.
- $string_{in}$ is a pointer to the string whose width is to be determined.

Example

```
#include <ui_dsp.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;

    // Get the display independent size of text.
    int height = display.TextHeight("This is sample text");
    .
    .
    .
}
```

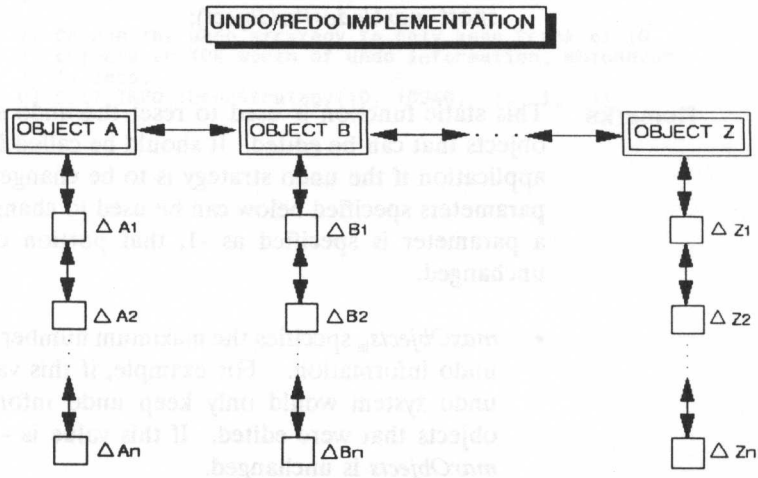
CHAPTER 8 – UI_EDIT_INFO

Overview

The UI_EDIT_INFO class serves as one of the base classes to all window object classes that can be edited. (The other base class is UI_WINDOW_OBJECT). The public members of the UI_EDIT_INFO class (declared in **UI_WIN.HPP**) are:

```
class UI_EDIT_INFO
{
public:
    static void UndoStrategy(short maxObjects, long maxBytes,
        short maxUndos, long maxBytesPerObject,
        short maxUndosPerObject);
};
```

The graphic image below shows the conceptual operation of the undo system within the Zinc Interface Library:



Whenever an end-user enters, deletes, or modifies information in a field, the undo system keeps track of the changes from the original information. Information concerning the strategy used to save information changes can be set by calling the **UI_EDIT_INFO::-UndoStrategy** function (discussed below).

See also

The example file **XEDIT.CPP**, which gives a complete example of the UI_EDIT_INFO class.

"Chapter 29—UIW_FORMATTED_STRING" of this manual, which describes a window object derived from the UI_EDIT_INFO class.

"Chapter 34—UIW_NUMBER" of this manual, which describes a window object derived from the UI_EDIT_INFO class.

"Chapter 41—UIW_STRING" of this manual, which describes a window object derived from the UI_EDIT_INFO class.

UI_EDIT_INFO::UndoStrategy

Syntax

#include <ui_gen.hpp>

static void UI_EDIT_INFO::UndoStrategy(int *maxObjects*, long *maxBytes*, int *maxUndos*, long *maxBytesPerObject*, short *maxUndosPerObject*);

Remarks

This static function is used to reset the undo strategy associated with objects that can be edited. It should be called in the early stages of an application if the undo strategy is to be changed. One or more of the parameters specified below can be used to change the undo strategy. If a parameter is specified as -1, that portion of the undo strategy is unchanged.

- *maxObjects*$_{in}$ specifies the maximum number of objects that can have undo information. For example, if this value were set to 10, the undo system would only keep undo information on the last 10 objects that were edited. If this value is -1 the undo strategy for *maxObjects* is unchanged.

- *maxBytes*$_{in}$ specifies the maximum amount of space (in bytes) to be used by the undo system. For example, if the programmer specified the maximum byte space to be 10k, the undo system would only keep 10k worth of undo information. If this value is -1 the undo strategy for *maxBytes* is unchanged.

- *maxUndos*$_{in}$ specifies the maximum number of undo operations that are saved by the undo system. If this value is -1 the undo strategy for *maxUndos* is unchanged.

- *maxBytesPerObject*$_{in}$ specifies the maximum number of bytes a particular edit object may take for its undo operations. If this value is -1 the undo strategy for *maxBytesPerObject* is unchanged.
- *maxUndosPerObject*$_{in}$ specifies the maximum number of undo operations that can be saved per edit object. If this value is -1 the undo strategy for *maxUndosPerObject* is unchanged.

Example

```
#include <ui_win.hpp>

ElementFunction1()
{
    // Initialize the system.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    // Change the undo strategy to only keep track of 10
    // objects or 10k worth of undo information, whichever
    // is less.
    UI_EDIT_INFO::UndoStrategy(10, 10240, -1, -1, -1);
    .
    .
    .
}
```

CHAPTER 9 – UI_ELEMENT

Overview

The UI_ELEMENT class serves as the base class to all window object classes, all device classes and several other specialized classes in the Zinc Interface Library. The public members of the UI_ELEMENT class (declared in **UI_GEN.HPP**) are:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;

    UI_ELEMENT(void);
    virtual ~UI_ELEMENT(void);
};
```

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

See also

The example file **XELEMENT.CPP**, which gives a complete example of the UI_ELEMENT class.

"Chapter 4—UI_DEVICE" of this manual, which describes an abstract class derived from the UI_ELEMENT class. The UI_DEVICE class is used as the base class for input devices such as the keyboard (UI_BIOS_KEYBOARD) and mouse (UI_MS_MOUSE) classes.

"Chapter 17—UI_LIST" of this manual, which describes the operation of list elements in a list class.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes a class derived from the UI_ELEMENT class. The UI_WINDOW_-OBJECT class is used as the base class for all the window objects described in the Programmer's Reference (e.g., UIW_BORDER, UIW_BUTTON, UIW_STRING, UIW_TEXT).

UI_ELEMENT::UI_ELEMENT

Syntax

```
#include <ui_gen.hpp>

UI_ELEMENT::UI_ELEMENT(void);
```

Remarks This constructor returns a pointer to a new UI_ELEMENT object.

Example

```
#include <ui_gen.hpp>

ElementFunction1()
{
    // Each declaration below calls the UI_ELEMENT constructor.
    UI_ELEMENT element1;
    UI_ELEMENT *element2;
    element2 = new UI_ELEMENT;
    UI_ELEMENT *element3 = new UI_ELEMENT;
    .
    .
    .
}
```

UI_ELEMENT::~UI_ELEMENT

Syntax

```
#include <ui_gen.hpp>

virtual UI_ELEMENT::~UI_ELEMENT(void);
```

Remarks This 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>

ElementFunction1()
{
    UI_ELEMENT element1;
    UI_ELEMENT *element2;
    element2 = new UI_ELEMENT;
    UI_ELEMENT *element3 = new UI_ELEMENT;
    .
    .
    .
```

```
    // Call the destructor for elements 2 and 3.  The
    // element1 destructor is automatically called when the
    // scope of this function ends.
    delete element3;
    delete element2;
}
```

CHAPTER 10 – UI_ERROR_SYSTEM

Overview

The UI_ERROR_SYSTEM class is the base class that the Zinc Interface Library and programmers use to report run-time errors. It is the default error class if no other class is specified. The public members of the UI_ERROR_SYSTEM class (declared in **UI_WIN.HPP**) are:

```
class UI_ERROR_SYSTEM
{
public:
    // UI_ERROR_SYSTEM(void);  Use the default C++ constructor.
    virtual ~UI_ERROR_SYSTEM(void);
    virtual void Beep(void);

    virtual void ReportError(UI_WINDOW_MANAGER *windowManager,
        USHORT objectFlags, char *format, ...);
};
```

Global variables

UI_ERROR_SYSTEM **_errorSystem* is a pointer to the error system. This variable originally points to a static UI_ERROR_SYSTEM class object. It may be reset by the programmer to point to a class object derived from the UI_ERROR_SYSTEM base class. The external declaration for this variable is contained in **UI_WIN.HPP**. The actual declaration of this variable is contained in **G_ERROR.CPP**.

See also

The example file **XERR.CPP**, which gives a complete example of the UI_ERROR_SYSTEM class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the error system.

"Chapter 11—UI_ERROR_WINDOW_SYSTEM" of this manual, which describes a window based error system class.

UI_ERROR_SYSTEM::UI_ERROR_SYSTEM

Syntax

#include <ui_win.hpp>

UI_ERROR_SYSTEM::UI_ERROR_SYSTEM(void);

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

NOTE: If the constructed error system class object is assigned to the global variable *_errorSystem* it is not automatically destroyed by the system. Therefore, the class object should be destroyed by the programmer at the end of the application program.

Example

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

ExampleFunction1()
{
    // Initialize the Zinc Interface Library.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);

    _errorSystem = new UI_ERROR_WINDOW_SYSTEM;
    _helpSystem = new UI_HELP_WINDOW_SYSTEM("demo.hlp",
        windowManager);
    .
    .
    .
    // Restore the basic error and help systems.
    delete _errorSystem;
    _errorSystem = new UI_ERROR_SYSTEM;
    delete _helpSystem;
    _helpSystem = new UI_HELP_SYSTEM;
    .
    .
    .
}
```

UI_ERROR_SYSTEM::~UI_ERROR_SYSTEM

Syntax #include <ui_win.hpp>

virtual UI_ERROR_SYSTEM::~UI_ERROR_SYSTEM(void);

Remarks This virtual destructor destroys the class information associated with the UI_ERROR_SYSTEM object.

Example

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

ExampleFunction1()
{
    // Initialize the Zinc Library Interface.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);

    _errorSystem = new UI_ERROR_WINDOW_SYSTEM;
    _helpSystem = new UI_HELP_WINDOW_SYSTEM("demo.hlp",
        windowManager);
    .
    .
    .
    // Restore the basic error and help systems.
    delete _helpSystem;
    _helpSystem = new UI_HELP_SYSTEM;
    delete _errorSystem;
    _errorSystem = new UI_ERROR_SYSTEM;
    .
    .
    .
    // Clean up. Order is important!
    delete _helpSystem;
    delete _errorSystem;
    delete windowManager;
    delete eventManager;
    delete display;
}
```

UI_ERROR_SYSTEM::ReportError

Syntax

#include <ui_win.h>

virtual void UI_ERROR_SYSTEM::ReportError(
    const UI_WINDOW_MANAGER **windowManager*,
    USHORT *objectFlags*, const char **format*[, *argument*, ...]);

Remarks

This virtual function is used to report an error via the error system. The **UI_ERROR_SYSTEM::ReportError** function simply beeps. This function is declared virtual, so additional error systems can be written to override this function. All arguments—*windowManager*, *objectFlags*, *format* and [*argument*, ...]—are unused by this function.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Initialize the Zinc Library Interface.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    // Since it is the default error system.  This function just
    // beeps.
    errorSeverity = "critical";
    _errorSystem->ReportError(_windowManager, -1,
        "This is a %s error.", errorSeverity);
    .
    .
    .
}
```

CHAPTER 11 – UI_ERROR_WINDOW_SYSTEM

Overview The UI_ERROR_WINDOW_SYSTEM class is a windowed implementation of the UI_ERROR_SYSTEM class, which is used to report run-time errors. The public members of the UI_ERROR_WINDOW_SYSTEM class (declared in **UI_WIN.HPP**) are:

```
class UI_ERROR_WINDOW_SYSTEM : public UI_ERROR_SYSTEM
{
public:
    UI_ERROR_WINDOW_SYSTEM(void);
    virtual ~UI_ERROR_WINDOW_SYSTEM(void);

    virtual void ReportError(UI_WINDOW_MANAGER *windowManager,
        USHORT objectFlags, char *format, ...);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ERROR_SYSTEM
{
public:
    virtual void Beep(void);

    virtual void ReportError(UI_WINDOW_MANAGER *windowManager,
        USHORT objectFlags, char *format, ...);
};

class UI_ERROR_WINDOW_SYSTEM : public UI_ERROR_SYSTEM;
```

See also The example file **XERR.CPP**, which gives a complete example of the UI_ERROR_WINDOW_SYSTEM class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the error system.

"Chapter 10—UI_ERROR_SYSTEM" of this manual, which describes the base class from which the UI_ERROR_WINDOW_SYSTEM class is derived.

UI_ERROR_WINDOW_SYSTEM::UI_ERROR_WINDOW_SYSTEM

Syntax

```
#include <ui_win.h>

UI_ERROR_WINDOW_SYSTEM::
    UI_ERROR_WINDOW_SYSTEM(void);
```

Remarks

This constructor returns a pointer to a new UI_ERROR_WINDOW_-SYSTEM class object.

NOTE: If the constructed error system class object is assigned to the global variable *_errorSystem* it is not automatically destroyed by the system. Therefore, the class object should be destroyed by the programmer at the end of the application program.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Initialize the Zinc Library Interface.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);

    _errorSystem = new UI_ERROR_WINDOW_SYSTEM;
    _helpSystem = new UI_HELP_WINDOW_SYSTEM("demo.hlp",
        windowManager);
    .
    .
    .
    // Restore the basic error and help systems.
    delete _helpSystem;
    _helpSystem = new UI_HELP_SYSTEM;
    delete _errorSystem;
    _errorSystem = new UI_ERROR_SYSTEM;
    .
    .
    .
}
```

UI_ERROR_WINDOW_SYSTEM::~UI_ERROR_WINDOW_SYSTEM

Syntax #include <ui_win.h>

virtual UI_ERROR_WINDOW_SYSTEM::
~UI_ERROR_WINDOW_SYSTEM(void);

Remarks This virtual destructor destroys the class information associated with the UI_ERROR_WINDOW_SYSTEM object.

Example

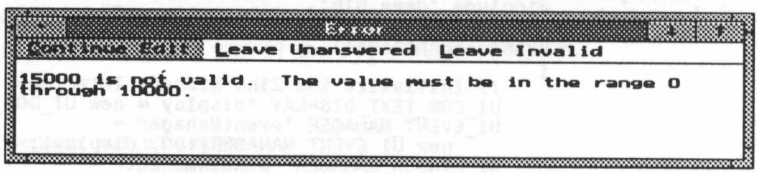
```
#include <ui_win.hpp>
#include "demo.hlh"

ExampleFunction1()
{
    // Initialize the Zinc Library Interface.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);

    _errorSystem = new UI_ERROR_WINDOW_SYSTEM;
    _helpSystem = new UI_HELP_WINDOW_SYSTEM("demo.hlp",
        windowManager);
    .
    .
    .
    // Restore the basic error and help systems.
    delete _helpSystem;
    _helpSystem = new UI_HELP_SYSTEM;
    delete _errorSystem;
    _errorSystem = new UI_ERROR_SYSTEM;
    .
    .
    .
    // Restore the Zinc Library Interface. Order is important!
    delete _helpSystem;
    delete _errorSystem;
    delete windowManager;
    delete eventManager;
    delete display;
}
```

UI_ERROR_WINDOW_SYSTEM::ReportError

Syntax #include <ui_win.h>

virtual void UI_ERROR_WINDOW_SYSTEM::ReportError(
const UI_WINDOW_MANAGER *_windowManager_,
USHORT _objectFlags_, const char *_format_[, _argument_, ...]);

Remarks *This function is used to report an error via the error system. The figure* below shows a graphic UI_ERROR_WINDOW_SYSTEM presentation window:



The error message is displayed in the text portion of the window (shown as "15000 is not valid...").

- *windowManager*$_{in}$ is a pointer to the window manager where the error message is to be displayed.
- *objectFlags*$_{in}$ indicate the type of action that can take place when the error occurs. If the error code is not -1, the "Leave unanswered" and "Leave invalid" options are selectable. The "Continue edit" option is always available.
- *format*$_{in}$ is the **printf** style format that controls how the string is converted.
- *argument*, ... are the **printf** style arguments that are used by the *format* string.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Initialize the Zinc Library Interface.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
```

```
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    _errorSystem = new UI_ERROR_WINDOW_SYSTEM;
    .
    .
    .
    // Report an error. -1 only allows the user to press escape.
    // Since it is the window error system a window is displayed
    // with the message.
    errorSeverity = "critical";
    _errorSystem->ReportError(_windowManager, -1,
        "This is a %s error.", errorSeverity);
    .
    .
    .
}
```

CHAPTER 12 – UI_EVENT

Overview The UI_EVENT structure is used to store all information passed through the Zinc Interface Library. The UI_EVENT structure (declared in **UI_EVT.HPP**) has the following fields:

```
struct UI_KEY
{
    UCHAR value;
    UCHAR shiftState;
};

struct UI_POSITION
{
    int column;
    int line;
};

struct UI_REGION
{
    int left;
    int top;
    int right;
    int bottom;
};

struct UI_EVENT
{
    int type;
    USHORT rawCode;
    union
    {
        UI_KEY key;
        UI_REGION region;
        UI_POSITION position;
        void *data;
    };
};
```

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

  -32,767 to -1,000—Reserved by the Zinc Interface Library for future use.

  -999 to -1—Reserved by the Zinc Interface Library for system messages. The following pre-defined system events (declared in **UI_EVT.HPP**) are important for the programmer to understand:

  S_CANCEL—The window manager received an event by the end-user that was a request to cancel the information in the current window. The current window can be obtained by calling the **UI_WINDOW_MANAGER::First** function.

S_CONTINUE—A message sent by a programmer specified procedure that requires a lot of processor time but wants to check the status of the event queue or give time to other device objects. If this message is sent to the event manager, the next event received by the window object will be the S_CONTINUE message.

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. This message is sent back to the programmer whenever the message is object specific and no object is attached to the window manager.

S_REDISPLAY—Re-displays the screen display. Sending this message causes the window manager to clear the screen display and repaint all the windows attached to the display.

S_UNKNOWN—The event passed to the window manager was not recognized by the window manager or any window attached to the screen display.

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 UI_CURSOR class object.

E_DEVICE(99)—Identification for generic device communication.

E_KEY(10)—Identification for the UI_BIOS_KEYBOARD class object.

E_MOUSE(30)—Identification for the UI_MS_MOUSE class object.

The following additional raw device identifications are reserved by the Zinc Interface Library for future use: 11-19, 31-39, 50-59, 71-79, 90-98. The remaining values 0-9, 20-29, 40-49, 60-69, 80-89 can be used by the programmer.

100 to 9,999—Reserved by the Zinc Interface Library for logical events. The following logical events (declared in **UI_MAP.HPP**) are important for the programmer to understand:

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_HELP—If this message is sent by the programmer to the window manager, help will be displayed about the application program. Otherwise, this message is received by the programmer, indicating help has been presented to the end-user.

10,000 to 32,767—Available to the programmer for private use. These values are not used by the Zinc Interface Library.

- *rawCode* is the raw code value for the type of sending device. The following devices (declared in **UI_EVT.HPP**) use the *rawCode* event field:

  UI_BIOS_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 0x3C00. In this case, the UI_EVENT structure would contain the following values:

  ```
  event.type = E_KEY;
  event.rawCode = 0x3C00;
  event.key.value = 0;        // low 8 bits of the rawCode.
  event.key.shiftState = 0;
  ```

  UI_MS_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.key.shiftState = <current mouse row position>;

- *key*, *region*, *position* and *data* are types of specific information associated with the event. For more information about the contents of these fields, refer to the type of event (discussed briefly above and referenced below).

See also The example file **XEVENT.CPP**, which gives a complete example of the UI_EVENT structure.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the use of events within the Zinc Interface Library.

"Chapter 1—UI_BIOS_KEYBOARD" of this manual, which describes an input device that uses the UI_EVENT structure to store input information.

"Chapter 13—UI_EVENT_MANAGER" of this manual, which describes the operation of class objects that generate event information.

"Chapter 14—UI_EVENT_MAP" of this manual, which describes the use of *event.types* and *event.rawCodes* in the system event mapping scheme.

"Chapter 18—UI_MS_MOUSE" of this manual, which describes an input device that uses the UI_EVENT structure to store input information.

"Chapter 24—UI_WINDOW_MANAGER" of this manual, which describes the operation of a class object that receives and processes event information.

Example

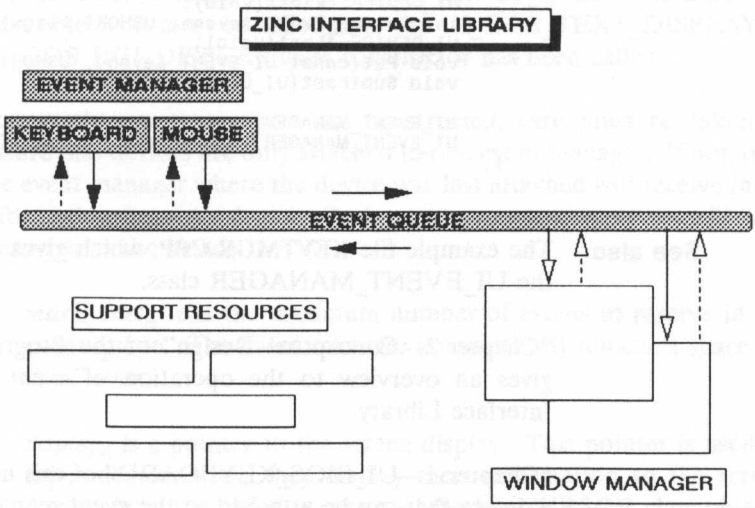
```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Attach the devices directly to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager
```

```
        + new UI_BIOS_KEYBOARD
        + new UI_MS_MOUSE
        + new UI_CURSOR;
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    UI_EVENT event;
    int ccode;
    do
    {
        // Get an event from the event manager.
        eventManager->Get(event, Q_NORMAL);

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

static void Exit(void *item, UI_EVENT &event)
{
    // 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);
}

ExampleFunction2()
{
    .
    .
    .
    // Construct the control pull-down item's menu.
    pullDownItem = new UIW_PULL_DOWN_ITEM(" ~Control ",
        MNF_NO_FLAGS, 0);
    *pullDownItem
        + new UIW_POP_UP_ITEM("~Help...", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS, Help));
        + new UIW_POP_UP_ITEM("E~xit", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS, Exit));
    .
    .
    .
}
```

CHAPTER 13 – UI_EVENT_MANAGER

Overview

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 the Zinc Interface Library modules (e.g., keyboard input information as well as system messages). The graphic illustration below shows the conceptual operation of the event manager within the library:



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 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 public members of the UI_EVENT_MANAGER class (declared in **UI_EVT.HPP**) are:

```
class UI_EVENT_MANAGER
{
public:
    UI_EVENT_MANAGER(int maxEvents, UI_DISPLAY *display);
    virtual ~UI_EVENT_MANAGER(void);

    void Add(UI_DEVICE *device);
    int DeviceState(int deviceType, USHORT deviceState);
    int Event(const UI_EVENT &event);
    UI_DEVICE *First(void);
    int Get(UI_EVENT &event, USHORT flags);
    UI_DEVICE *Last(void);
    void Put(const UI_EVENT &event, USHORT flags);
    void Subtract(UI_DEVICE *device);

    UI_EVENT_MANAGER &operator + (UI_DEVICE *device)
    UI_EVENT_MANAGER &operator - (UI_DEVICE *device)
};
```

See also The example file **XEVTMGR.CPP**, which gives a complete example of the UI_EVENT_MANAGER class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the operation of event manager in the Zinc Interface Library.

"Chapter 1—UI_BIOS_KEYBOARD" of this manual, which describes a device that can be attached to the event manager.

"Chapter 2—UI_CURSOR" of this manual, which describes a device that can be attached to the event manager.

"Chapter 4—UI_DEVICE" of this manual, which provides additional information about the operation (e.g., addition, subtraction, state change) of device classes within the event manager.

"Chapter 18—UI_MS_MOUSE" of this manual, which describes a device that can be attached to the event manager.

UI_EVENT_MANAGER::UI_EVENT_MANAGER

Syntax

#include <ui_evt.hpp>

UI_EVENT_MANAGER::UI_EVENT_MANAGER(int *maxEvents*, UI_DISPLAY **display*);

Remarks

This constructor returns a pointer to a new UI_EVENT_MANAGER class object. It must be called after the UI_DOS_TEXT_DISPLAY or UI_DOS_BGI_DISPLAY class constructor has been called.

If multiple event managers are constructed, care must be taken to ensure that devices are only attached to one event manager. If not, only the event manager where the device was last attached will receive input information from the device. Both event managers, however, will send messages to the device.

- $maxEvents_{in}$ tells the maximum number of events to reserve in the event queue. The event manager automatically allocates space for *maxEvents*.

- $display_{in}$ 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 UI_CURSOR class object).

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    UI_EVENT_MANAGER eventManager(100, &display);
    .
    .
    .
}
```

```
ExampleFunction2()
{
    // Attach the devices directly to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager
        + new UI_BIOS_KEYBOARD
        + new UI_MS_MOUSE
        + new UI_CURSOR;
    .
    .
    .
}
```

UI_EVENT_MANAGER::~UI_EVENT_MANAGER

Syntax #include <ui_evt.hpp>

virtual UI_EVENT_MANAGER::~UI_EVENT_MANAGER(void);

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_evt.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    UI_EVENT_MANAGER eventManager(100, &display);
    .
    .
    .
    // The destructors for the display and event manager are
    // automatically called when the scope of this routine ends.
}

ExampleFunction2()
{
    // Attach the devices directly to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager
        + new UI_BIOS_KEYBOARD
        + new UI_MS_MOUSE
        + new UI_CURSOR;
    .
    .
    .
```

```
    // Destroy the event manager (with the keyboard, mouse and
    // cursor) and the screen display.
    delete eventManager;
    delete display;
}
```

UI_EVENT_MANAGER::Add

Syntax #include <ui_evt.hpp>

void UI_EVENT_MANAGER::Add(UI_DEVICE **device*);

Remarks This function adds a new device to the event manager. The position of the device is determined by the event manager according to the device's type (device types are constant values ranging from 1 to 99). For example, the devices UI_BIOS_KEYBOARD and UI_MS_MOUSE each have a unique event identifier—E_KEY for UI_BIOS_-KEYBOARD, and E_MOUSE for UI_MS_MOUSE. The constant value for E_KEY is 10, while the constant value for E_MOUSE is 30. Thus, the UI_BIOS_KEYBOARD class object will always be placed in the event manager's device list before the UI_MS_MOUSE class object. The order of devices in the event manager is important because of the event manager's polling algorithm. For example, a recording device could be defined by the programmer. Such a device would need to be the last device polled to guarantee that all input information (keyboard and mouse) had been received by the event manager before the recorder evaluated the input queue.

- $device_{in}$ is a pointer to the new UI_DEVICE class object to be added to the event manager's device list.

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    // Attach the devices individually to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);

    UI_BIOS_KEYBOARD *keyboard = new UI_BIOS_KEYBOARD;
    UI_MS_MOUSE *mouse = new UI_MS_MOUSE;
```

```
        eventManager->Add(keyboard);
        eventManager->Add(mouse);
        eventManager->Add(new UI_CURSOR);
        .
        .
        .
}
```

UI_EVENT_MANAGER::DeviceState

Syntax #include <ui_evt.hpp>

int UI_EVENT_MANAGER::DeviceState(int *deviceType*, USHORT *deviceState*);

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 UI_BIOS_-KEYBOARD 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 UI_MS_-MOUSE class object (if it is in the device list).

  E_KEY—Sends the state information to the UI_BIOS_-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**) can be:

  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.

Any other device state that is recognized by *deviceType*. For example, the UI_MS_MOUSE also recognizes the following state information:

DM_EDIT—Displays a '|' cursor.
DM_DIAGONAL_ULLR—Displays a '⤡' cursor.
DM_DIAGONAL_LLUR—Displays a '⤢' cursor.
DM_MOVE—Displays a hand cursor.
DM_HORIZONTAL—Displays a '↔' cursor.
DM_VERTICAL—Displays a '↕' cursor.
DM_VIEW—Displays a normal '↖'cursor.
DM_WAIT—Displays an hour-glass cursor.

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    // Attach the devices directly to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager
        + new UI_BIOS_KEYBOARD
        + new UI_MS_MOUSE
        + new UI_CURSOR;

    // Display an hour glass until the program is ready
    // to receive user input.
    eventManager->DeviceState(E_MOUSE, DM_WAIT);
    .
    .
    .
}
```

UI_EVENT_MANAGER::Get

Syntax #include <ui_evt.hpp>

int UI_EVENT_MANAGER::Get(UI_EVENT *&event*,
USHORT *flags*);

Remarks This function gets an event from the event manager's input queue, if one is available.

- *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 until an event is received from one of the input devices.

  Q_DESTROY—Destroys the event information from the event manager after it is copied to *event*.

  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—Immediately returns from the **UI_EVENT_-MANAGER::Get** function, event 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_POLL and Q_DESTROY flags.

Q_POLL—Makes sure 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>

ExampleFunction1()
{
    // Attach the devices directly to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager
        + new UI_BIOS_KEYBOARD
        + new UI_MS_MOUSE
        + new UI_CURSOR;
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    UI_EVENT event;
    int ccode;
    do
    {
        // Get an event from the event manager.
        eventManager->Get(event, Q_NORMAL);

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

UI_EVENT_MANAGER::Put

Syntax #include <ui_evt.hpp>

void UI_EVENT_MANAGER::Put(const UI_EVENT *&event*, USHORT *position*);

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.
- *position*$_{in}$ indicates where to put the event. The following flags (declared in **UI_EVT.HPP**) are recognized by the **UI_EVENT_MANGER::Put** function:

  Q_BEGIN—Puts the event information at the beginning of the input queue.

  Q_END—Puts the event information at the end of the input queue.

Example

```
#include <ui_win.hpp>

static void Exit(void *item, UI_EVENT &event)
{
    // 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);
}

ExampleFunction1()
{
    .
    .
    .
    // Construct the control pull-down item's menu.
    pullDownItem = new UIW_PULL_DOWN_ITEM(" ~Control ",
        MNF_NO_FLAGS, 0);
    *pullDownItem
        + new UIW_POP_UP_ITEM("~Help...", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS, Help))
        + new UIW_POP_UP_ITEM("E~xit", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS, Exit));
    .
    .
    .
}
```

UI_EVENT_MANAGER::Subtract

Syntax #include <ui_evt.hpp>

void UI_EVENT_MANAGER::Subtract(UI_DEVICE **device*);

Remarks This function removes the specified device from the event manager's list of devices. This routine does not call the destructor associated with the device.

NOTE: If an input device (e.g, UI_BIOS_KEYBOARD, UI_CURSOR, UI_MS_MOUSE) is attached to the event manager, it will automatically be destroyed when the event manager is destroyed.

- $device_{in}$ is a pointer to the device to be removed from the event manager's list of devices. The *device* class destructor is not called by **UI_EVENT_MANAGER::Subtract.**

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    // Attach the devices individually to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);

    UI_BIOS_KEYBOARD *keyboard = new UI_BIOS_KEYBOARD;
    UI_MS_MOUSE *mouse = new UI_MS_MOUSE;

    eventManager->Add(keyboard);
    eventManager->Add(mouse);
    eventManager->Add(new UI_CURSOR);
    .
    .
    .
    eventManager->Subtract(keyboard);
    eventManager->Subtract(mouse);

    // This call automatically calls the destructor for UI_CURSOR.
    delete eventManager;
    delete keyboard;
    delete mouse;
    delete display;
}
```

UI_EVENT_MANAGER::operator +

Syntax

```
#include <ui_evt.hpp>

UI_EVENT_MANAGER &UI_EVENT_MANAGER::
    operator + (UI_DEVICE *device);
```

Remarks

This overload operator adds an input device to the event manager. This operator overload is equivalent to calling the **UI_EVENT_MANAGER::Add** routine, except that it allows the chaining of device additions to the event manager.

- *returnValue*$_{out}$ is the UI_EVENT_MANAGER reference. Returning the reference to the UI_EVENT_MANAGER object allows chaining of the **UI_EVENT_MANAGER::operator+** overload operator.

- *device*$_{in}$ is a pointer to the UI_DEVICE class object to be added to the event manager's device list.

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    // Attach the devices individually to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);

    UI_BIOS_KEYBOARD *keyboard = new UI_BIOS_KEYBOARD;
    UI_MS_MOUSE *mouse = new UI_MS_MOUSE;
    *eventManager
        + keyboard
        + mouse
        + new UI_CURSOR;
    .
    .
    .
}
```

UI_EVENT_MANAGER::operator –

Syntax

```
#include <ui_evt.hpp>

UI_EVENT_MANAGER &UI_EVENT_MANAGER::
    operator – (UI_DEVICE *device);
```

Remarks

This overload operator removes an input device from the event manager. This operator overload is equivalent to calling the **UI_EVENT_MANAGER::Delete** routine, except that it allows the chaining of device deletions from the event manager.

- $returnValue_{out}$ is the UI_EVENT_MANAGER reference. Returning the reference to the UI_EVENT_MANAGER object allows chaining of the **UI_EVENT_MANAGER::operator–** overload operator.
- $device_{in}$ is a pointer to the UI_DEVICE class object that is to be deleted from the event manager's device list.

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    // Attach the devices individually to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);

    UI_BIOS_KEYBOARD *keyboard = new UI_BIOS_KEYBOARD;
    UI_MS_MOUSE *mouse = new UI_MS_MOUSE;
    *eventManager
        + keyboard
        + mouse
        + new UI_CURSOR;
    .
    .
    .
    *eventManager
        - keyboard
        - mouse;

    // This call automatically calls the destructor for UI_CURSOR.
    delete eventManager;
    delete keyboard;
    delete mouse;
    delete display;
}
```

CHAPTER 14 – UI_EVENT_MAP

Overview

The UI_EVENT_MAP structure is used to map raw input device events to logical events. For example, the Zinc Interface Library declares default event mapping for the UI_BIOS_KEYBOARD and UI_MS_-MOUSE 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.

<Ctrl C> — Mapped to L_EXIT; 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 for editing operations.

The UI_EVENT_MAP structure (declared in **UI_MAP.HPP**) has the following fields:

```
struct UI_EVENT_MAP
{
    int windowID;
    int logicalValue;
    int eventType;
    USHORT rawCode;
};
```

- *windowID* is the window identification for which the match applies. (A full list of window identifications is given in **UI_MAP.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_TEXT).

- *logicalValue* is the logical event to map. (A full list of logical values is given in **UI_MAP.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).

  L_WINDOW_NEXT—Moves to the next window. This logical event is only understood by the window manager.

- *eventType* is the raw device identification. The following event types (declared in **UI_EVT.HPP**) are pre-defined by the Zinc Interface Library:

  E_KEY—Identification for the UI_BIOS_KEYBOARD class object. This device generates keyboard input information.

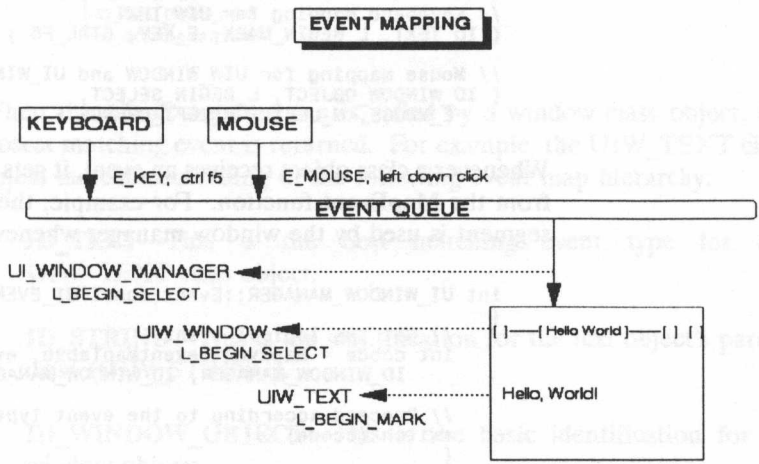
  E_MOUSE—Identification for the UI_MS_MOUSE class object. This device generates mouse input information.

- *rawCode* is a unique code that is used to map against the *event.rawCode* value. The following input devices (declared in **UI_EVT.HPP**) generate raw-code values that can be mapped with the UI_EVENT_MAP structure:

  UI_BIOS_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 0x3C00.

UI_MS_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 the Zinc Interface Library, raw events, received from input devices at run-time, are interpreted at each level of the application according to the type of operation. For example, the graphic illustration below shows how the <Ctrl F5> key and left mouse click would be interpreted at each level of the Zinc Interface Library (where a text field is the current window object):



The <Ctrl F5> key and left-mouse button are processed in the following manner:

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

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

- third, the window evaluates the event and passes it to the proper window object. The mouse event is interpreted as an L_BEGIN_SELECT logical event, while the keyboard event is passed directly to the UIW_TEXT window object.
- finally, the UIW_TEXT window object evaluates both the keyboard and mouse events as the L_BEGIN_MARK command.

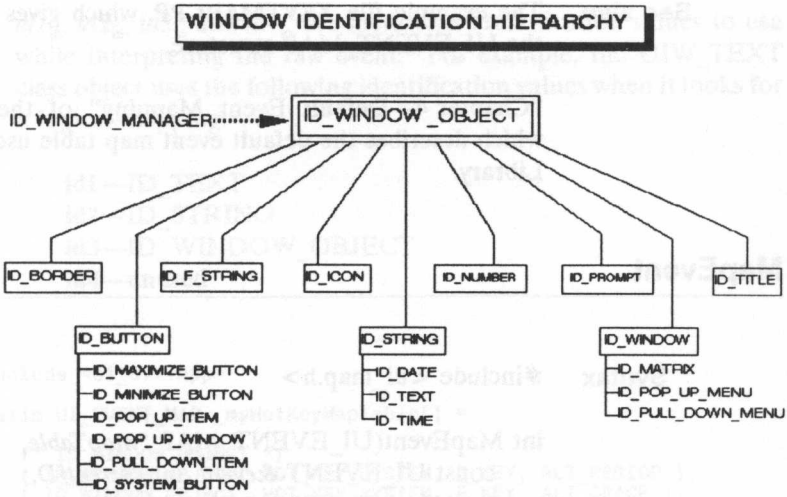
Logical mapping is accomplished through an event mapping procedure that uses the UI_EVENT_MAP structure as the interpretation key. The example above showed the use of three logical map entries:

```
// Mouse mapping for UIW_TEXT.
{ ID_TEXT, L_BEGIN_MARK, E_MOUSE, M_LEFT | M_LEFT_CHANGE }

// Keyboard mapping for UIW_TEXT.
{ ID_TEXT, L_BEGIN_MARK, E_KEY, CTRL_F5 }

// Mouse mapping for UIW_WINDOW and UI_WINDOW_MANAGER.
{ ID_WINDOW_OBJECT, L_BEGIN_SELECT,
    E_MOUSE, M_LEFT | M_LEFT_CHANGE }
```

Whenever a class object receives an event, it gets a logical interpretation from the **MapEvent** function. For example, the following partial code segment is used by the window manager whenever it gets an event:

```
int UI_WINDOW_MANAGER::Event(const UI_EVENT &event)
{
    // Get the logical event.
    int ccode = MapEvent(eventMapTable, event,
        ID_WINDOW_MANAGER, ID_WINDOW_MANAGER);

    // Proceed according to the event type.
    switch (ccode)
    {
    case L_WINDOW_MOVE:
    case L_WINDOW_SIZE:
        .
        .
        .
    case L_VIEW:
    case L_BEGIN_SELECT:
    case L_CONTINUE_SELECT:
    case L_END_SELECT:
        .
        .
        .
    }
    .
    .
    .
}
```

The figure below shows the logical event hierarchy:



When the **MapEvent** function is called by a window class object, the closest matching event is returned. For example, the UIW_TEXT class object matches according to the following event map hierarchy:

ID_TEXT—This is the best matching event type for the UIW_TEXT class object.

ID_STRING—This is the identification for the text object's parent class (UIW_STRING).

ID_WINDOW_OBJECT—This is the basic identification for all window objects.

Global variables

UI_EVENT_MAP **_eventMapTable* is a pointer to the event map table used by all UI_WINDOW_OBJECT class objects and the window manager to determine the logical meaning of raw events. The external declaration for this variable is contained in **UI_MAP.HPP**. The actual declaration of this variable is contained in **G_EVENT.CPP**.

UI_EVENT_MAP **_hotKeyMapTable* is a pointer to the hot key table used by all high-level windows to determine sub-object hot key equivalents. The external declaration for this variable is contained in **UI_MAP.HPP**. The actual declaration of this variable is contained in **G_HOTKEY.CPP**.

See also The example file **XEVTMAP.CPP**, which gives a complete example of the UI_EVENT_MAP structure.

"Chapter 4—Default Event Mapping" of the Programmer's Guide, which describes the default event map table used by the Zinc Interface Library.

MapEvent

Syntax

```
#include <ui_map.h>

int MapEvent(UI_EVENT_MAP *mapTable,
    const UI_EVENT &event, int currentID,
    int id1 = ID_WINDOW_OBJECT,
    int id2 = ID_WINDOW_OBJECT,
    int id3 = ID_WINDOW_OBJECT,
    int id4 = ID_WINDOW_OBJECT);
```

Remarks This 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.
- *currentID*$_{in}$ is the object identification to use as the best match. For example, the UIW_TEXT uses ID_TEXT as the current identification for its best match.

- $id1_{in}$, $id2_{in}$, $id3_{in}$ and $id4_{in}$ 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_STRING
  id3—ID_WINDOW_OBJECT
  id4—unused

Example

```
#include <ui_evt.hpp>

static UI_EVENT_MAP _myHotKeyMapTable[] =
{
    /* ID_WINDOW_OBJECT */
    { 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 }
    /* End of array */
    { ID_END, 0, 0, 0 }
};

const int MY_EVENT = -1;
static UI_EVENT_MAP _myEventMapTable[] =
{
    /* MY_EVENT */
    { MY_EVENT, L_EXIT, E_KEY, ALT_F10 },
    { MY_EVENT, L_EXIT, E_KEY, ALT_X },
    /* End of array */
    { ID_END, 0, 0, 0 }
};

ExampleFunction1()
{
    // Initialize the Zinc Interface Library.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager
        + new UI_BIOS_KEYBOARD + new UI_MS_MOUSE + new UI_CURSOR;
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    // Simplify the global hot key table.
    _hotKeyTable = _myHotKeyMapTable;
```

```
    // Process the system events.
    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);
    .
    .
    .
}
```

CHAPTER 15 – UI_HELP_SYSTEM

Overview

The UI_HELP_SYSTEM class is the base class that Zinc Interface Library uses to give help to an end-user during run-time. It is the default help class if no other class is specified. The public members of the UI_HELP_SYSTEM class (declared in **UI_WIN.HPP**) are:

```
class UI_HELP_SYSTEM
{
public:
    UCHAR installed;

    // UI_HELP_SYSTEM(void);
    virtual ~UI_HELP_SYSTEM(void);

    virtual void DisplayHelp(UI_WINDOW_MANAGER *windowManager,
        int helpContext);
};
```

- *installed* indicates whether the help system successfully installed. This value is TRUE if the help system is activated. Otherwise, this value is FALSE.

Global variables

UI_HELP_SYSTEM **_helpSystem* is a pointer to the Zinc Interface help system. The default setting for this variable points to a static UI_HELP_SYSTEM class object. The external declaration for this variable is contained in **UI_WIN.HPP**. The actual declaration of this variable is contained in **G_HELP.CPP**.

See also

The example file **XHELP.CPP**, which gives a complete example of the UI_HELP_SYSTEM class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the help system.

"Chapter 16—UI_HELP_WINDOW_SYSTEM" of this manual, which describes a window based help system class.

UI_HELP_SYSTEM::UI_HELP_SYSTEM

Syntax #include <ui_win.h>

UI_HELP_SYSTEM::UI_HELP_SYSTEM(void);

Remarks This constructor returns a pointer to a new UI_HELP_SYSTEM class object.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Initialize the Zinc Library Interface.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);

    _errorSystem = new UI_ERROR_WINDOW_SYSTEM;
    _helpSystem = new UI_HELP_WINDOW_SYSTEM("demo.hlp",
        windowManager);
    .
    .
    .
    // Restore the basic error and help systems.
    delete _helpSystem;
    _helpSystem = new UI_HELP_SYSTEM;
    delete _errorSystem;
    _errorSystem = new UI_ERROR_SYSTEM;
    .
    .
    .
    // Restore the Zinc Library Interface. Order is important!
    delete _helpSystem;
    delete _errorSystem;
    delete windowManager;
    delete eventManager;
    delete display;
}
```

UI_HELP_SYSTEM::~UI_HELP_SYSTEM

Syntax #include <ui_win.h>

virtual UI_HELP_SYSTEM::~UI_HELP_SYSTEM(void);

Remarks This virtual destructor destroys the class information associated with the UI_HELP_SYSTEM object.

NOTE: If the programmer re-defines the *_helpSystem* global variable the destructor for the help system is not called by the Zinc Interface Library.

Example

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

ExampleFunction1()
{
    // Initialize the Zinc Library Interface.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);

    _errorSystem = new UI_ERROR_WINDOW_SYSTEM;
    _helpSystem = new UI_HELP_WINDOW_SYSTEM("demo.hlp",
        windowManager);
    .
    .
    .
    // Restore the basic error and help systems.
    delete _helpSystem;
    _helpSystem = new UI_HELP_SYSTEM;
    delete _errorSystem;
    _errorSystem = new UI_ERROR_SYSTEM;
    .
    .
    .
    // Restore the Zinc Library Interface. Order is important!
    delete _helpSystem;
    delete _errorSystem;
    delete windowManager;
    delete eventManager;
    delete display;
}
```

UI_HELP_SYSTEM::DisplayHelp

Syntax

#include <ui_win.h>

virtual void UI_HELP_SYSTEM::DisplayHelp(
    UI_WINDOW_MANAGER **windowManager*, int *helpContext*);

Remarks

This virtual function is used to present help information via the help system. The **UI_HELP_SYSTEM::DisplayHelp** function is a stub. This function is declared virtual so that additional help systems can be written to override the function. All arguments—*windowManager* and *helpContext* —are unused by this function.

CHAPTER 16 – UI_HELP_WINDOW_SYSTEM

Overview

The UI_HELP_WINDOW_SYSTEM class is a windowed implementation of the UI_HELP_SYSTEM class, which is used to display help to the end-user during an application. The public members of the UI_HELP_WINDOW_SYSTEM class (declared in **UI_WIN.HPP**) are:

```
class UI_HELP_WINDOW_SYSTEM : public UI_HELP_SYSTEM
{
public:
    UI_HELP_WINDOW_SYSTEM(char *helpFileName,
        UI_WINDOW_MANAGER *windowManager,
        int defaultHelpContext = NO_HELP_CONTEXT);
    virtual ~UI_HELP_WINDOW_SYSTEM(void);

    virtual void DisplayHelp(UI_WINDOW_MANAGER *windowManager,
        int helpContext);
};
```

Inheritance

The programmer should be aware of the following inherited member functions and variables:

```
class UI_HELP_SYSTEM
{
public:
    UCHAR installed;

    virtual void DisplayHelp(UI_WINDOW_MANAGER *windowManager,
        int helpContext);
};

class UI_HELP_WINDOW_SYSTEM : public UI_HELP_SYSTEM;
```

Global variables

UI_HELP_SYSTEM **_helpSystem* is a pointer to the Zinc Interface Library help system. To use this windowed help system this pointer must be set to the constructed UI_HELP_WINDOW_SYSTEM class object. The external declaration for this variable is contained in **UI_WIN.HPP**. The actual declaration of this variable is contained in **G_HELP.CPP**.

Generating help files

The help context information is read from a binary help file on the disk when needed. This file is created from an ascii text file using the **GENHELP.EXE** utility which is supplied with the Zinc Interface Library.

For example, the text file **NOTEPAD.TXT** below was converted into a binary help file using **GENHELP.EXE**:

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

Press <Esc> to continue...

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

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

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

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

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

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

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

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

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

See also The example file **XHELP.CPP**, which gives a complete example of the UI_HELP_WINDOW_SYSTEM class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the help system.

"Chapter 15—UI_HELP_SYSTEM" of this manual, which describes the base class from which the UI_HELP_WINDOW_SYSTEM class is derived.

UI_HELP_WINDOW_SYSTEM::UI_HELP_WINDOW_SYSTEM

Syntax

```
#include <ui_win.h>
#include "filename.hlh"

UI_HELP_WINDOW_SYSTEM::UI_HELP_WINDOW_SYSTEM(
    char *helpFileName,
    UI_WINDOW_MANAGER *windowManager,
    int defaultHelpContext = NO_HELP_CONTEXT);
```

Remarks This constructor returns a pointer to a new UI_HELP_WINDOW_-SYSTEM class object.

- *helpFileName*$_{in}$ is a pointer to a string containing the name of the binary help file. This file is generated 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.
- *defaultHelpContext*$_{in}$ is the help context to present when no specific help context is available.

Example

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

ExampleFunction1()
{
    // Initialize the Zinc Library Interface.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);

    _errorSystem = new UI_ERROR_WINDOW_SYSTEM;
    _helpSystem = new UI_HELP_WINDOW_SYSTEM("notepad.hlp",
        windowManager, HELP_GENERAL);
    .
    .
    .
    // Restore the basic error and help systems.
    delete _helpSystem;
    _helpSystem = new UI_HELP_SYSTEM;
    delete _errorSystem;
    _errorSystem = new UI_ERROR_SYSTEM;
    .
    .
    .
}
```

UI_HELP_WINDOW_SYSTEM::~UI_HELP_WINDOW_SYSTEM

Syntax

```
#include <ui_win.h>
#include "filename.hlh"

virtual UI_HELP_WINDOW_SYSTEM::
    ~UI_HELP_WINDOW_SYSTEM(void);
```

Remarks

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

Example

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

ExampleFunction1()
{
    // Initialize the Zinc Library Interface.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);

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

```
    // Restore the basic error and help systems.
    delete _helpSystem;
    _helpSystem = new UI_HELP_SYSTEM;
    delete _errorSystem;
    _errorSystem = new UI_ERROR_SYSTEM;
    .
    .
    .
    // Restore the Zinc Library Interface. Order is important!
    delete _helpSystem;
    delete _errorSystem;
    delete windowManager;
    delete eventManager;
    delete display;
}
```

UI_HELP_WINDOW_SYSTEM::DisplayHelp

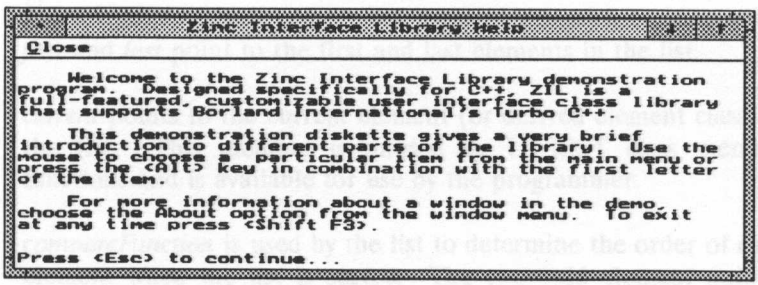
Syntax

#include <ui_win.h>
#include "*filename*.hlh"

virtual void UI_HELP_WINDOW_SYSTEM::
DisplayHelp(UI_WINDOW_MANAGER **windowManager*,
int *helpContext*);

Remarks

This function is used to present help information via the help system. The pictures below show graphic and text implementations of the UI_HELP_WINDOW_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_WINDOW_SYSTEM constructor.

Example

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

ExampleFunction1()
{
    // Initialize the Zinc Library Interface.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);

    _errorSystem = new UI_ERROR_WINDOW_SYSTEM;
    _helpSystem = new UI_HELP_WINDOW_SYSTEM("phonebk.hlp",
        windowManager, HELP_GENERAL);
    .
    .
    .
    // Call the help system to _display general help.
    _helpSystem->DisplayHelp(_windowManager, HELP_PHONE);
    .
    .
    .
}
```

CHAPTER 17 – UI_LIST

Overview The UI_LIST class is used to store doubly-linked list elements derived from the UI_ELEMENT base class. 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 public members of the UI_LIST class (declared in **UI_GEN.HPP**) are:

```
class UI_LIST
{
public:
    UI_ELEMENT *first;
    UI_ELEMENT *last;
    UI_ELEMENT *current;
    int (*compareFunction)(void *element1, void *element2);

    UI_LIST(
        int (*compare)(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);
    void Destroy(void);
    UI_ELEMENT *Get(int index);
    UI_ELEMENT *Get(int (*findFunction)(void *element,
        void *matchData), void *matchData);
    int Index(UI_ELEMENT const *element);
    void Sort(void);
    UI_ELEMENT *Subtract(UI_ELEMENT *element);

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

- *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. This member is unused by UI_LIST class member functions and is available for use by the programmer.
- *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.

See also The example file **XLIST.CPP**, which gives a complete example of the UI_LIST class.

"Chapter 9—UI_ELEMENT" of this manual, which describes the list elements used by the UI_LIST class.

"Chapter 13—UI_EVENT_MANAGER" of this manual, which describes a class that uses the UI_LIST class to store input devices.

"Chapter 24—UI_WINDOW_MANAGER" of this manual, which describes a class that uses the UI_LIST class to store windows.

"Chapter 46—UIW_WINDOW" of this manual, which describes a class that uses the UI_LIST class to store window objects.

UI_LIST::UI_LIST

Syntax

```
#include <ui_gen.hpp>

UI_LIST::UI_LIST(
    int (*compare) (void *element1, void *element2) = NULL);
```

Remarks

This constructor returns a pointer to a new UI_LIST object.

- *compare*$_{in}$ 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. If a negative value is returned, then *element1* is less than *element2*. Otherwise, a positive value indicates that *element1* is greater than *element2*.

Example

```
#include <string.h>      // include for the ITEM class.
#include <stdio.h>       // include for the ITEM class.
#include <ui_gen.hpp>

// Define a derived class from the UI_ELEMENT base class.
class ITEM : public UI_ELEMENT
{
private:
    char *name;

public:
    ITEM(char *a_name)
        { name = strdup(a_name); }
    ~ITEM(void)
        { delete name; }
    static Compare(void *item1, void *item2)
        { return (strcmp(((ITEM *)item1)->name,
        ((ITEM *)item2)->name)); }
    static Find(void *item1, void *matchData)
        { return (strcmp(((ITEM *)item1)->name,
        (char *)matchData)); }
};

ExampleFunction1()
{
    // Each declaration below calls the UI_LIST constructor.
    UI_LIST list1;
    UI_LIST *list2 = new UI_LIST;
    UI_LIST list3(ITEM::Compare);
    UI_LIST *list4 = new UI_LIST(ITEM::Compare);
    .
    .
    .
}
```

UI_LIST::~UI_LIST

Syntax

#include <ui_gen.hpp>

virtual UI_LIST::~UI_LIST(void);

Remarks

This virtual destructor destroys the class information associated with the UI_LIST object. This destructor calls the destructor associated with each element in the list.

Example

```
#include <string.h>      // include for the ITEM class.
#include <stdio.h>  // include for the ITEM class.
#include <ui_gen.hpp>

// Define a derived class from the UI_ELEMENT base class.
class ITEM : public UI_ELEMENT
{
private:
    char *name;

public:
    ITEM(char *a_name)
        { name = strdup(a_name); }
    ~ITEM(void)
        { delete name; }
    static Compare(void *item1, void *item2)
        { return (strcmp(((ITEM *)item1)->name,
        ((ITEM *)item2)->name)); }
    static Find(void *item1, void *matchData)
        { return (strcmp(((ITEM *)item1)->name,
        (char *)matchData)); }
};

ExampleFunction1()
{
    UI_LIST list1;
    UI_LIST *list2 = new UI_LIST;
    UI_LIST list3(ITEM::Compare);
    UI_LIST *list4 = new UI_LIST(ITEM::Compare);
    .
    .
    .
    // 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

Syntax

#include <ui_gen.hpp>

UI_ELEMENT *UI_LIST::Add(UI_ELEMENT **newElement*);
or
UI_ELEMENT *UI_LIST::Add(UI_ELEMENT **element*,
UI_ELEMENT **newElement*);

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, the return value 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 must be called to sort the list when this routine is used.

- *returnValue*$_{out}$ is a pointer to *newElement* if the addition was successful. Otherwise, the return value 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.

Example

```
#include <string.h>      // include for the ITEM class
#include <stdio.h>       // include for the ITEM class
#include <ui_gen.hpp>

// Define a derived class from the UI_ELEMENT base class.
class ITEM : public UI_ELEMENT
{
private:
    char *name;

public:
    ITEM(char *a_name)
        { name = strdup(a_name); }
    ~ITEM(void)
        { delete name; }
    static Compare(void *item1, void *item2)
        { return (strcmp(((ITEM *)item1)->name,
        ((ITEM *)item2)->name)); }
    static Find(void *item1, void *matchData)
        { return (strcmp(((ITEM *)item1)->name,
        (char *)matchData)); }
};
```

```
ExampleFunction1()
{
    // Add elements to a lists.
    UI_LIST list1;
    list1.Add(new UI_ELEMENT);
    list1.Add(new UI_ELEMENT);
    .
    .
    .
}
ExampleFunction2()
{
    // Add items to a list.  The items are automatically added
    // in ascending order.
    UI_LIST *list2 = new LIST(ITEM::Compare);
    list2->Add(new ITEM("Item2");
    list2->Add(new ITEM("Item1");
    .
    .
    .
}
```

UI_LIST::Count

Syntax

```
#include <ui_gen.hpp>

int UI_LIST::Count(void);
```

Remarks

This routine counts the number of elements in the UI_LIST object.

- *returnValue*$_{out}$ is a value containing the number of elements in the list.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    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::Destroy

Syntax

```
#include <ui_gen.hpp>

void UI_LIST::Destroy(void);
```

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.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_LIST list1;
    UI_ELEMENT *element1 = new UI_ELEMENT;
    list1.Add(element1);
    .
    .
    .
    // Destroy all the elements of the list.
    list1.Destroy();
    .
    .
    .
}

ExampleFunction2()
{
    UI_LIST *list2 = new LIST;
    *list2 + new UI_ELEMENT + new UI_ELEMENT + new UI_ELEMENT;
    .
    .
    .
    // 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::Get

Syntax

```
#include <ui_gen.hpp>

UI_ELEMENT *UI_LIST::Get(int index);
    or
UI_ELEMENT *UI_LIST::Get(
    int (*findFunction)(void *element, void *matchData),
    void *matchData);
```

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 *findFunction*. This method eliminates the need for the programmer to set up a loop to access a specific element in the list but where the element's index is unknown.

- *returnValue*$_{out}$ is a pointer to the matching list element. This value is NULL if no element matches *matchData*.

- *findFunction*$_{in}$ is a pointer to a programmer supplied function that compares a specified element with the typecast *matchData*. If an exact match is made this function must return a 0. Any non-zero value indicates 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 **UI_LIST::Get** routine will call the *findFunction* routine with this argument as the *matchData* parameter.

Example

```
#include <string.h>     // include for the ITEM class
#include <stdio.h>  // include for the ITEM class
#include <ui_gen.hpp>

// Define a derived class from the UI_ELEMENT base class.
class ITEM : public UI_ELEMENT
{
private:
    char *name;

public:
    ITEM(char *a_name)
        { name = strdup(a_name); }
    ~ITEM(void)
        { delete name; }
    static Compare(void *item1, void *item2)
        { return (strcmp(((ITEM *)item1)->name,
        ((ITEM *)item2)->name)); }
    static Find(void *item1, void *matchData)
        { return (strcmp(((ITEM *)item1)->name,
        (char *)matchData)); }
};

ExampleFunction1()
{
    UI_LIST list;
    list + new ITEM("Item1") + new ITEM("Item2");
    .
    .
    .
    // Get the 2nd element in the list.
    UI_ELEMENT *element = list.Get(2);
    // Get the element that matches the "Item2" pattern.
    ITEM *item = (ITEM *)list.Get(ITEM::Find, "Item2");
    .
    .
    .
}

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 UI_LIST::Index(UI_ELEMENT const **element*);

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, a -1 is returned.
- *element*$_{in}$ is a pointer to the list element to find. This element must either be UI_ELEMENT or derived from the UI_ELEMENT class.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    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.";
}

ExampleFunction2()
{
    UI_LIST list;
    UI_ELEMENT *element1 = new UI_ELEMENT;
    .
    .
    .
    // See if element1 is in the list.
    if (list.Index(element1) != -1)
        cout << "Element1 was found in the list.";
    else
        cout << "Element1 was NOT found in the list.";
}
```

UI_LIST::Sort

Syntax #include <ui_gen.hpp>

void UI_LIST::Sort(void);

Remarks This routine sorts the UI_LIST object according to the class compare function (specified in the class constructor). If the list has no compare function, no sort occurs.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_LIST list;
    ITEM *item3 = new ITEM("Item3");
    ITEM *item1 = new ITEM("Item1");
    ITEM *item2 = new ITEM("Item2");
    list + item3 + item1 + item2;
    .
    .
    .
    list.compareFunction = ITEM::Compare;
    // Sort a list of items.
    list.Sort();
}

ExampleFunction2()
{
    UI_LIST *list = new UI_LIST(ITEM::CompareAscending);
    list
        + new ITEM("Item3")
        + new ITEM("Item2")
        + new ITEM("Item1");
    .
    .
    .
    // Sort the list according to a descending order.
    list->compareFunctoin = ITEM::CompareDecending;
    list->Sort();
}
```

UI_LIST::Subtract

Syntax #include <ui_gen.hpp>

UI_ELEMENT *UI_LIST::Subtract(UI_ELEMENT **element*);

Remarks This function removes an element from the UI_LIST object. 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 deleted element.
- *element*$_{in}$ is a pointer to the element to be deleted from the list. The *element* class destructor is not called by **UI_LIST::Subtract**.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    // Construct a list, then add elements to it.
    UI_LIST list1;
    UI_ELEMENT *element1 = new UI_ELEMENT;
    list1.Add(element1);
    .
    .
    .
    // Delete a particular element from a list.
    list1.Delete(element1);
    .
    .
    .
}

ExampleFunction2()
{
    // Construct a list, then add elements to it using the
    // + operator overload.
    UI_LIST *list2 = new LIST;
    *list2 + new UI_ELEMENT + new UI_ELEMENT + new UI_ELEMENT;
    .
    .
    .
    // Manually delete each element in the list.
    UI_ELEMENT *element = list2->head;
    while (element)
        element = list2->Delete(element);
    .
    .
    .
}
```

UI_LIST::operator +

Syntax #include <ui_gen.hpp>

UI_LIST &UI_LIST::operator + (UI_ELEMENT **element*);

Remarks This overload 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 the object derived from the UI_ELEMENT base class that is to be added to the list.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_LIST list;
    UI_ELEMENT *element1 = new UI_ELEMENT;
    UI_ELEMENT *element2 = new UI_ELEMENT;
    // Add elements to the list using their pointers.
    list + element1 + element2;
}

ExampleFunction2()
{
    UI_LIST *list = new UI_LIST;
    // Add constructed elements directly to the list.
    *list + new UI_ELEMENT;
    *list + new UI_ELEMENT + new UI_ELEMENT;
    .
    .
    .
}
```

UI_LIST::operator –

Syntax #include <ui_gen.hpp>

UI_LIST &UI_LIST::operator – (UI_ELEMENT **element*);

Remarks This overload operator deletes an element from the UI_LIST object. This operator overload is equivalent to calling the **UI_LIST::Delete** routine, except that it allows the chaining of list element deletions from the UI_LIST object.

- *returnValue*$_{out}$ is the UI_LIST reference. Returning the reference to the list allows chaining of the **UI_LIST::operator–** overload operator.

- *element*$_{in}$ is a pointer to the UI_ELEMENT class element or the object derived from the UI_ELEMENT base class that is to be removed from the list.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_LIST list;
    UI_ELEMENT element[10];
    for (int i = 0; i < 10; i++)
        list + element[i];  // Add the element to the list.
    .
    .
    .
    // Remove the array of elements from the list.
    for (i = 0; i < 10; i++)
        list - element[i];  // Remove the element from the list.
}

ExampleFunction2()
{
    UI_LIST *list = new UI_LIST;
    *list + new UI_ELEMENT;
    *list + new UI_ELEMENT + new UI_ELEMENT;
    .
    .
    .
    // Manually remove all the elements from a list.
    UI_ELEMENT t_element;
    for (UI_ELEMENT element = list->head; element; element =
    t_element)
    {
        t_element = element->next;
        list - element;  // Remove the element from the list.
        delete element;
    }
}
```

CHAPTER 18 – UI_MS_MOUSE

Overview

The UI_MS_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. The public members of the UI_MS_MOUSE class (declared in **UI_EVT.HPP**) are:

```
class UI_MS_MOUSE : public UI_DEVICE
{
public:
    UI_MS_MOUSE(USHORT initialState = D_VIEW);
    virtual ~UI_MS_MOUSE(void);
};
```

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 UI_EVENT
{
    int type;            // The type of event (E_MOUSE).
    USHORT rawCode;      // The mouse's shift and pressed state.
    union
    {
        UI_KEY key;
        UI_REGION region;
        UI_POSITION position;   // The mouse position.
        void *data;
    };
};
```

- *type* is the event type. The mouse device always generates an E_MOUSE type event.
- *rawCode* is the keyboard's shift state and mouse's button states. The raw code may be one or more of the following flags (declared in **UI_EVT.HPP**):

  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 then 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, then 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 then 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_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.

- *position.column* is 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 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.

See also The example file **XMSMOUSE.CPP**, which gives a complete example of the UI_MS_MOUSE class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the operation of device classes within the event manager.

"Chapter 1—UI_BIOS_KEYBOARD" of this manual, which describes an additional device derived from the UI_DEVICE class.

"Chapter 2—UI_CURSOR" of this manual, which describes an additional device derived from the UI_DEVICE class.

"Chapter 4—UI_DEVICE" of this manual, which describes the base class from which the UI_MS_MOUSE class is derived.

"Chapter 13—UI_EVENT_MANAGER" of this manual, which describes the operation (e.g., addition, subtraction, state change) of device classes within the event manager.

UI_MS_MOUSE::UI_MS_MOUSE

Syntax

```
#include <ui_evt.hpp>

UI_MS_MOUSE::UI_MS_MOUSE(
    USHORT initialState = DM_VIEW);
```

Remarks This constructor returns a pointer to a new UI_MS_MOUSE class object. It should be called after the following class constructors have been called:

1—UI_DOS_BGI_DISPLAY *or* UI_DOS_TEXT_DISPLAY, then

2—UI_EVENT_MANAGER

NOTE: If the mouse device is attached to the event manager, it will automatically be destroyed when the event manager is destroyed.

- *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.)

NOTE: The UI_MS_MOUSE constructor automatically activates the mouse cursor. This cursor is not affected by the state of the device. The only way to remove a mouse cursor is by hiding it (using the **UI_EVENT_MANAGER::DevicesHide** function or by calling its destructor).

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    UI_EVENT_MANAGER eventManager(100, &display);

    // Initialize the mouse.
    UI_MS_MOUSE *mouse = new UI_MS_MOUSE;
    eventManager + mouse;
    .
    .
    .
}

ExampleFunction2()
{
    // Attach the devices directly to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager
        + new UI_BIOS_KEYBOARD
        + new UI_MS_MOUSE
        + new UI_CURSOR;
    .
    .
    .
}
```

UI_MS_MOUSE::~UI_MS_MOUSE

Syntax

```
#include <ui_evt.hpp>

virtual UI_MS_MOUSE::~UI_MS_MOUSE(void);
```

Remarks

This virtual destructor destroys the class information associated with the UI_MS_MOUSE object and closes the interrupt associated with the mouse. Care should be taken to only destroy a mouse device that is not attached to the event manager.

Example

```
#include <ui_evt.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    UI_EVENT_MANAGER eventManager(100, &display);

    // Initialize the mouse.
    UI_MS_MOUSE *mouse = new UI_MS_MOUSE;
    eventManager + mouse;
    .
    .
    .
    // Remove the mouse from the event manager, then call its
    // destructor.  We could also have left the mouse alone.  Its
    // destructor would automatically be called when the event
    // manager destructor was called.
    eventManager - mouse;
    delete mouse;
}

ExampleFunction2()
{
    // Attach the devices directly to the event manager.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    *eventManager
        + new UI_BIOS_KEYBOARD
        + new UI_MS_MOUSE
        + new UI_CURSOR;
    .
    .
    .
    // This automatically calls the cursor destructor.
    delete eventManager;
    delete display;
}
```

CHAPTER 19 – UI_PALETTE

Overview The UI_PALETTE structure is used by the Zinc Interface Library class objects for color information. The UI_PALETTE structure (declared in **UI_EVT.HPP**) has the following fields:

```
#define attrib(foreground, background) \
    (((background) << 4) + (foreground))

struct UI_PALETTE
{
    /* Text mode */
    UCHAR fillCharacter;
    UCHAR colorAttribute;
    UCHAR monoAttribute;

    /* Graphics mode */
    UCHAR fillPattern;
    UCHAR color;
    UCHAR bwColor;
    UCHAR grayScaleColor;
};
```

- *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* are the attributes of the foreground and background colors respectively for color text display mode.
- *monoAttributes* are the attributes 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.
- *color* are the attributes of the foreground and background colors respectively for VGA, VGA monochrome and EGA graphics display modes.
- *bwColor* are the attributes of the foreground and background colors respectively for CGA and Hercules graphics display modes.
- *grayScaleColor* are the attributes of the foreground and background colors respectively for EGA monochrome graphics display mode.

Global variables

UI_PALETTE *_*backgroundPalette* is a pointer to the background palette. The external declaration for this variable is contained in **UI_DSP.HPP**. The actual declaration of this variable is contained in **G_PBACK.CPP**.

See also

The example file **XPALETTE.CPP**, which gives a complete example of the UI_PALETTE structure.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the use of events within the Zinc Interface Library.

"Chapter 6—UI_DOS_BGI_DISPLAY" of this manual, which is a class that uses the UI_PALETTE structure in many of its member functions.

"Chapter 6—UI_DOS_TEXT_DISPLAY" of this manual, which is a class that uses the UI_PALETTE structure in many of its member functions.

CHAPTER 20 – UI_PALETTE_MAP

Overview

The UI_PALETTE_MAP structure is used by the Zinc Interface Library class objects for color map information. The UI_PALETTE_MAP structure (declared in **UI_MAP.HPP**) has the following fields:

```
typedef struct UI_PALETTE_MAP_STRUCT
{
    int windowID;
    int logicalValue;
    UI_PALETTE palette;
} UI_PALETTE_MAP;
```

Global variables

UI_PALETTE_MAP **_errorPaletteMapTable* is a pointer to the error palette table. This is the palette table used by the error system. The external declaration for this variable is contained in **UI_MAP.HPP**. The actual declaration of this variable is contained in **G_PERROR.CPP**.

UI_PALETTE_MAP **_helpPaletteMapTable* is a pointer to the help palette table. This is the palette table used by the help system. The external declaration for this variable is contained in **UI_MAP.HPP**. The actual declaration of this variable is contained in **G_PHELP.CPP**.

UI_PALETTE_MAP **_normalPaletteMapTable* is a pointer to the normal palette table. This is the default palette table used by all window objects. The external declaration for this variable is contained in **UI_MAP.HPP**. The actual declaration of this variable is contained in **G_PNORM.CPP**.

See also

The example file **XPALETTE.CPP**, which gives a complete example of the UI_PALETTE_MAP structure.

"Chapter 5—Default Palette Mapping" of the Programmer's Guide, which describes the default palette map table used by the Zinc Interface Library.

CHAPTER 21 – UI_POSITION

Overview

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 (declared in **UI_GEN.HPP**) has the following fields:

```
struct UI_POSITION
{
    int column;
    int line;
};
```

- *column* is a vertical position indicator.
- *line* is a horizontal position indicator.

See also

"Chapter 12—UI_EVENT" of this manual, which describes a higher-level structure that uses the UI_POSITION structure in a union field.

"Chapter 12—UI_MS_MOUSE" of this manual, which describes a class object that uses the UI_POSITION structure to indicate its position on the screen.

CHAPTER 22 – UI_REGION

Overview The UI_REGION structure is used to store rectangular information (e.g., window object regions) int the library. The UI_REGION structure (declared in **UI_GEN.HPP**) has the following fields:

```
struct UI_REGION
{
    int left;
    int top;
    int right;
    int bottom;
};
```

- *left* and *top* is the starting position of the region.
- *right* and *bottom* is the ending position of the region.

See also "Chapter 12—UI_EVENT" of this manual, which describes a higher-level structure that uses the UI_REGION structure in a union field.

"Chapter 6—UI_DOS_BGI_DISPLAY" of this manual, which describes a class object that receives UI_REGION structures in many of its member functions.

"Chapter 7—UI_DOS_TEXT_DISPLAY" of this manual, which describes a class object that receives UI_REGION structures in many of its member functions.

"Chapter 6—UI_WINDOW_OBJECT" of this manual, which describes a class object that has UI_REGION member variables.

CHAPTER 23 – UI_TIME

Overview 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 44—UIW_TIME" of this manual for information about the time window object.) The public members of the UI_TIME class (declared on **UI_GEN.HPP**) are:

```
class UI_TIME
{
public:
    UI_TIME(void);
    UI_TIME(int packedTime);
    UI_TIME(int hour, int minute, int second = 0,
        int hundredth = 0);
    UI_TIME(const char *string, USHORT tmFlags = TMF_NO_FLAGS);
    virtual ~UI_TIME(void);

    void Export(void);
    void Export(int packedTime);
    void Export(int *hour, int *minute, int *second,
        int *hundredth);
    void Export(char *string, USHORT tmFlags);
    void Import(void);
    int Import(int packedTime);
    int Import(int hour, int minute, int second = 0,
        int hundredth = 0);
    int Import(const char *string, USHORT tmFlags);
    void NamesSet(char *am = NULL, char *pm = NULL);

    int operator > (UI_TIME& rightOperand);
    int operator < (UI_TIME& rightOperand);
    int operator == (UI_TIME& rightOperand);
};
```

See also The example file **XTIME.CPP**, which gives a complete example of the UI_TIME class.

"Chapter 3—UI_DATE" of this manual, which describes a similar low-level class that maintains date information.

"Chapter 44—UIW_TIME" of this manual, which describes a high-level window object that uses the UI_TIME class to store display information.

UI_TIME::UI_TIME

Syntax

```
#include <ui_gen.hpp>

UI_TIME(void);
    or
UI_TIME(int packedTime);
    or
UI_TIME(int hour, int minute, int second, int hundredth = 0);
    or
UI_TIME(const char *string,
    USHORT tmFlags = TMF_NO_FLAGS);
```

Remarks

These overloaded functions 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 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).

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 dependant information (supplied by all DOS based systems):

  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.

  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 the minutes and "15" would be interpreted as the 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 a seconds and the value "15" would be interpreted as hundredths of seconds.

  TMF_SYSTEM—Fills a blank time with the system time. For example, if a blank ascii time value were specified by the programmer and the TMF_SYSTEM flag were set, then the time would be set to the current system time (e.g., "1:10pm").

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_TIME time1;                          // System date initialization.
    UI_TIME time2(12, 0, 0);                // Integer initialization.
    UI_TIME *time3 =                        // String initialization.
      new UI_TIME("12:00:00pm");
    .
    .
    .
}
```

UI_TIME::~UI_TIME

Syntax

#include <ui_gen.hpp>

virtual UI_TIME::~UI_TIME(void);

Remarks

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

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_TIME time1;                  // System date initialization.
    UI_TIME time2(12, 0, 0);        // Integer initialization.
    UI_TIME *time3 =                // String initialization.
        new UI_TIME("12:00:00pm");
    .
    .
    .
    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 UI_TIME::Export(void);

or

void UI_TIME::Export(int *packedTime*);

or

void UI_TIME::Export(int **hour*, int **minute*, int **second*, int **hundredth* = 0);

or

void UI_TIME::Export(char **string*, USHORT *tmFlags*);

Remarks

The first overloaded function sets the system time according to the UI_TIME object's time information. This function only works on environments where time information can be set.

The second overloaded function returns time information through a single packed integer argument.

- *packedTime*$_{in/out}$ is a packed representation of the time (whose format is the same as 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).

The third 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 59.

The fourth 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 dependant information (supplied by all DOS based systems):

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

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

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

  TMF_NO_FLAGS—Does not associate any special flags with the format function. In this case, the ascii time is formatted using the default country information. This flag should not be used in conjunction with any other TMF flag.

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

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

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

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

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

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

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_TIME time;       // Use a system time.

    // Print out the time in various forms.
    int packedTime;
    time.Export(&packedTime);
    printf("Packed time value: %x\n", packedTime);

    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, 128, 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>

int UI_TIME::Import(void);
or
int UI_TIME::Import(int *packedTime*);
or
int UI_TIME::Import(int *hour*, int *minute*, int *second*,
int *hundredth* = 0);
or
int UI_TIME::Import(const char **string*, USHORT *tmFlags*);

Remarks

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

The second overloaded function sets the time information through a single packed integer argument.

- *returnValue*$_{out}$ is 0 if the packed time value was successfully imported. Otherwise, the return value is -1.
- *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).

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

- *returnValue*$_{out}$ is 0 (TMI_OK) if the time values were successfully imported. Otherwise, the return value is TMI_INVALID.
- *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 0 (TMI_OK) if the ascii string parsed to a valid date. Otherwise, one of the following error codes (declared in **UI_GEN.HPP**) is returned:

  TMI_INVALID—The hour, minute, second or hundredths value was out of range.

  TMI_VALUE_MISSING—The time string is blank and the TMF_SYSTEM flag is not set.

- *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 dependant information (supplied by all DOS based systems):

  TMF_NO_FLAGS—Does not associate any special flags with the UI_TIME class object. In this case, the ascii time is interpreted using the default country information. This flag should not be used in conjunction with any other TMF flag.

  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 the minutes and "15" would be interpreted as the 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 the seconds and the value "15" would be interpreted as the hundredths of seconds.

  TM_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 (e.g., "1:10pm").

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_TIME time;   // Initialize a system time.

    // Import the time in various forms, then print out
    // the results.
    char asciiTime[128];
    time.Import(1990, 1, 1);
    time.Export(asciiTime, DTF_NO_FLAGS);
    printf("Ascii time value: %s\n", asciiTime);

    time.Import("1-1-1990");
    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::NamesSet

Syntax

```
#include <ui_gen.hpp>

void UI_TIME::NamesSet(const char *am = NULL,
    const char *pm = NULL);
```

Remarks This static function is used to re-define the ascii ante- and post-meridian name values. The default ante- and post-meridian values are: "a.m." and "p.m."

- am_{in} is a pointer to the new ante-meridian value. This value must be allocated by the programmer and remain active throughout program execution. If this argument is NULL, *am* is reset to point to the default ante-meridian value (i.e., "a.m.").

- pm_{in} is a pointer to the new post-meridian value. This value must be allocated by the programmer and remain active throughout program execution. If this argument is NULL, *pm* is reset to point to the default post-meridian value (i.e., "a.m.").

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    // Redefine the time name values.
    UI_TIME::NamesSet("AM", "PM");
    .
    .
    .
    // Restore the time name values.
    UI_TIME::NamesSet(NULL, NULL);
}
```

UI_TIME::operator >

Syntax

```
#include <ui_gen.hpp>

int UI_TIME::operator > (UI_TIME& rightOperand)
```

Remarks This operator overload determines whether the UI_TIME object is chronologically greater than the time specified by *rightOperand*.

- *returnValue*$_{out}$ is TRUE if the UI_TIME object is chronologically greater than *rightOperand*. Otherwise, the return value is FALSE.

- *rightOperand*$_{in}$ is the UI_TIME object against which to compare.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_TIME currentTime; // Initialize a system time class.
    UI_TIME beginLunch("12:00 pm");
    UI_TIME endLunch("1:00 pm");

    // Check the times.
    if (currentTime < beginLunch)
        printf("It's morning.\n");
    else if (currentTime == endLunch || currentTime > endLunch)
        printf("It's afternoon.\n");
    else
        printf("Gone to lunch.\n")
}
```

UI_TIME::operator <

Syntax

#include <ui_gen.hpp>

int UI_TIME::operator < (UI_TIME& *rightOperand*)

Remarks

This operator overload determines whether the UI_TIME object is chronologically less than the time specified by *rightOperand*.

- *returnValue*$_{out}$ is TRUE if the UI_TIME object is chronologically less than *rightOperand*. Otherwise, the return value is FALSE.

- *rightOperand*$_{in}$ is the UI_TIME object against which to compare.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_TIME currentTime; // Initialize a system time class.
    UI_TIME beginLunch("12:00 pm");
    UI_TIME endLunch("1:00 pm");
```

```
    // Check the times.
    if (currentTime < beginLunch)
        printf("It's morning.\n");
    else if (currentTime == endLunch || currentTime > endLunch)
        printf("It's afternoon.\n");
    else
        printf("Gone to lunch.\n")
}
```

UI_TIME::operator ==

Syntax

```
#include <ui_gen.hpp>

int UI_TIME::operator == (UI_TIME& rightOperand)
```

Remarks

This operator overload determines whether the UI_TIME object is chronologically equal to the time specified by *rightOperand*.

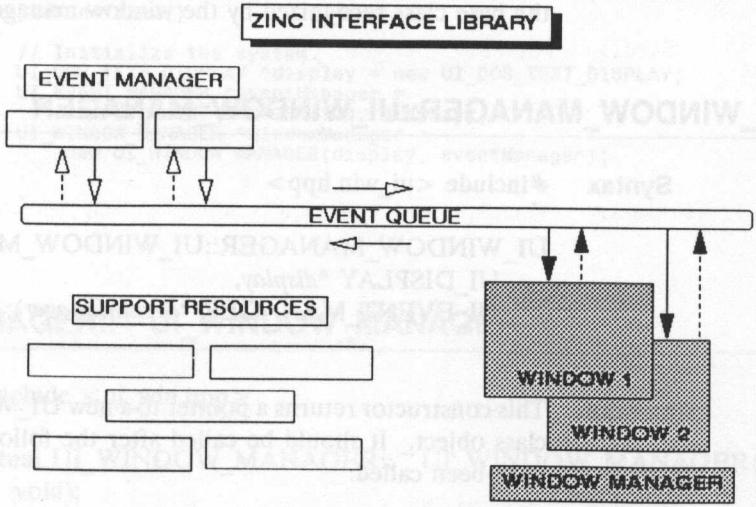
- *returnValue*$_{out}$ is TRUE if the UI_TIME object is chronologically equal to *rightOperand*. Otherwise, the return value is FALSE.
- *rightOperand*$_{in}$ is the UI_TIME object against which to compare.

Example

```
#include <ui_gen.hpp>

ExampleFunction1()
{
    UI_TIME currentTime; // Initialize a system time class.
    UI_TIME beginLunch("12:00 pm");
    UI_TIME endLunch("1:00 pm");

    // Check the times.
    if (currentTime < beginLunch)
        printf("It's morning.\n");
    else if (currentTime == endLunch || currentTime > endLunch)
        printf("It's afternoon.\n");
    else
        printf("Gone to lunch.\n")
}
```

CHAPTER 24 – UI_WINDOW_MANAGER

Overview 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 position on the screen.

The public members of the UI_WINDOW_MANAGER class (declared in **UI_WIN.HPP**) are:

```
class UI_WINDOW_MANAGER
{
public:
    UI_WINDOW_MANAGER(UI_DISPLAY *display,
        UI_EVENT_MANAGER *eventManager);
    virtual ~UI_WINDOW_MANAGER(void);

    void Add(UI_WINDOW_OBJECT *object);
    int Event(const UI_EVENT &event);
    void Subtract(UI_WINDOW_OBJECT *object);

    UI_WINDOW_MANAGER &operator + (void *object);
    UI_WINDOW_MANAGER &operator - (void *object);
};
```

See also The example file **XWINMGR.CPP**, which gives a complete example of the UI_WINDOW_MANAGER class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which describes the operation of the window manager in the Zinc Interface Library.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes the base class recognized by the window manager.

UI_WINDOW_MANAGER::UI_WINDOW_MANAGER

Syntax

```
#include <ui_win.hpp>

UI_WINDOW_MANAGER::UI_WINDOW_MANAGER(
    UI_DISPLAY *display,
    UI_EVENT_MANAGER *eventManager);
```

Remarks This constructor returns a pointer to a new UI_WINDOW_MANAGER class object. It should be called after the following class constructors have been called:

1—UI_DOS_BGI_DISPLAY *or* UI_DOS_TEXT_DISPLAY, then

2—UI_EVENT_MANAGER

- *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 to send private communication to the window 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 '|').

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Initialize the system.
    UI_DOS_TEXT_DISPLAY display;
    UI_EVENT_MANAGER eventManager(100, display);
    UI_WINDOW_MANAGER windowManager(&display, &eventManager);
    .
    .
    .
}

ExampleFunction2()
{
    // Initialize the system.
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
}
```

UI_WINDOW_MANAGER::~UI_WINDOW_MANAGER

Syntax

```
#include <ui_win.hpp>

virtual UI_WINDOW_MANAGER::~UI_WINDOW_MANAGER(
    void);
```

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>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY display;
    UI_EVENT_MANAGER eventManager(100, display);
    UI_WINDOW_MANAGER windowManager(&display, &eventManager);
    .
    .
    .
    // The windowManager, eventManager and display are
    // automatically destroyed when the scope of this
    // routine ends.
}
```

```
ExampleFunction2()
{
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    // Restore the system.  Managers need to be destroyed
    // in opposite order.
    delete windowManager;
    delete eventManager;
    delete display;
}
```

UI_WINDOW_MANAGER::Add

Syntax

```
#include <ui_win.hpp>

void UI_WINDOW_MANAGER::Add(
    UI_WINDOW_OBJECT *object);
```

Remarks

This routine adds a new window object to the window manager. The window object must be a class derived from the UI_WINDOW_OBJECT base class. This routine adds the window object to the front of the window manager's list of objects. This means all new events will be sent to this object until an event is received that changes the window object flow. Once a window object is attached to the window manager, it is created and painted to the screen. It is also available to receive input information when the **UI_WINDOW_MANAGER::Event** function is called. The last object to be attached using the **UI_WINDOW_MANAGER::Add** routine becomes the current window until an operation is performed that changes its position (e.g., a mouse click on a different window).

- *object*$_{in}$ is a pointer to the new UI_WINDOW_OBJECT class object to be added to the window manager's window list.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    // Add a simple window to the screen.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    windowManager->Add(window);
    .
    .
    .
}
```

UI_WINDOW_MANAGER::Event

Syntax #include <ui_win.hpp>

int UI_WINDOW_MANAGER::Event(const UI_EVENT *&event*);

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 and object the event should go. For example, the keyboard event E_KEY is processed to the current window object. System events, however, may be processed in different manners.

- *returnValue*$_{out}$ is the type of action that the event 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 significant meaning. The following return codes (declared in **UI_MAP.HPP** and **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_HELP—If this message is sent by the programmer to the window manager, help will be displayed about the application program. Otherwise, this message is received by the programmer, indicating help has been presented to the end-user.

S_CANCEL—The window manager received an event by the end-user that was a request to cancel the information in the current window. The current window can be obtained by calling the **UI_WINDOW_MANAGER::First** function.

S_CONTINUE—A message sent by a programmer specified procedure that requires a lot of processor time but wants to check the status of the event queue or give time to other device objects. If this message is sent to the event manager, the next event received by the window object will be the S_CONTINUE message.

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. This message is sent back to the programmer whenever the message is object specific and no object is attached to the window manager.

S_REDISPLAY—Re-displays the screen display. Sending this message causes the window manager to clear the screen display and repaint all the windows attached to the display.

S_UNKNOWN—The event passed to the window manager was not recognized by the window manager or 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>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    // Get events until the L_EXIT logical key is entered.
    UI_EVENT event;
    do
    {
        eventManager->Get(event, Q_NORMAL);
        event.type = windowManager->Event(event);
    } while (event.type != L_EXIT);
    .
    .
    .
}
```

UI_WINDOW_MANAGER::Subtract

Syntax

```
#include <ui_win.hpp>

void UI_WINDOW_MANAGER::Subtract(
    UI_WINDOW_OBJECT *object);
```

Remarks

This routine removes a window object from the window manager. Once a window object is removed, it is removed from the screen display and will not receive event information. This routine does not call the destructor associated with the window object.

NOTE: If a window object (e.g., UIW_WINDOW, UIW_PULL_-DOWN_MENU) is attached to the window manager, it is automatically subtracted and destroyed when the window manager is destroyed.

- *object*$_{in}$ is a pointer to the window object to be removed from the window manager's list of window objects. Once the object is removed, it is no longer displayed to the screen and does not receive event information.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    // Add a simple window to the screen.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    windowManager->Add(window);
    .
    .
    .
    // Clean up the system.
    windowManager->Subtract(window);
    delete windowManager;
    delete eventManager;
    delete display;
}
```

UI_WINDOW_MANAGER::operator +

Syntax

```
#include <ui_win.hpp>

UI_WINDOW_MANAGER &UI_WINDOW_MANAGER::
    operator + (UI_WINDOW_OBJECT *object);
```

Remarks

This overload operator adds a window object to the window manager. Using this operator overload is equivalent to calling the **UI_WINDOW_-MANAGER::Add** function, except that it allows the chaining of window elements to the window manager. If the window object is already attached to the window manager, calling this operator overload causes the window object to recompute and redisplay its information to the screen display.

- *returnValue*$_{out}$ is the UI_WINDOW_MANAGER reference. Returning the reference to the window manager allows chaining of the **UI_WINDOW_MANAGER::operator+** overload operator.
- *object*$_{in}$ is a pointer to the UI_WINDOW_OBJECT to be added to the window manager's list of window objects.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    // Add a simple window to the screen.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);

    *windowManager + window;
    .
    .
    .
}
```

UI_WINDOW_MANAGER::operator –

Syntax

```
#include <ui_win.hpp>

UI_WINDOW_MANAGER &UI_WINDOW_MANAGER::
    operator – (UI_WINDOW_OBJECT *object);
```

Remarks This overload operator removes a window object from the window manager. Using this operator overload is equivalent to calling the **UI_WINDOW_MANAGER::Subtract** function except that it allows the chaining of window objects from the window manager.

- *returnValue*$_{out}$ is the UI_WINDOW_MANAGER reference. Returning the reference to the window manager allows chaining of the **UI_WINDOW_MANAGER::operator–** overload operator.

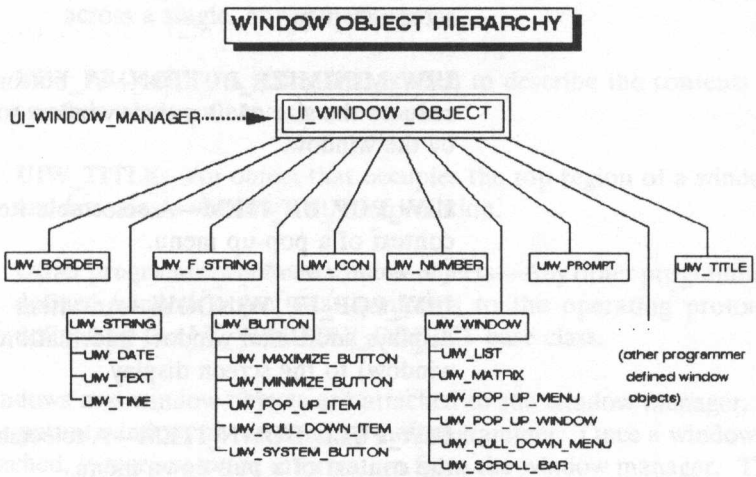
- *object*$_{in}$ is a pointer to the window object to be removed from the window manager's list of window objects.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    UI_DOS_TEXT_DISPLAY *display = new UI_DOS_TEXT_DISPLAY;
    UI_EVENT_MANAGER *eventManager =
        new UI_EVENT_MANAGER(100, display);
    UI_WINDOW_MANAGER *windowManager =
        new UI_WINDOW_MANAGER(display, eventManager);
    .
    .
    .
    // Add a simple window to the screen.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);

    *windowManager + window;
    .
    .
    .
    *windowManager - window;

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

CHAPTER 25 – UI_WINDOW_OBJECT

Overview The UI_WINDOW_OBJECT class is an abstract class that defines basic information associated with window objects (e.g., borders, buttons, menus). Since the UI_WINDOW_OBJECT class is abstract, it cannot be used as a constructed class. Rather, derived classes, such as UIW_BORDER, UIW_BUTTON, 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_STRING—A field used to enter, display, or modify an ascii string buffer.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Windows and window objects are attached to the window manager, or to a parent window, at run-time by the programmer. Once a window is attached, it receives event information from the window manager. The public members of the UI_WINDOW_OBJECT (defined in **UI_WIN.HPP**) are:

```
class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};
```

- *eventMapTable* is a pointer to the event map table associated with the object. This table initially points to the *_eventMapTable* global

palette map table. This pointer can be reset by the programmer. (For more information about the *_eventMapTable* variable see "Chapter 14—UI_EVENT_MAP" of this manual.)

- *defaultDepth* is the depth (for 3-dimensional appearance) of UI_WINDOW_OBJECT class objects.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of 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 positions on the first character of the window field (from another window field) then presses a key (without having previously pressed any movement keys.)

  WOF_BORDER—Draws a single line border around the window object. If the application program is running in text mode and the field occupies 1 line of the screen display, setting this option draws display braces (i.e., '[' ']') around the object.

  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 re-setting the 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_NO_ALLOCATE_DATA—Prevents the window object from allocating a data space to store the window object's information. If this flag is set, the programmer must allocate the data passed to the window object.

  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—The window object is not a form field. If this flag is set the window object will occupy all the remaining space of its parent window.

WOF_NO_INVALID—Prevents the "Leave invalid" option from being selectable by the end-user when incomplete or invalid window object information is entered.

WOF_NO_UNANSWERED—Prevents the "Leave unanswered" option from being selectable by the end-user when incomplete or invalid window object information is entered.

WOF_NON_SELECTABLE—Prevents the window object from being selected. If this flag is set, the user will not be able to position on the window object.

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—The window object cannot be edited. If this flag is set, the end-user will not be able to change the window object's information but will be able to browse through the information.

- *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. 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_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_NO_STATUS—No status flags are associated with the window object at the current time.

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 non-selected state.

WOS_UNANSWERED—The window object's data is in an "unanswered" state.

- *true* is the true coordinate of the object on the screen. 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.
- *parent* is a pointer to the window object's parent. For example, if a high-level window were created and contained several child objects then the parent object for the children would be the window.
- *display* is a pointer to the associated screen display. This pointer is set by the window manager when the object is attached to the screen display.
- *eventManager* is a pointer to the associated window object's event manager. This pointer is set by the window manager when the object is attached to the screen display.
- *windowManager* is a pointer to the associated window manager. This pointer is set by the window manager when the object is attached to the screen display.
- *paletteMapTable* is a pointer to a palette map table. This palette table is used to determine the color combinations used to display a window object.

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 the Zinc Interface Library (see the references below).

See also "Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 26—UIW_BORDER" of this manual which describes a window object derived from the UI_WINDOW_OBJECT class.

"Chapter 27—UIW_BUTTON" of this manual which describes a window object derived from the UI_WINDOW_OBJECT class.

"Chapter 29—UIW_FORMATTED_STRING" of this manual which describes a window object derived from the UI_WINDOW_OBJECT class.

"Chapter 30—UIW_ICON" of this manual which describes a window object derived from the UI_WINDOW_OBJECT class.

"Chapter 34—UIW_NUMBER" of this manual which describes a window object derived from the UI_WINDOW_OBJECT class.

"Chapter 38—UIW_PROMPT" of this manual which describes a window object derived from the UI_WINDOW_OBJECT class.

"Chapter 41—UIW_STRING" of this manual which describes a window object derived from the UI_WINDOW_OBJECT class.

"Chapter 45—UIW_TITLE" of this manual which describes a window object derived from the UI_WINDOW_OBJECT class.

"Chapter 46—UIW_WINDOW" of this manual which describes a window object derived from the UI_WINDOW_OBJECT class.

UI_WINDOW_OBJECT::Next

Syntax

```
#include <ui_win.hpp>

UI_WINDOW_OBJECT *UI_WINDOW_OBJECT::Next(void);
```

Remarks This advanced function returns a pointer to the next window object in the parent window's list of window objects.

- *returnValue*$_{out}$ is a pointer to the next window object in the parent window's list of window objects.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a new window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
    .
    .
    .
    // Find out if any fields are invalid.
    int invalidFields = FALSE;
    for (UI_WINDOW_OBJECT *object = window->First();
        object; object = object->Next())
        if (FlagSet(object->woStatus, WOS_INVALID))
        {
            invalidFields = TRUE;
            break;
        }
    .
    .
    .
}
```

UI_WINDOW_OBJECT::Previous

Syntax

```
#include <ui_win.hpp>

UI_WINDOW_OBJECT *UI_WINDOW_OBJECT::Previous(void);
```

Remarks This advanced function returns a pointer to the previous window object in the parent window's list of window objects.

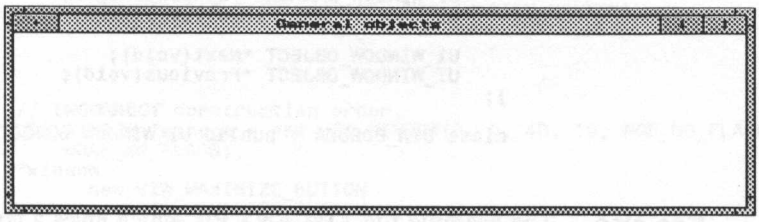
- *returnValue*$_{out}$ is a pointer to the previous window object in the parent window's list of window objects.

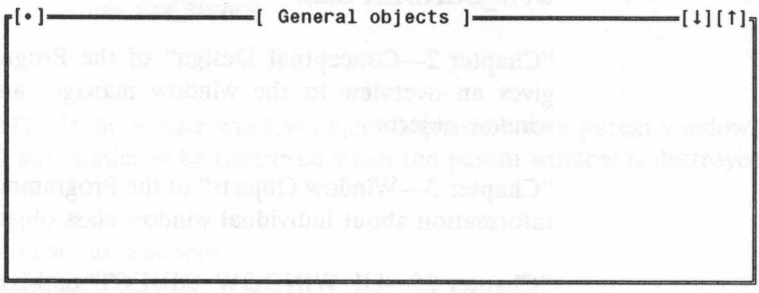
Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a new window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
    .
    .
    .
    // Find out if any fields are invalid.
    int invalidFields = FALSE;
    for (UI_WINDOW_OBJECT *object = window->Last();
        object; object = object->Previous())
        if (FlagSet(object->woStatus, WOS_INVALID))
        {
            invalidFields = TRUE;
            break;
        }
    .
    .
    .
}
```

CHAPTER 26 – UIW_BORDER

Overview The UIW_BORDER class is used to draw a border around a window. The figures below show graphic and textual implementations of a window with a UIW_BORDER class object (the outer-most region of the window):





The public members of the UIW_BORDER class (declared in **UI_WIN.HPP**) are:

```
class UIW_BORDER : public UI_WINDOW_OBJECT
{
public:
    UIW_BORDER(void);
    virtual ~UIW_BORDER(void);
}
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};
```

```
class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_BORDER : public UI_WINDOW_OBJECT;
```

See also The example file **XWGEN.CPP**, which gives a complete example of the UIW_BORDER class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes the base class from which the UIW_BORDER class is derived.

UIW_BORDER::UIW_BORDER

Syntax #include <ui_win.hpp>

UIW_BORDER::UIW_BORDER(void);

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, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        .
        .
        .

2)  // INCORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_BORDER
        .
        .
        .
```

NOTE: If the border window object is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
}
```

UIW_BORDER::~UIW_BORDER

Syntax

#include <ui_win.hpp>

virtual UIW_BORDER::~UIW_BORDER(void);

Remarks This virtual destructor destroys the class information associated with the UIW_BORDER object. Care should be taken to only destroy border objects that are not attached to a parent window.

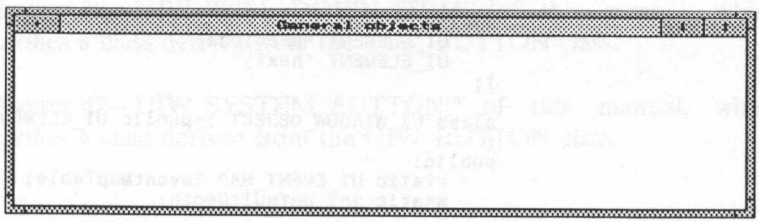
Example

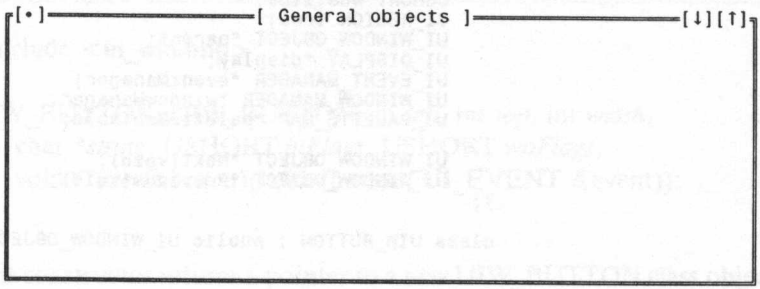
```
#include <ui_win.hpp>

ExampleFunction1()
{
    UIW_BORDER *border = new UIW_BORDER;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_DESTROY);
    *window
        + border
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
    // Manually destroy the border and its parent window.
    *window - border;
    delete border;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the border object
    // destructor.
}
```

CHAPTER 27 – UIW_BUTTON

Overview

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 figures below show graphic and textual implementations of UIW_BUTTON objects (i.e., the maximize button, minimize button and system button):





The public members of the UIW_BUTTON class (declared in **UI_WIN.HPP**) are:

```
class UIW_BUTTON : public UI_WINDOW_OBJECT
{
public:
    int depth;
    USHORT btFlags;

    UIW_BUTTON(int left, int top, int width,
        char *string, USHORT btFlags, USHORT woFlags,
        void (*userFunction)(void *button, UI_EVENT &event));
    virtual ~UIW_BUTTON(void);

    const char *DataGet(void);
    void DataSet(char *string);
};
```

- *depth* is the depth (for 3-dimensional appearance) of the UIW_BUTTON class object. The default depth is set by the static UI_WINDOW_OBJECT member variable *defaultDepth*.

- *btFlags* are flags associated with the UIW_BUTTON class. These flags are described in the UIW_BUTTON constructor.

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_BUTTON : public UI_WINDOW_OBJECT;
```

See also The example file **XWMISC.CPP**, which gives a complete example of the UIW_BUTTON class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes the base class from which the UIW_BUTTON class is derived.

"Chapter 32—UIW_MAXIMIZE_BUTTON" of this manual, which describes a class derived from the UIW_BUTTON class.

"Chapter 33—UIW_MINIMIZE_BUTTON" of this manual, which describes a class derived from the UIW_BUTTON class.

"Chapter 35—UIW_POP_UP_ITEM" of this manual, which describes a class derived from the UIW_BUTTON class.

"Chapter 37—UIW_POP_UP_WINDOW" of this manual, which describes a class derived from the UIW_BUTTON class.

"Chapter 39—UIW_PULL_DOWN_ITEM" of this manual, which describes a class derived from the UIW_BUTTON class.

"Chapter 42—UIW_SYSTEM_BUTTON" of this manual, which describes a class derived from the UIW_BUTTON class.

UIW_BUTTON::UIW_BUTTON

Syntax #include <ui_win.hpp>

UIW_BUTTON::UIW_BUTTON(int *left*, int *top*, int *width*,
char **string*, USHORT *btFlags*, USHORT *woFlags*,
void (**userFunction*)(void *button, UI_EVENT &event));

Remarks This constructor returns a pointer to a new UIW_BUTTON class object.

NOTE: If the button object is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- $left_{in}$ and top_{in} is the starting position of the button within its parent field.
- $width_{in}$ is the width of the button. (The height of the button is determined automatically by the UIW_BUTTON class object.)
- $string_{in}$ is a pointer to the string information associated with the button. This pointer is used by the button object if the WOF_NO_-

ALLOCATE_DATA flag is set. Otherwise, the string information is copied into a buffer allocated by the UIW_BUTTON class object.

- *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_CHECK_MARK—Marks the first position of the button's string information with a check-mark if the button has been selected (i.e., the WOS_SELECTED status flag is set).

  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.

- *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 border around the button object. In graphics mode, setting this option draws a single line border around the object. In text mode, setting this option draws display braces (i.e., '[' ']') around the object.

  WOF_JUSTIFY_CENTER—Center-justifies the *string* information associated with the button object.

  WOF_JUSTIFY_RIGHT—Right-justifies the *string* information associated with the button object.

WOF_NO_ALLOCATE_DATA—Prevents the button object from allocating a string buffer that stores the button's string information. If this flag is set, the programmer must allocate the string buffer (passed as the *string* 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—The button object cannot be selected. If this flag is set, the user will not be able to select the button.

- *userFunction*$_{in}$ is a programmer-defined function that is called whenever the button object is selected. A button object is selected whenever the user positions on the button and presses <Enter> or when the left mouse button is clicked. The following parameters are passed to *userFunction* when the button is selected:

  button$_{in}$ is a pointer to the UIW_BUTTON class object or class object derived from the UIW_BUTTON 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 function. Since this argument is a copy of the original event, it may be changed by the programmer.

Example

```
#include <ui_win.hpp>

static void Exit(void *button, UI_EVENT &event)
{
    event.type = L_EXIT;
    UI_EVENT_MANAGER *eventManager =
        ((UIW_BUTTON *)button)->eventManager;
    eventManager->Put(event);
}

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
```

```
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
        + new UIW_BUTTON(22, 4, 14, "Exit Program", BTF_NO_FLAGS,
          WOF_JUSTIFY_CENTER, Exit);
    .
    .
    .
}
```

UIW_BUTTON::~UIW_BUTTON

Syntax

```
#include <ui_win.hpp>

virtual UIW_BUTTON::~UIW_BUTTON(void);
```

Remarks

This virtual destructor destroys the class information associated with the UIW_BUTTON object. Care should be taken to only destroy button objects that are not attached to a parent window.

Example

```
#include <ui_win.hpp>

static void Exit(void *button, UI_EVENT &event)
{
    event.type = L_EXIT;
    UI_EVENT_MANAGER *eventManager =
        ((UIW_BUTTON *)button)->eventManager;
    eventManager->Put(event);
}

ExampleFunction1()
{
    // Manually add a button to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    UIW_BUTTON *button = new UIW_BUTTON(22, 4, 14, "Exit Program",
        BTF_NO_FLAGS, WOF_JUSTIFY_CENTER, Exit);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
        + button;
    .
    .
    .
    // Manually destroy the button and its parent window.
    *window - button;
    delete button;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the button object
    // destructor.
}
```

UIW_BUTTON::DataGet

Syntax

#include <ui_win.hpp>

const char *UIW_BUTTON::DataGet(void);

Remarks

This function gets the current string information associated with the UIW_BUTTON class object. This function returns a pointer to a constant character array. Thus, the contents of the array cannot be directly modified by the programmer.

- *returnValue*$_{out}$ is a constant pointer to the button's string information.

Example

```
#include <ui_win.hpp>

static void ButtonToggle(void *data, UI_EVENT &event)
{
    UIW_BUTTON *button = (UIW_BUTTON *)data;

    // Toggle the button string.
    const char *string = button->DataGet();
    if (!strcmp("Off", string))
        button->DataSet("On");
    else
        button->DataSet("Off");
}

ExampleFunction1()
{
    // Add a button to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
        + new UIW_BUTTON(22, 4, 14, "Off", BTF_NO_FLAGS,
            WOF_JUSTIFY_CENTER, ButtonToggle);
    .
    .
    .
}
```

UIW_BUTTON::DataSet

Syntax

```
#include <ui_win.hpp>

void UIW_BUTTON::DataSet(char *string);
```

Remarks

This function resets the current string information associated with the UIW_BUTTON class object or tells the class object that key flags, associated with the button object, have been changed.

- $string_{in/out}$ is a pointer to the new string information. 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_BUTTON class object is destroyed. Otherwise, the information associated with this argument is copied by the UIW_BUTTON class object. If this argument is NULL, no string information is changed, but the button is re-displayed.

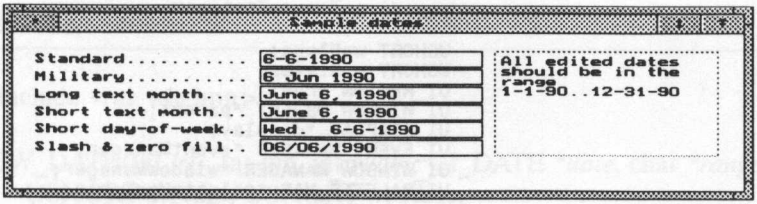
Example

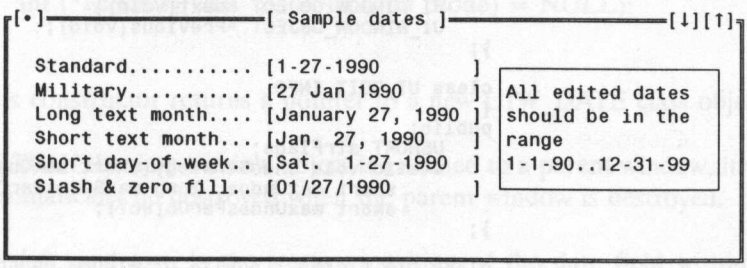
```
#include <ui_win.hpp>

static void ButtonToggle(void *data, UI_EVENT &event)
{
    UIW_BUTTON *button = (UIW_BUTTON *)data;

    // Toggle the button string.
    const char *string = button->DataGet();
    if (!strcmp("Off", string))
        button->DataSet("On");
    else
        button->DataSet("Off");
}

ExampleFunction1()
{
    // Add a button to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
        + new UIW_BUTTON(22, 4, 14, "Off", BTF_NO_FLAGS,
          WOF_JUSTIFY_CENTER, ButtonToggle);
    .
    .
    .
}
```

CHAPTER 28 – UIW_DATE

Overview

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 3—UI_DATE" of this manual for information about the low-level date storage object). The figures below show graphic and textual implementations of a window with several variations of the UIW_DATE class object:





The public members of the UIW_DATE class (declared in **UI_WIN.HPP**) are:

```
class UIW_DATE : public UIW_STRING
{
public:
    UIW_DATE(int left, int top, int width, UI_DATE *date,
        char *range, USHORT dtFlags, USHORT woFlags,
        int (*validate)(void *dateField, int ccode) = NULL);
    virtual ~UIW_DATE(void);

    const UI_DATE *DataGet(void);
    void DataSet(UI_DATE *date);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UI_EDIT_INFO
{
public:
    USHORT strFlags;
    static void UndoStrategy(short maxObjects, long maxBytes,
        short maxUndos, long maxBytesPerObject,
        short maxUndosPerObject);
};

class UIW_STRING : public UI_WINDOW_OBJECT, public UI_EDIT_INFO
{
public:
    const char *DataGet(void);
    void DataSet(char *string, int maxLength = -1);
};

class UIW_DATE : public UIW_STRING;
```

See also The example file **XWDATE.CPP**, which gives a complete example of the UIW_DATE class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 3—UI_DATE" of this manual, which describes the low-level date storage class object.

"Chapter 41—UIW_STRING" of this manual, which describes the base class from which the UIW_DATE class is derived.

"Chapter 44—UIW_TIME" of this manual, which describes a similar high-level class object that stores time information.

UIW_DATE::UIW_DATE

Syntax #include <ui_win.hpp>

UIW_DATE(int *left*, int *top*, int *width*, UI_DATE **date*, char **range*,
USHORT *dtFlags*, USHORT *woFlags*,
int (**validate*)(void *dateField, int ccode) = NULL);

Remarks This constructor returns a pointer to a new UIW_DATE class object.

NOTE: If the date window object is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- $left_{in}$ and top_{in} is 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 accepted. This string is 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 dependant information supplied by all DOS based systems:

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

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

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

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

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

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

DTF_NO_FLAGS—Does not associate any special flags with the UIW_DATE class object. In this case, the date will be displayed and interpreted using the default country information. For example, the U.S. date "DEC 4 1989" would be shown as "4 DEC 1989" if default country information specified a European format, or "1989 DEC 4" if the default country information specified a Japanese format. This flag should not be used in conjunction with any other DTF flag.

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

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

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

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

DTF_SYSTEM—Fills a blank date with the system date. For example, if a blank ascii date were entered by the end-user and the DTF_SYSTEM flag were set, then the date would be set to the system date (e.g., "December 4, 1990").

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

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

the DTF_US_FORMAT flag set are: "March 28, 1990," "12/31/1980" and "Jan 3, 2003."

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

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the date object. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_DATE class object:

  WOF_AUTO_CLEAR—Automatically clears the date buffer if the end-user positions on to the first character of the date field (from another window field) then presses a key (without having previously pressed any movement or editing keys).

  WOF_BORDER—Draws a border around the date object. In graphics mode, setting this option draws a single line border around the object. In text mode, setting this option draws display braces (i.e., '[' ']') around the object.

  WOF_INVALID—Sets the initial status of the date field to be "invalid." An invalid date fits in the absolute range determined by the object type (i.e., "1-1-100..12-31-32767") but does not fulfill all the requirements specified by the program. For example, a date may initially be set to "3-12-90," but the final date, edited by the end-user, must be in the range "12-1-90..12-31-90." The initial date in this example fits the absolute requirements of a UIW_DATE class object but does not fit into the specified range.

  WOF_JUSTIFY_CENTER—Center-justifies the date information within the date field.

  WOF_JUSTIFY_RIGHT—Right-justifies the date information within the date field.

  WOF_NO_ALLOCATE_DATA—Prevents the date object from allocating a UI_DATE class object to store the date information. If this flag is set, the programmer must allocate

the UI_DATE (passed as the *date* parameter) that is used by the date object.

WOF_NO_FLAGS—Does not associate any special window 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 flags.

WOF_NO_INVALID—Prevents the "Leave invalid" option from being selectable by the end-user when an incomplete or invalid date is entered.

WOF_NO_UNANSWERED—Prevents the "Leave unanswered" option from being selectable by the end-user when an incomplete or invalid date is entered.

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.

- *validate*$_{in}$ is a programmer defined function that is called whenever:

  1—a date string is entered and the user moves to a different field in the form, or

  2—the user moves to a different window on the screen.

  The *validate* function is <u>not</u> called if the date does not fit the absolute range for dates or if the date is outside the default range (specified by the *range* argument passed in on the UIW_DATE constructor). The following arguments are passed to *validate* when a new date is entered:

  dateField$_{in}$—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.

ccode$_{in}$—The logical or system code that caused the validate function to be called. This code (declared in UI_EVT.HPP) will be one of the following constant values:

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. This code is sent after editing operations have been performed, if the date is valid for the absolute value of date field ranges and if date is valid for the programmer defined *range*.

The validate function's *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.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Add several time fields to a window.
    UIW_WINDOW *window = UIW_WINDOW(0, 0, 45, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_PROMPT(2, 1, "Standard.....", WOF_NO_FLAGS)
        + new UIW_DATE(22, 1, 20, &date, DT_SYSTEM, DT_SYSTEM,
            DT_SYSTEM, "1-1-90..12-31-99", DTF_NO_FLAGS,
            WOF_BORDER)
        + new UIW_PROMPT(2, 3, "Long text month....", WOF_NO_FLAGS)
        + new UIW_DATE(22, 3, 20, &date, "1-1-90..12-31-99",
            DTF_ALPHA_MONTH | DTF_SYSTEM, WOF_BORDER)
        + new UIW_PROMPT(2, 6, "Slash & zero fill..", WOF_NO_FLAGS)
        + new UIW_DATE(22, 6, 20, &date, "1-1-90..12-31-99",
            DTF_SLASH | DTF_ZERO_FILL | DTF_SYSTEM, WOF_BORDER);
        .
        .
        .
}
```

UIW_DATE::~UIW_DATE

Syntax

```
#include <ui_win.hpp>

virtual UIW_DATE::~UIW_DATE(void);
```

Remarks This virtual destructor destroys the class information associated with the UIW_DATE object. Care should be taken to only destroy those date objects that are not attached to a parent window.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    UI_DATE date; // system date
    UIW_DATE *dateField = new UIW_DATE(9, 1, 20, &date,
        "", DTF_ALPHA_MONTH | DTF_SYSTEM, WOF_BORDER);
    UI_TIME time; // system time
    UIW_TIME *timeField = new UIW_TIME(9, 1, 20, &time,
        "", TMF_SECONDS, WOF_BORDER);

    // Create a window with system date and time information.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Date..", WOF_NO_FLAGS)
        + dateField
        + new UIW_PROMPT(2, 1, "Time..", WOF_NO_FLAGS)
        + timeField;
    .
    .
    .
    // Manually destroy the date field and its parent window.
    *window - dateField;
    delete dateField;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the date object destructor.
}
```

UIW_DATE::DataGet

Syntax #include <ui_win.hpp>

const UI_DATE *UIW_DATE::DataGet(void);

Remarks This function gets the current date information associated with the UIW_DATE class object. This function returns a pointer to a constant UI_DATE variable. Thus, the contents of this variable cannot be directly modified by the programmer.

- *returnValue*$_{out}$ is a constant pointer to the UI_DATE variable.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    UI_DATE date; // system date
    UIW_DATE *dateField = new UIW_DATE(9, 1, 20, &date,
        "", DTF_ALPHA_MONTH | DTF_SYSTEM, WOF_BORDER);
    UI_TIME time; // system time
    UIW_TIME *timeField = new UIW_TIME(9, 2, 20, &time,
        "", TMF_SECONDS, WOF_BORDER);

    // Create a window with system date and time information.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Date..", WOF_NO_FLAGS)
        + dateField
        + new UIW_PROMPT(2, 2, "Time..", WOF_NO_FLAGS)
        + timeField;
    .
    .
    .
    // Reset the date and time information.
    extern int ResetDateTime(void);
    if (ResetDateTime())
    {
        date.Import();  // Get the new system date.
        time.Import();  // Get the new system time.
        dateField->DataSet(&date);
        timeField->DataSet(&time);
    }
    else
    {
        date = *dateField->DataGet();
        time = *timeField->DataGet();
    }
    .
    .
    .
}
```

UIW_DATE::DataSet

Syntax

#include <ui_win.hpp>

void UIW_DATE::DataSet(UI_DATE **date*);

Remarks

This function resets the current date information associated with the UIW_DATE class object or tells the class object that key flags, associated with the date object, have been changed.

- $date_{in/out}$ is a pointer to the new date information. 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. Otherwise, the information associated with this argument is copied by the UIW_DATE class object. If this argument is NULL, no date information is changed, but the date field is re-displayed.

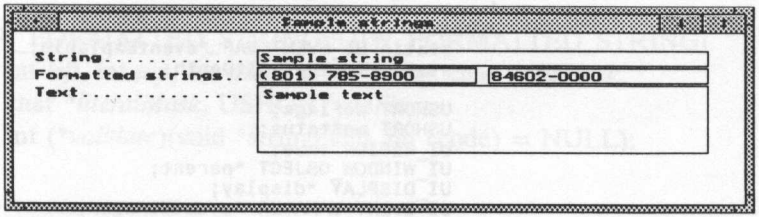
Example

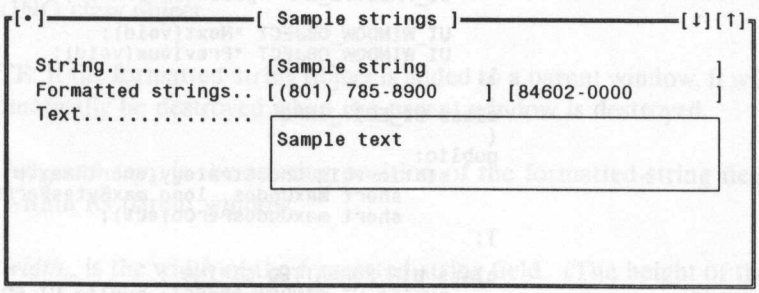
```
#include <ui_win.hpp>

ExampleFunction1()
{
    UI_DATE date; // system date
    UIW_DATE *dateField = new UIW_DATE(9, 1, 20, &date,
        "", DTF_ALPHA_MONTH | DTF_SYSTEM, WOF_BORDER);
    UI_TIME time; // system time
    UIW_TIME *timeField = new UIW_TIME(9, 2, 20, &time,
        "", TMF_SECONDS, WOF_BORDER);

    // Create a window with system date and time information.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Date..", WOF_NO_FLAGS)
        + dateField
        + new UIW_PROMPT(2, 2, "Time..", WOF_NO_FLAGS)
        + timeField;
    .
    .
    .
    // Reset the date and time information.
    extern int ResetDateTime(void);
    if (ResetDateTime())
    {
        date.Import();  // Get the new system date.
        time.Import();  // Get the new system time.
        dateField->DataSet(&date);
        timeField->DataSet(&time);
    }
    else
    {
        date = *dateField->DataGet();
        time = *timeField->DataGet();
    }
    .
    .
    .
}
```

CHAPTER 29 – UIW_FORMATTED_STRING

Overview

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 figures below show graphic and textual implementations of two UIW_FORMATTED_STRING class objects:





The public members of the UIW_FORMATTED_STRING class (declared in **UI_WIN.HPP**) are:

```
class UIW_FORMATTED_STRING :
    public UI_WINDOW_OBJECT, public UI_EDIT_INFO
{
public:
    UIW_FORMATTED_STRING(int left, int top, int width,
        char *string, char *editMask, char *literalMask,
        USHORT woFlags,
        int (*validate)(void *stringField, int ccode) = NULL);
    virtual ~UIW_FORMATTED_STRING(void);

    const char *DataGet(void);
    void DataSet(char *string);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UI_EDIT_INFO
{
public:
    static void UndoStrategy(short maxObjects, long maxBytes,
        short maxUndos, long maxBytesPerObject,
        short maxUndosPerObject);
};

class UIW_FORMATTED_STRING :
    public UI_WINDOW_OBJECT, public UI_EDIT_INFO;
```

See also The example file **XWSTRING.CPP**, which gives a complete example of the UIW_FORMATTED_STRING class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes the base class from which the UIW_FORMATTED_STRING class is derived.

"Chapter 41—UIW_STRING" of this manual, which describes a similar class derived from the UIW_WINDOW_OBJECT and UI_EDIT_INFO classes.

UIW_FORMATTED_STRING::UIW_FORMATTED_STRING

Syntax

```
#include <ui_win.hpp>

UIW_FORMATTED_STRING::UIW_FORMATTED_STRING(
    int left, int top, int width, char *string, char *editMask,
    char *literalMask, USHORT woFlags,
    int (*validate)(void *stringField, int ccode) = NULL);
```

Remarks This constructor returns a pointer to a new UIW_FORMATTED_-STRING class object.

NOTE: If the formatted-string object is added to a parent window, it will automatically be destroyed when the parent window is destroyed.

- $left_{in}$ and top_{in} is 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.)
- $string_{in}$ is an initial string that conforms to the *editMask* and *literalMask* arguments. This pointer is used by the formatted-string object if the WOF_NO_ALLOCATE_DATA flag is set for the field. Otherwise, the string is copied into a buffer, allocated by the UIW_FORMATTED_STRING object.
- $editMask_{in}$ is a character array that indicates the type of characters that can be entered at different positions in the format 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 upper-case.

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 letter 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 or 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 upper-case.

- *literalMask*$_{in}$ 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. This argument is copied by the UIW_FORMATTED_STRING class object.

- *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 positions on the first character of the field (from another window field) then presses a key (without having previously pressed any movement or editing keys).

  WOF_BORDER—Draws a border around the formatted string object. In graphics mode, setting this option draws a single line border around the object. In text mode, setting this option draws display brackets (i.e., '[' ']') around the object.

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 *validate* function (discussed below). For example, a formatted-string field (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 string buffer that stores the string information. (This has no effect on the edit mask or literal mask.) If this flag is set, the programmer must allocate the string buffer (passed as the *string* parameter) that is used by the formatted-string object.

WOF_JUSTIFY_CENTER—Center-justifies the string information within the formatted-string field.

WOF_JUSTIFY_RIGHT—Right-justifies the string information within the formatted-string field.

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_NO_INVALID—Prevents the "Leave invalid" option from being selectable by the end-user when an incomplete or invalid formatted string is entered.

WOF_NO_UNANSWERED—Prevents the "Leave unanswered" option from being selectable by the end-user when an incomplete or invalid formatted string is entered.

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.

- *validate*$_{in}$ is a programmer defined function that is called whenever:

  1—a formatted string is entered and the user moves to a different field in the form, or

  2—the user moves to a different window on the screen.

  The following arguments are passed to *validate* when formatted-string information is entered:

  stringField$_{in}$—A pointer to the UIW_FORMATTED_STRING class object or the class object derived from the UIW_-FORMATTED_STRING object base class. This argument must be typecast by the programmer.

  ccode$_{in}$—The logical or system code that caused the validate function to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

  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 validate 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>

ExampleFunction1()
{
    // Add formatted string fields to a window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Formatted strings..", WOF_NO_FLAGS)
        + new UIW_FORMATTED_STRING(22, 1, 41, "8017858900",
            "LNNNLLNNNLNNNN", "(...) ...-....", WOF_BORDER)
        + new UIW_FORMATTED_STRING(43, 2, 20, "846020000",
            "NNNNNLNNNN", ".....-....", WOF_BORDER);
    .
    .
    .
}
```

UIW_FORMATTED_STRING::~UIW_FORMATTED_STRING

Syntax

```
#include <ui_win.hpp>

virtual UIW_FORMATTED_STRING::
    ~UIW_FORMATTED_STRING(void);
```

Remarks

This virtual destructor destroys the class information associated with the UIW_FORMATTED_STRING object. Care should be taken to only destroy formatted-string objects that are not attached to a parent window.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Add a string field to the window.
    UIW_FORMATTED_STRING *string1 =
        new UIW_FORMATTED_STRING(22, 1, 21, "8017858900",
        "LNNNLLNNNLNNNN", "(...) ...-....", WOF_BORDER);
    UIW_FORMATTED_STRING *string2 =
        new UIW_FORMATTED_STRING(43, 2, 20, "846020000",
        "NNNNNLNNNN", ".....-....", WOF_BORDER)

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Formatted strings..", WOF_NO_FLAGS)
        + string1
        + string2;
    .
    .
    .
```

```
    // Manually destroy the formatted string fields and their
    // parent window.
    *window - string1 - string2;
    delete string1;
    delete string2;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the formatted string field
    // destructors.
}
```

UIW_FORMATTED_STRING::DataGet

Syntax

```
#include <ui_win.hpp>

const char *UIW_FORMATTED_STRING::DataGet(void);
```

Remarks

This function gets the current formatted-string buffer information associated with the UIW_FORMATTED_STRING class object. This function returns a pointer to a constant character array. Thus, the contents of the array cannot be directly modified by the programmer.

- *returnValue*$_{out}$ is a constant pointer to the formatted-string buffer.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a formatted string field.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    UIW_FORMATTED_STRING *stringField =
        new UIW_FORMATTED_STRING(22, 1, 41, "8017858900",
            "LNNNLLNNNLNNNN", "(...) ...-....", WOF_BORDER);

    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Formatted strings..", WOF_NO_FLAGS)
        + stringField;
    .
    .
    .
    // Get the contents of the formatted string buffer.
    const char *buffer = stringField->DataGet();
    .
    .
    .
}
```

UIW_FORMATTED_STRING::DataSet

Syntax #include <ui_win.hpp>

void DataSet(char *_string_);

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.

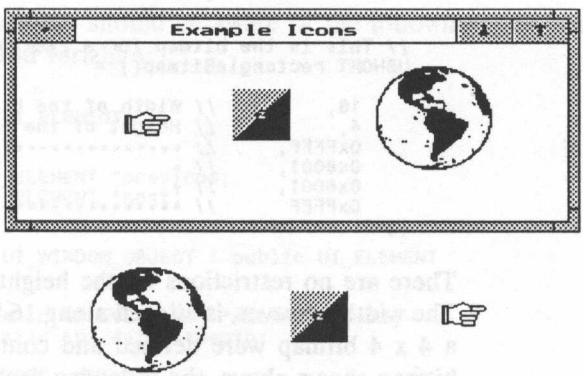
- $string_{in}$ is a pointer to the new string. This string must conform to the *editMask* and *literalMask* arguments passed to the formatted-string constructor. If the WOF_NO_ALLOCATE_DATA flag is set, this argument must be space, allocated by the programmer, that is not destroyed until the UIW_FORMATTED_STRING 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>

ExampleFunction1()
{
    // Manually add a formatted string field to a window.
    UIW_FORMATTED_STRING *stringField =
        new UIW_FORMATTED_STRING(22, 1, 41, "8017858900",
            "LNNNLLNNNLNNNN", "(...) ...-....", WOF_BORDER);

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Formatted strings..", WOF_NO_FLAGS)
        + stringField;
    .
    .
    .
    // Reset the formatted string field buffer.
    stringField->DataSet("8017858998");
    .
    .
    .
}
```

CHAPTER 30 – UIW_ICON

Overview The UIW_ICON class is used to display a bitmap image to the screen. It is only available when using a graphic screen display (i.e., the UI_DOS_BGI_DISPLAY class). The figure below shows the graphic implementation of several UIW_ICON objects:



The public members of the UIW_ICON class (declared in **UI_WIN.HPP**) are:

```
class UIW_ICON : public UI_WINDOW_OBJECT
{
public:
    static USHORT repeatRate;
    USHORT icFlags;

    UIW_ICON(int left, int top, USHORT **bitmapArray,
        UI_PALETTE *paletteArray, USHORT icFlags,
        USHORT woFlags,
        void (*userFunction)(void *icon, UI_EVENT &event));
    virtual ~UIW_ICON(void);

    void DataGet(const USHORT **bitmapArray,
        const UI_PALETTE *paletteArray);
    void DataSet(USHORT **bitmapArray,
        UI_PALETTE *paletteArray);
};
```

A bitmap is defined by an array of USHORT (unsigned short) values. Each bitmap must be constructed in the following manner:

1—The first USHORT value must contain the width of the bitmap in pixels.

2—The second USHORT value must contain the height of the bitmap in pixels.

3—All remaining USHORT values contain the actual bitmap pattern. Each of these values is evaluated from high- to low-bit.

For example, a 16 x 4 bitmap (16 columns, 4 lines) that draws a rectangle could be represented in the following manner:

```
// This is the bitmap for a rectangle.
USHORT rectangleBitmap[] =
{
    16,          // Width of the bitmap pattern.
    4,           // Height of the bitmap pattern.
    0xFFFF,      // ................
    0x8001,      // .              .
    0x8001,      // .              .
    0xFFFF       // ................
};
```

There are no restrictions on the height and width of a bitmap image. The width however, is aligned along 16 bit boundaries. For example, if a 4 x 4 bitmap were defined and contained a similar pattern to the bitmap shown above, the following declaration could be made:

```
// This is the bitmap for a box.
USHORT boxBitmap[] =
{
    4,           // Width of the bitmap pattern.
    4,           // Height of the bitmap pattern.
    0xF000,      // ....      Only the top four bits
    0x9000,      // .  .      of these values are
    0x9000,      // .  .      evaluated.
    0xF000       // ....
};
```

Similarly, a 32 x 4 rectangle bitmap would be represented in the following manner:

```
// This is the bitmap for 32 pixel wide rectangle.
USHORT rectangleBitmap[] =
{
    32,                  // Width of the bitmap pattern.
    4,                   // Height of the bitmap pattern.
                         // Each line is represented by
                         // 2 USHORTS.
    0xFFFF, 0xFFFF,      // ................................
    0x8000, 0x0001,      // .                              .
    0x8000, 0x0001,      // .                              .
    0xFFFF, 0xFFFF       // ................................
};
```

The highlight bits (i.e., those bits whose values are 1) of the bitmap are drawn with the foreground color of the palette argument. The non-

highlight bits (i.e., those bits whose values are 0) of the bitmap are drawn with the background color of the associated color palette.

The constants BITMAP_WIDTH and BITMAP_HEIGHT are declared to give array access to the height and width values of a specified bitmap. Bitmaps do not have text screen equivalents.

Inheritance

The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_ICON : public UI_WINDOW_OBJECT;
```

See also

The example file **XWMISC.CPP**, which gives a complete example of the UIW_ICON class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes the base class from which the UIW_ICON class is derived.

"Chapter 27—UIW_BUTTON" of this manual, which describes a similar selectable class object.

UIW_ICON::UIW_ICON

Syntax #include <ui_win.hpp>

UIW_ICON::UIW_ICON(int *left*, int *top*, USHORT ***bitmapArray*,
UI_PALETTE **paletteArray*, USHORT *icFlags*,
USHORT *woFlags*,
void (**userFunction*)(void *icon, UI_EVENT &event));

Remarks This constructor returns a pointer to a new UIW_ICON class object.

NOTE: If the icon is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- $left_{in}$ and top_{in} is the starting position of the icon in the window. The ending position is determined by the bitmap pattern.
- $bitmapArray_{in}$ is an array of bitmap entries where the last entry is NULL (see example below).
- $paletteArray_{in}$ is an array of palette entries. This array must match the *bitmapArray* for total number of entries.
- $icFlags_{in}$ 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_DOUBLE_CLICK—Completes the icon action on a double-click, rather than on a single down-click and release action. This flag has no effect on keyboard interfaces.

  ICF_DOWN_CLICK—Completes the icon action on a button down-click, rather than on a down-click and release action. This flag has no effect on keyboard interfaces.

ICF_NO_FLAGS—No special options are selected with the UIW_ICON class object. In this state the icon requires a down and up click from the mouse to complete an action.

- *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 object.

  WOF_NO_FLAGS—Uses default information about the icon object.

  WOF_NON_FIELD_REGION—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*, *top* and *width* 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—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 user defined function that is called whenever the icon is selected. An icon is selected whenever the user is positioned on the icon and presses <Enter>, or when the left mouse button is clicked. The following parameters are passed to *userFunction* when the icon is selected:

  icon$_{in}$ is a pointer to the UIW_ICON class object or class object derived from the UIW_ICON 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.

Example

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

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

static UI_PALETTE handPalettes[] = { {
    '\260', attrib(BLACK, WHITE), attrib(MONO_DIM, MONO_BLACK),
    INTERLEAVE_FILL, attrib(BLACK, WHITE),
    attrib(BW_WHITE, BW_WHITE), attrib(GS_GRAY, GS_GRAY) } };

ExampleFunction1()
{
    // Attach the icon to a window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Example Icons", WOF_JUSTIFY_CENTER)
        + new UIW_ICON(7, 3, handBitmaps, handPalettes,
            ICF_NO_FLAGS, WOF_NO_FLAGS);

    // Attach the icon to the window manager.
    UIW_ICON *icon = new UIW_ICON(7, 3, handBitmaps, handPalettes,
        ICF_NO_FLAGS, WOF_NO_FLAGS);
    extern UI_WINDOW_MANAGER *windowManager;
    *windowManager + icon;
    .
    .
    .
}
```

UIW_ICON::~UIW_ICON

Syntax

#include <ui_win.hpp>

virtual UIW_ICON::~UIW_ICON(void);

Remarks

This virtual destructor destroys the class information associated with the UIW_ICON object. Care should be taken to only destroy icon objects that are not attached to a the parent window.

Example

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

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

static UI_PALETTE handPalettes[] = { {
    '\260', attrib(BLACK, WHITE), attrib(MONO_DIM, MONO_BLACK),
    INTERLEAVE_FILL, attrib(BLACK, WHITE),
    attrib(BW_WHITE, BW_WHITE), attrib(GS_GRAY, GS_GRAY) } };

ExampleFunction1()
{
    // Attach the icon to the window.
    UIW_ICON *icon1 = new UIW_ICON(7, 3, handBitmaps, handPalettes,
        ICF_NO_FLAGS, WOF_NO_FLAGS);

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Example Icons", WOF_JUSTIFY_CENTER)
        + icon1;

    // Attach the icon to the window manager.
    UIW_ICON *icon2 = new UIW_ICON(0, 0, handBitmaps, handPalettes,
        ICF_NO_FLAGS, WOF_NO_FLAGS);
    extern UI_WINDOW_MANAGER *windowManager;
    *windowManager + icon2;
    .
    .
    .
    // Manually destroy the window and screen icons.
    *window - icon1;
    delete icon1;
    delete window;
    *windowManager - icon2;
    delete icon2;
    delete windowManager;
    // We could have just called "delete window" to destroy icon1.
    // Its destructor would have automatically called the icon
    // object destructor.
}
```

UIW_ICON::DataGet

Syntax

```
#include <ui_win.hpp>

void UIW_ICON::DataGet(const USHORT **bitmapArray,
    const UI_PALETTE *paletteArray);
```

Remarks

This function gets the current bitmap and palette information associated with the UIW_ICON class object. Both returned arguments are constants. Thus the arguments cannot be directly modified by the programmer.

- *bitmapArray*$_{out}$ is a constant array of bitmap entries.
- *paletteArray*$_{out}$ is a constant array of palette entries.

Example

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

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

static UI_PALETTE blueHand[] = { {
    '\260', attrib(BLACK, WHITE), attrib(MONO_DIM, MONO_BLACK),
    INTERLEAVE_FILL, attrib(BLUE, WHITE),
    attrib(BW_WHITE, BW_WHITE), attrib(GS_GRAY, GS_GRAY) } };
static UI_PALETTE redHand[] = { {
    '\260', attrib(BLACK, WHITE), attrib(MONO_DIM, MONO_BLACK),
    INTERLEAVE_FILL, attrib(RED, WHITE),
    attrib(BW_WHITE, BW_WHITE), attrib(GS_GRAY, GS_GRAY) } };

ExampleFunction1()
{
    // Attach the icon to the window.
    UIW_ICON *icon = new UIW_ICON(7, 3, handBitmaps, blueHand,
        ICF_NO_FLAGS, WOF_NO_FLAGS);
```

```
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Example Icons", WOF_JUSTIFY_CENTER)
        + icon;
    .
    .
    .
    // Change the hand color.
    const UI_PALETTE_MAP *palette;
    icon->DataGet(NULL, palette);
    if (palette[0]->color == attrib(BLUE, WHITE))
        icon->DataSet(NULL, redHand);
    else
        icon->DataSet(NULL, blueHand);
}
```

UIW_ICON::DataSet

Syntax

#include <ui_win.hpp>

void UIW_ICON::DataSet(USHORT ***bitmapArray*, UI_PALETTE ****paletteArray*);

Remarks

This function resets the bitmap and/or palette information associated with the UIW_ICON class object or tells the class object that key flags, associated with the icon, have been changed.

- $bitmapArray_{out}$ is the new array of bitmap entries.
- $paletteArray_{out}$ is the new array of palette entries.

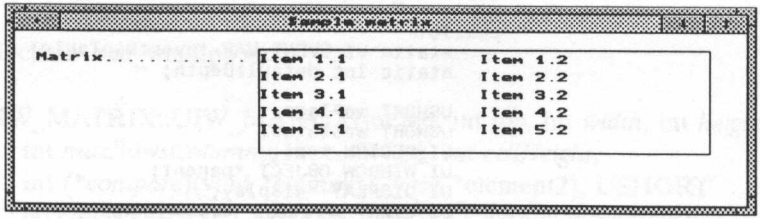
Example

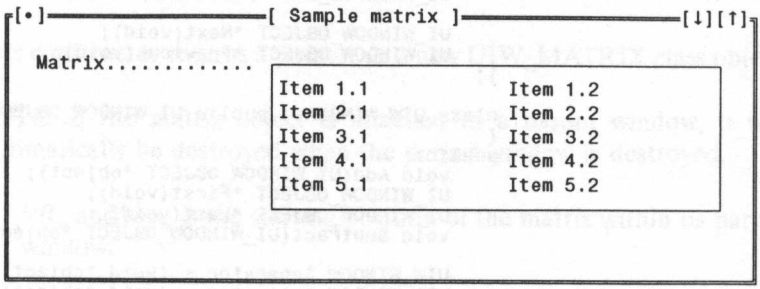
```
#include <graphics.h>
#include <ui_win.hpp>

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

```
static UI_PALETTE blueHand[] = { {
    '\260', attrib(BLACK, WHITE), attrib(MONO_DIM, MONO_BLACK),
    INTERLEAVE_FILL, attrib(BLUE, WHITE),
    attrib(BW_WHITE, BW_WHITE), attrib(GS_GRAY, GS_GRAY) } };
static UI_PALETTE redHand[] = { {
    '\260', attrib(BLACK, WHITE), attrib(MONO_DIM, MONO_BLACK),
    INTERLEAVE_FILL, attrib(RED, WHITE),
    attrib(BW_WHITE, BW_WHITE), attrib(GS_GRAY, GS_GRAY) } };

ExampleFunction1()
{
    // Attach the icon to the window.
    UIW_ICON *icon = new UIW_ICON(7, 3, handBitmaps, blueHand,
        ICF_NO_FLAGS, WOF_NO_FLAGS);

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Example Icons", WOF_JUSTIFY_CENTER)
        + icon;
    .
    .
    .
    // Change the hand color.
    const UI_PALETTE_MAP *palette;
    icon->DataGet(NULL, palette);
    if (palette[0]->color == attrib(BLUE, WHITE))
        icon->DataSet(NULL, redHand);
    else
        icon->DataSet(NULL, blueHand);
}
```

CHAPTER 31 – UIW_MATRIX

Overview The UIW_MATRIX class is used to display related information in a row/column fashion within a window. The figures below show graphic and textual implementations of a UIW_MATRIX object with several string objects:





The public members of the UIW_MATRIX class (declared in **UI_WIN.HPP**) are:

```
class UIW_MATRIX : public UIW_WINDOW
{
public:
    USHORT mxFlags;

    UIW_MATRIX(int left, int top, int width, int height,
        int maxRowsColumns, int cellWidth, int cellHeight,
        int (*compare)(void *element1, void *element2),
        USHORT mxFlags, USHORT woFlags,
        USHORT woAdvancedFlags);
    virtual ~UIW_MATRIX(void);
};
```

- *mxFlags* are flags that determine the display pattern of matrix items. A complete description of these flags is given in the matrix class constructor.

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_WINDOW : public UI_WINDOW_OBJECT
{
public:
    void Add(UI_WINDOW_OBJECT *object);
    UI_WINDOW_OBJECT *First(void);
    UI_WINDOW_OBJECT *Last(void)
    void Subtract(UI_WINDOW_OBJECT *object);

    UIW_WINDOW &operator + (void *object);
    UIW_WINDOW &operator - (void *object);
};

class UIW_MATRIX : public UIW_WINDOW;
```

See also The example file **XWMATRIX.CPP**, which gives a complete example of the UIW_MATRIX class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes the base class from which the UIW_WINDOW class is derived.

"Chapter 36—UIW_POP_UP_MENU" of this manual, which describes a similar class derived from the UIW_WINDOW class.

"Chapter 40—UIW_PULL_DOWN_MENU" of this manual, which describes a similar class derived from the UIW_WINDOW class.

UIW_MATRIX::UIW_MATRIX

Syntax #include <ui_win.hpp>

UIW_MATRIX::UIW_MATRIX(int *left*, int *top*, int *width*, int *height*, int *maxRowsColumns*, int *cellWidth*, int *cellHeight*, int (**compare*)(void *element1, void *element2), USHORT *mxFlags*, USHORT *woFlags*, USHORT *woAdvancedFlags*);

Remarks This constructor returns a pointer to a new UIW_MATRIX class object.

NOTE: If the matrix object is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- $left_{in}$ and top_{in} is the starting position of the matrix within its parent window.
- $width_{in}$ is the width of the matrix field.
- $height_{in}$ is the height of the matrix field.
- $maxRowsColumns_{in}$ specifies the maximum number of matrix elements that can be placed horizontally (if the MXF_COLUMNS_FILL flag is set) or vertically (if the MXF_ROWS_FILL flag is set) on the screen.
- $cellWidth_{in}$ specifies the maximum width of a single matrix item.
- $cellHeight_{in}$ specifies the maximum height of a single matrix item.
- $compare_{in}$ is a programmer defined function used to determine the order of matrix items. New items are placed at the end of a matrix

list if this value is NULL. 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. If a negative value is returned, then *element1* is less than *element2*. Otherwise, a positive value indicates that *element1* is greater than *element2*.

- *mxFlags*$_{in}$ are flags that determine the display pattern of the matrix items. The following flags (declared in **UI_WIN.HPP**) control the general presentation of the matrix items:

  MXF_COLUMN_FILL—Puts the matrix items in a column filled position. Setting this flag will ensure that the columns of the matrix are filled before the rows are filled.

  MXF_NO_FLAGS—Sorts the matrix items according to *compare*. If this flag is set, the programmer must specify the screen position of each matrix item upon its creation.

  MXF_ROW_FILL—Puts the matrix items in a row filled position. Setting this flag will ensure that the rows of the matrix are filled before the columns are filled.

- *woFlags*$_{in}$ are flags (general to all window objects) that determine the general operation of the matrix object. The following flags (declared in **UI_WIN.HPP**) change the presentation of, or interaction with, a UIW_MATRIX class object:

  WOF_BORDER—Draws a single line border around the matrix object.

  WOF_NO_FLAGS—Does not associate any special flags with the matrix. This flag should not be used in conjunction with any other WOF flag.

WOF_NON_FIELD_REGION—The matrix object is not a form field. If this flag is set and the matrix is attached to a higher-level window, then the *left*, *top*, *width* and *height* arguments are ignored and the matrix will occupy any remaining space within the parent window. Otherwise, this advanced flag should only be used when attaching a matrix object directly to the screen display.

WOF_NON_SELECTABLE—The matrix 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 matrix items.

- *woAdvancedFlags*$_{in}$ are flags that determine the advanced operation of the matrix object.

  WOAF_NO_FLAGS—Does not associated any special advanced flags with the matrix. Setting this flag allows the user to move, size and interact with the matrix in a normal fashion. This flag should not be used in conjunction with any other WOAF flag.

  WOAF_TEMPORARY—The matrix only occupies the screen temporarily. Once another window is selected from the screen, the temporary matrix is destroyed.

  WOAF_NO_DESTROY—Prevents the window manager from calling the matrix destructor. If this flag is set, the matrix can be removed from the screen display, but the programmer must call the associated matrix destructor.

  WOAF_NO_SIZE—Prevents the end-user from changing the size of the matrix 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 matrix from the screen display.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create the matrix field.
    UIW_MATRIX * matrix = new UIW_MATRIX(22, 1, 41, 6, 5, 14, 1, 0,
        MXF_NO_FLAGS, WOF_BORDER, WOAF_NO_FLAGS);
    *matrix
        + new UIW_STRING(0, 0, 19, "Item 1.1", 64, STF_NO_FLAGS,
            WOF_NO_FLAGS)
        + new UIW_STRING(20, 0, 19, "Item 1.2", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(0, 1, 19, "Item 2.1", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(20, 1, 19, "Item 2.2", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS);

    // Attach the matrix to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + UIW_BORDER
        + new UIW_TITLE(" Sample matrix ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Matrix.....", WOF_NO_FLAGS)
        + matrix;
    .
    .
    .
}
```

UIW_MATRIX::~UIW_MATRIX

Syntax

#include <ui_win.hpp>

virtual UIW_MATRIX::~UIW_MATRIX(void);

Remarks

This virtual destructor destroys the class information associated with the UIW_MATRIX object. Care should be taken to only destroy matrix objects that are not automatically destroyed by the parent window.

Example

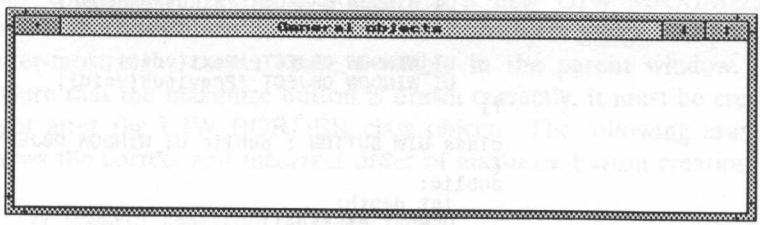
```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create the matrix field.
    UIW_MATRIX * matrix = new UIW_MATRIX(22, 1, 41, 6, 5, 14, 1, 0,
        MXF_NO_FLAGS, WOF_BORDER, WOAF_NO_FLAGS);
    *matrix
        + new UIW_STRING(0, 0, 19, "Item 1.1", 64, STF_NO_FLAGS,
            WOF_NO_FLAGS)
        + new UIW_STRING(20, 0, 19, "Item 1.2", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(0, 1, 19, "Item 2.1", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS)
        + new UIW_STRING(20, 1, 19, "Item 2.2", 64,
            STF_NO_FLAGS, WOF_NO_FLAGS);
```

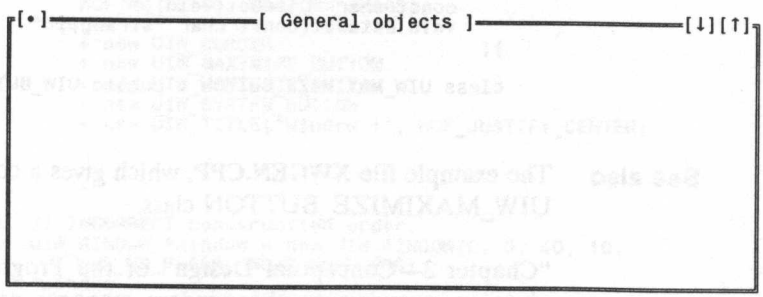
```
    // Attach the matrix to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + UIW_BORDER
        + new UIW_TITLE(" Sample matrix ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Matrix.............", WOF_NO_FLAGS)
        + matrix;
    .
    .
    .
    // Manually destroy the matrix field and its parent window.
    *window - matrix;
    delete matrix;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the matrix object
    // destructor.
}
```

CHAPTER 32 – UIW_MAXIMIZE_BUTTON

Overview

The UIW_MAXIMIZE_BUTTON class is used to maximize a window. A maximized window fills the entire display screen. The figures below show graphic and textual implementations of a window with a UIW_-MAXIMIZE_BUTTON class object (the button with the '↑' character):





The public members of the UIW_MAXIMIZE_BUTTON class (declared in **UI_WIN.HPP**) are:

```
class UIW_MAXIMIZE_BUTTON : public UIW_BUTTON
{
public:
    UIW_MAXIMIZE_BUTTON(void);
    virtual ~UIW_MAXIMIZE_BUTTON(void);
};
```

Inheritance

The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};
```

```
class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_BUTTON : public UI_WINDOW_OBJECT
{
public:
    int depth;
    USHORT btFlags;

    const char *DataGet(void);
    void DataSet(const char *string);
};

class UIW_MAXIMIZE_BUTTON : public UIW_BUTTON;
```

See also The example file **XWGEN.CPP**, which gives a complete example of the UIW_MAXIMIZE_BUTTON class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 27—UIW_BUTTON" of this manual, which describes the base class from which the UIW_MAXIMIZE_BUTTON class is derived.

"Chapter 33—UIW_MINIMIZE_BUTTON" of this manual, which describes an additional class derived from the UIW_BUTTON class.

"Chapter 42—UIW_SYSTEM_BUTTON" of this manual, which describes an additional class derived from the UIW_BUTTON class.

UIW_MAXIMIZE_BUTTON::UIW_MAXIMIZE_BUTTON

Syntax

```
#include <ui_win.hpp>

UIW_MAXIMIZE_BUTTON::UIW_MAXIMIZE_BUTTON(void);
```

Remarks

This constructor returns a pointer to a new UIW_MAXIMIZE_BUTTON class object. The maximize button object always occupies the outer-most right corner space available 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,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        .
        .
        .

2)  // INCORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_MINIMIZE_BUTTON
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_BORDER
        .
        .
        .
```

NOTE: If the maximize button is added to a parent window, it will automatically be destroyed when the parent window is destroyed.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
```

```
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
}
```

UIW_MAXIMIZE_BUTTON::~UIW_MAXIMIZE_BUTTON

Syntax

```
#include <ui_win.hpp>

virtual UIW_MAXIMIZE_BUTTON::
    ~UIW_MAXIMIZE_BUTTON(void);
```

Remarks

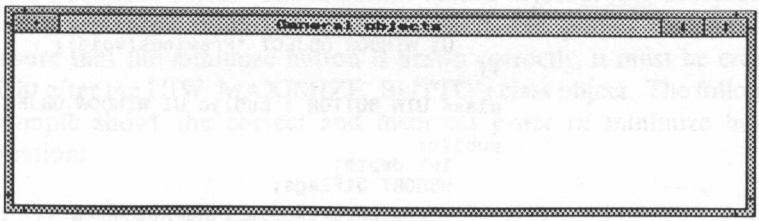
This virtual destructor destroys the class information associated with the UIW_MAXIMIZE_BUTTON object. Care should be taken to only destroy maximize button objects that are not attached to a parent window.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_MAXIMIZE_BUTTON *maxButton = new UIW_MAXIMIZE_BUTTON;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + maxButton
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
    // Remove the maximize button and its parent window.
    *window - maxButton;
    delete maxButton;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the maximize button
    // destructor.
}
```

CHAPTER 33 – UIW_MINIMIZE_BUTTON

Overview The UIW_MINIMIZE_BUTTON class is used to minimize a window. A minimized window is reduced to its smallest representable form. The figures below show graphic and textual implementations of a window with a UIW_MINIMIZE_BUTTON class object (the button with the '↓' character):



```
┌[•]──────────────[ General objects ]──────────────[↓][↑]┐
│                                                       │
│                                                       │
│                                                       │
│                                                       │
└───────────────────────────────────────────────────────┘
```

The public members of the UIW_MINIMIZE_BUTTON class (declared in **UI_WIN.HPP**) are:

```
class UIW_MINIMIZE_BUTTON : public UIW_BUTTON
{
public:
    UIW_MINIMIZE_BUTTON(void);
    virtual ~UIW_MINIMIZE_BUTTON(void);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};
```

```
class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_BUTTON : public UI_WINDOW_OBJECT
{
public:
    int depth;
    USHORT btFlags;

    const char *DataGet(void);
    void DataSet(const char *string);
};

class UIW_MINIMIZE_BUTTON : public UIW_BUTTON;
```

See also The example file **XWGEN.CPP**, which gives a complete example of the UIW_MINIMIZE_BUTTON class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 27—UIW_BUTTON" of this manual, which describes the base class from which the UIW_MINIMIZE_BUTTON class is derived.

"Chapter 32—UIW_MAXIMIZE_BUTTON" of this manual, which describes an additional class derived from the UIW_BUTTON class.

"Chapter 42—UIW_SYSTEM_BUTTON" of this manual, which describes an additional class derived from the UIW_BUTTON class.

UIW_MINIMIZE_BUTTON::UIW_MINIMIZE_BUTTON

Syntax

```
#include <ui_win.hpp>

UIW_MINIMIZE_BUTTON::UIW_MINIMIZE_BUTTON(void);
```

Remarks

This constructor returns a pointer to a new UIW_MINIMIZE_BUTTON class object. The minimize button object always occupies the outer-most right corner space available in the parent window. To ensure that the minimize button is drawn correctly, it must be created 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,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        .
        .
        .

2)  // INCORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_MINIMIZE_BUTTON
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_BORDER
        .
        .
        .
```

NOTE: If the minimize button is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
```

```
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
}
```

UIW_MINIMIZE_BUTTON::~UIW_MINIMIZE_BUTTON

Syntax

```
#include <ui_win.hpp>

virtual UIW_MINIMIZE_BUTTON::
    ~UIW_MINIMIZE_BUTTON(void);
```

Remarks

This virtual destructor destroys the class information associated with the UIW_MINIMIZE_BUTTON object. Care should be taken to only destroy minimize button objects that are not attached to a parent window.

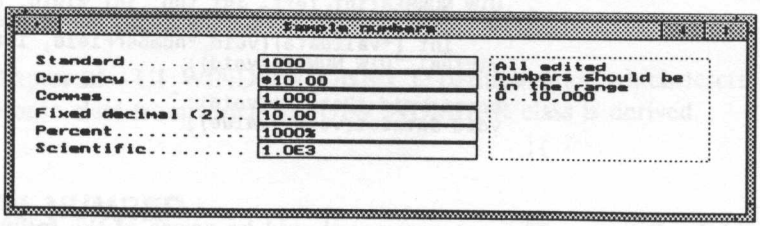
Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_MAXIMIZE_BUTTON *minButton = new UIW_MINIMIZE_BUTTON;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + minButton
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
    // Manually destroy the minimize button and its parent window.
    *window - minButton;
    delete minButton;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the minimize button
    // destructor.
}
```

CHAPTER 34 – UIW_NUMBER

Overview

The UIW_NUMBER class is used to display numeric information to the screen and to collect information, in numeric form, from an end user. The figures below show graphic and textual implementations of a window with several variations of the UIW_NUMBER class object:




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

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

The public members of the UIW_NUMBER class (declared in **UI_WIN.HPP**) are:

```
class UIW_NUMBER : public UI_WINDOW_OBJECT, public UI_EDIT_INFO
{
public:
    USHORT nmFlags;

    UIW_NUMBER(int left, int top, int width, char *value,
        char *range, USHORT nmFlags, USHORT woFlags,
        int (*validate)(void *numberField, int ccode) = NULL);
    UIW_NUMBER(int left, int top, int width, unsigned char *value,
        char *range, USHORT nmFlags, USHORT woFlags,
        int (*validate)(void *numberField, int ccode) = NULL);
    UIW_NUMBER(int left, int top, int width, short *value,
        char *range, USHORT nmFlags, USHORT woFlags,
        int (*validate)(void *numberField, int ccode) = NULL);
    UIW_NUMBER(int left, int top, int width, unsigned short *value,
        char *range, USHORT nmFlags, USHORT woFlags,
        int (*validate)(void *numberField, int ccode) = NULL);
    UIW_NUMBER(int left, int top, int width, int *value,
        char *range, USHORT nmFlags, USHORT woFlags,
        int (*validate)(void *numberField, int ccode) = NULL);
```

```
    UIW_NUMBER(int left, int top, int width, unsigned int *value,
        char *range, USHORT nmFlags, USHORT woFlags,
        int (*validate)(void *numberField, int ccode) = NULL);
    UIW_NUMBER(int left, int top, int width, long *value,
        char *range, USHORT nmFlags, USHORT woFlags,
        int (*validate)(void *numberField, int ccode) = NULL);
    UIW_NUMBER(int left, int top, int width, unsigned long *value,
        char *range, USHORT nmFlags, USHORT woFlags,
        int (*validate)(void *numberField, int ccode) = NULL);
    UIW_NUMBER(int left, int top, int width, float *value,
        char *range, USHORT nmFlags, USHORT woFlags,
        int (*validate)(void *numberField, int ccode) = NULL);
    UIW_NUMBER(int left, int top, int width, double *value,
        char *range, USHORT nmFlags, USHORT woFlags,
        int (*validate)(void *numberField, int ccode) = NULL);
    virtual ~UIW_NUMBER(void);

    const void *DataGet(void);
    void DataSet(void *value);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UI_EDIT_INFO
{
public:
    static void UndoStrategy(short maxObjects, long maxBytes,
        short maxUndos, long maxBytesPerObject,
        short maxUndosPerObject);
};

class UIW_NUMBER :
    public UI_WINDOW_OBJECT, public UI_EDIT_INFO;
```

See also The example file **XWNUMBER.CPP**, which gives a complete example of the UIW_NUMBER class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes the base class from which the UIW_NUMBER class is derived.

UIW_NUMBER::UIW_NUMBER

Syntax #include <ui_win.hpp>

UIW_NUMBER::UIW_NUMBER(int *left*, int *top*, int *width*,
char ****value*, char **range*, USHORT *nmFlags*, USHORT *woFlags*,
int (**validate*)(void *numberField, int ccode) = NULL);
or
UIW_NUMBER::UIW_NUMBER(int *left*, int *top*, int *width*,
unsigned char ****value*, char **range*, USHORT *nmFlags*,
USHORT *woFlags*,
int (**validate*)(void *numberField, int ccode) = NULL);
or
UIW_NUMBER::UIW_NUMBER(int *left*, int *top*, int *width*,
short ****value*, char **range*, USHORT *nmFlags*, USHORT *woFlags*,
int (**validate*)(void *numberField, int ccode) = NULL);
or
UIW_NUMBER::UIW_NUMBER(int *left*, int *top*, int *width*,
unsigned short ****value*, char **range*, USHORT *nmFlags*,
USHORT *woFlags*, int (**validate*)(void *numberField, int ccode)
= NULL);
or
UIW_NUMBER::UIW_NUMBER(int *left*, int *top*, int *width*,
int ****value*, int **range*, USHORT *nmFlags*, USHORT *woFlags*,
int (**validate*)(UIW_NUMBER *object, int ccode) = NULL);
or

UIW_NUMBER::UIW_NUMBER(int *left*, int *top*, int *width*,
unsigned int *value, unsigned int **range*, USHORT *nmFlags*,
USHORT *woFlags*,
int (**validate*)(void *numberField, int ccode) = NULL);
or
UIW_NUMBER::UIW_NUMBER(int *left*, int *top*, int *width*,
long *value, char **range*, USHORT *nmFlags*, USHORT *woFlags*,
int (**validate*)(void *numberField, int ccode) = NULL);
or
UIW_NUMBER::UIW_NUMBER(int *left*, int *top*, int *width*,
unsigned long *value, char **range*, USHORT *nmFlags*,
USHORT *woFlags*,
int (**validate*)(void *numberField, int ccode) = NULL);
or
UIW_NUMBER::UIW_NUMBER(int *left*, int *top*, int *width*,
float *value, char **range*, USHORT *nmFlags*, USHORT *woFlags*,
int (**validate*)(void *numberField, int ccode) = NULL);
or
UIW_NUMBER::UIW_NUMBER(int *left*, int *top*, int *width*,
double *value, char **range*, USHORT *nmFlags*,
USHORT *woFlags*,
int (**validate*)(void *numberField, int ccode) = NULL);

Remarks This constructor returns a pointer to a new UIW_NUMBER class object. The type of number object created depends on the *value* argument.

NOTE: If the number object is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- $left_{in}$ and top_{in} is the starting position of number field within its parent window.
- $width_{in}$ is the width of the number field. (The height of the number field is determined automatically by the UIW_NUMBER class object.)

- *value*$_{in/out}$ is a constructor specific numeric value. 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).

  int—A number whose value is machine dependent.

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

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

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

  float—A single precision floating point number.

  double—A double precision floating point number.

- *range*$_{in}$ is a string that gives all the valid numeric ranges. For example, if a range of "1000..10000" were specified, the UIW_NUMERIC 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_NUMBER class object.

- *nmFlags*$_{in}$ gives information on how to display and interpret the numeric information. The following flags (declared in UI_-WIN.HPP) control the general presentation of a UIW_NUMBER class object:

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

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

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

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

  NMF_NO_FLAGS—Does not associate any special flags with the number object. In this case, the numeric information will be left-justified. This flag should not be used in conjunction with any other NMF flag.

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

  NMF_SCIENTIFIC—Displays the number in scientific format. This flag only has effect on real numeric types. Some example real numbers with the NMF_SCIENTIFIC flag set are: "1.0E3," "4.5E-40" and "1.195E." **NOTE**: 0 exponents are not displayed on a numeric field.

- *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_NUMBER class object:

  WOF_AUTO_CLEAR—Automatically clears the numeric buffer if the end-user positions on to the first character of the number field (from another window field) then presses a key (without having previously pressed any movement or editing keys).

  WOF_BORDER—Draws a border around the number object. In graphics mode, setting this option draws a single line border around the object. In text mode, setting this option draws display brackets (i.e., '[', ']') around the object.

  WOF_INVALID—Sets the initial status of the number field to be "invalid." Invalid numbers fit in the absolute range determined by the object type but do not fulfill all the requirements specified by the program. For example, an unsigned char number may initially be set to 200, but the final number, edited by the end-user, must be in the range "10..100." The initial number in this example fits the absolute requirements of an unsigned char UI_NUMBER 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 number object from allocating a numeric value to store the numeric information. If this flag is set, the programmer must allocate the number (passed as the *value* parameter) that is used by the number object.

  WOF_NO_FLAGS—Does not associate any special window flags with the number object. Setting this flag left-justifies the numeric information. This flag should not be used in conjunction with any other WOF flags.

WOF_NO_INVALID—Prevents the "Leave invalid" option in the error window from being selected by the end-user when an invalid number is entered.

WOF_NO_UNANSWERED—Prevents the "Leave unanswered" option in the error window from being selected by the end-user when an invalid number is entered.

WOF_NON_SELECTABLE—Prevents the number 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 number field to be "unanswered." An unanswered number field is displayed as blank space on the screen.

- *validate*$_{in}$ is a programmer defined function that is called whenever:

  1—a number is entered and the user moves to a different field in the form, or

  2—the user moves to a different window on the screen.

  The *validate* function is <u>not</u> called if the number does not fit the absolute range for its numeric type or if the number is outside the default range (specified by the *range* argument passed in on the UIW_NUMBER constructor). The following arguments are passed to *validate* when a new number is entered:

  numberField$_{in}$—A pointer to the UIW_NUMBER class object or the class object derived from the UIW_NUMBER object base class. This argument must be typecast by the programmer.

  ccode$_{in}$—The logical or system code that caused the validate function to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

  S_CURRENT—The number 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 number is valid for the absolute value of the numeric type and if the number is valid for the programmer defined *range*.

The validate function's *returnValue* should be 0 if the number is valid. Otherwise, the programmer should call the error system with an appropriate error message (see example below) and return -1.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Add a number field to the window.
    int ivalue = 0;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample numbers ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Standard.....", WOF_NO_FLAGS)
        + new UIW_NUMBER(22, 1, 20, &ivalue, "0..10000",
            NMF_NO_FLAGS, WOF_BORDER)
        + new UIW_PROMPT(2, 2, "Currency.....", WOF_NO_FLAGS)
        + new UIW_NUMBER(22, 2, 20, &ivalue, "0..10000",
            NMF_CURRENCY | NMF_DECIMAL(2), WOF_BORDER)
        + new UIW_PROMPT(2, 3, "Commas.....", WOF_NO_FLAGS)
        + new UIW_NUMBER(22, 3, 20, &ivalue, "0..10000",
            NMF_COMMAS, WOF_BORDER);
    .
    .
    .
}
```

UIW_NUMBER::~UIW_NUMBER

Syntax

#include <ui_win.hpp>

virtual UIW_NUMBER::~UIW_NUMBER(void);

Remarks

This virtual destructor destroys the class information associated with the UIW_NUMBER object. Care should be taken to only destroy those number objects that are not attached to a parent window.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a number field to the window.
    int ivalue = 0;
    UIW_NUMBER *numberField = new UIW_NUMBER(22, 1, 20, &ivalue,
        "0..10000", NMF_NO_FLAGS, WOF_BORDER);

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample numbers ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Standard.....", WOF_NO_FLAGS)
        + numberField;
    .
    .
    .
    // Manually destroy the number object and its parent window.
    *window - numberField;
    delete numberField;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the number object
    // destructor.
}
```

UIW_NUMBER::DataGet

Syntax

#include <ui_win.hpp>

const void *UIW_NUMBER::DataGet(void);

Remarks

This function gets the current numeric information associated with the UIW_NUMBER class object. This function returns a constant void pointer to the numeric data. Thus, the contents of the number cannot be directly modified by the programmer.

- *returnValue*$_{out}$ is a constant void pointer to the numeric value.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a number field to the window.
    int ivalue = 0;
    UIW_NUMBER *numberField = new UIW_NUMBER(22, 1, 20, &ivalue,
        "0..10000", NMF_NO_FLAGS, WOF_BORDER);

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample numbers ", WOF_JUSTIFY_CENTER)
```

```
        + new UIW_PROMPT(2, 1, "Standard.....", WOF_NO_FLAGS)
        + numberField;
    .
    .
    .
    // Get the numeric contents of the field.
    int newValue = *(int *)numberField->DataGet();
    .
    .
    .
}
```

UIW_NUMBER::DataSet

Syntax

#include <ui_win.hpp>

void UIW_NUMBER::DataSet(void *_value_);

Remarks

This function resets the current numeric information associated with the UIW_NUMBER class object or tells the class object that key flags, associated with the number 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_NUMBER class object is destroyed. Otherwise, the information associated with this argument is copied by the UIW_NUMBER 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

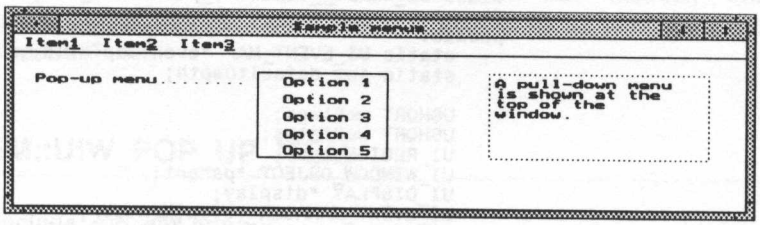
```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a number field to the window.
    int ivalue = 0;
    UIW_NUMBER *numberField = new UIW_NUMBER(22, 1, 20, &ivalue,
        "0..10000", NMF_NO_FLAGS, WOF_BORDER);

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample numbers ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Standard.....", WOF_NO_FLAGS)
        + numberField;
```

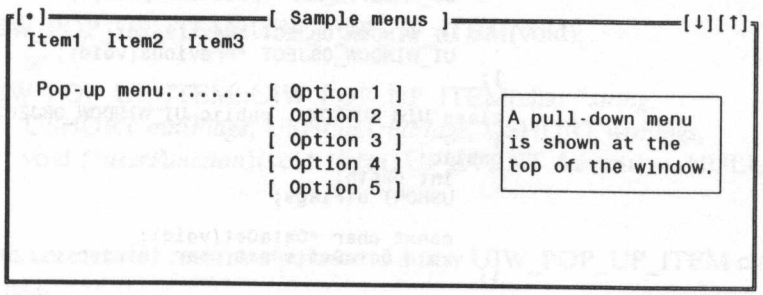
```
        .
        .
        .
        // Reset the numeric information for the field.
        int newValue = 100;
        numberField->DataSet(&newValue);
        .
        .
        .
}
```

CHAPTER 35 – UIW_POP_UP_ITEM

Overview

The UIW_POP_UP_ITEM class is used to display and select options associated with a list of menu items. The figures below show graphic and textual implementations of UIW_POP_UP_ITEM objects (shown as "Option 1" through "Option 5" on the pop-up menu):





The public members of the UIW_POP_UP_ITEM class (declared in **UI_WIN.HPP**) are:

```
class UIW_POP_UP_ITEM : public UIW_BUTTON
{
public:
    USHORT mniFlags;

    UIW_POP_UP_ITEM(char *string, USHORT mniFlags,
        USHORT btFlags, USHORT woFlags,
        void (*userFunction)(void *item, UI_EVENT &event));
    UIW_POP_UP_ITEM(void);
    virtual ~UIW_POP_UP_ITEM(void);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_BUTTON : public UI_WINDOW_OBJECT
{
public:
    int depth;
    USHORT btFlags;

    const char *DataGet(void);
    void DataSet(const char *string);
};

class UIW_POP_UP_ITEM : public UIW_BUTTON;
```

See also The example file **XWMENU.CPP**, which gives a complete example of the UIW_POP_UP_ITEM class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 27—UIW_BUTTON" of this manual, which describes the base class from which the UIW_POP_UP_ITEM class is derived.

"Chapter 36—UIW_POP_UP_MENU" of this manual, which describes a higher-level class object that uses pop-up menu items in its menu.

"Chapter 39—UIW_PULL_DOWN_ITEM" of this manual, which describes a higher-level class object that uses pop-up menu items in its pop-up menu.

"Chapter 42—UIW_SYSTEM_BUTTON" of this manual, which describes a button class object that uses pop-up menu items in its pop-up menu.

UIW_POP_UP_ITEM::UIW_POP_UP_ITEM

Syntax

```
#include <ui_win.hpp>

UIW_POP_UP_ITEM::UIW_POP_UP_ITEM(void);
   or
UIW_POP_UP_ITEM::UIW_POP_UP_ITEM(char *string,
   USHORT mniFlags, USHORT btFlags, USHORT woFlags,
   void (*userFunction)(void *item, UI_EVENT &event) = NULL);
```

Remarks

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

NOTE: If the pop-up menu item is attached to a parent menu, it will automatically be destroyed when the parent menu is destroyed.

The first constructor takes no arguments. It places a menu item separator (horizontal line) in the parent menu.

The second constructor takes the following arguments:

- *string*$_{in}$ is a pointer to the string information associated with the pull-down item. This pointer is used by the pull-down item if the WOF_NO_ALLOCATE_DATA flag is set. Otherwise, the string is copied into a buffer allocated by the UIW_PULL_DOWN_ITEM class object.

- *mniFlags*$_{in}$ gives information on how to display the item. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of the pop-up item:

  MNIF_DUAL_MONITOR—The menu item is only selectable when the system is running in a dual-monitor state.

  MNIF_MAXIMIZE—The menu item is only selectable when the parent window can be maximized.

  MNIF_MINIMIZE—The menu item is only selectable when the parent window can be minimized.

  MNIF_MOVE—The menu item is only selectable when the parent window can be moved.

  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 flag.

  MNIF_RESTORE—The menu item is only selectable when the parent window is in a maximized or minimized state.

  MNIF_SEPARATOR—The menu item is a separator (it has no text information associated with it).

  MNIF_SIZE—The menu item is only selectable when the parent window can be sized.

- *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_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).

  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_POP_UP_ITEM 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 menu item is selected.

- *userFunction*$_{in}$ is a programmer-defined function that is called whenever the menu 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 menu item and the supplied *userFunction* is NULL. These default functions send messages to maximize, minimize, move, restore, or size the parent window.

  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 item is selected:

  item$_{in}$ is a pointer to the UIW_POP_UP_ITEM class object or class object derived from the UIW_POP_UP_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 function. Since this argument is a copy of the original event, it may be changed by the programmer.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
```

```
            + &(*new UIW_SYSTEM_BUTTON
                + new UIW_POP_UP_ITEM("~Restore", MNIF_RESTORE,
                    BTF_NO_TOGGLE, WOF_NO_FLAGS)
                + new UIW_POP_UP_ITEM("~Move", MNIF_MOVE,
                    BTF_NO_TOGGLE, WOF_NO_FLAGS)
                + new UIW_POP_UP_ITEM("~Size", MNIF_SIZE,
                    BTF_NO_TOGGLE, WOF_NO_FLAGS))
            + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
        .
        .
        .
}

ExampleFunction2()
{
    // Create a window with a pop-up menu.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample menus ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Pop-up menu.....", WOF_NO_FLAGS)
        + &(*new UIW_POP_UP_MENU(22, 1, MNF_SELECT_ONE, WOF_BORDER,
            WOAF_NO_FLAGS)
            + new UIW_POP_UP_ITEM(" Option 1 ", 0, MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS)
            + new UIW_POP_UP_ITEM(" Option 2 ", 0, MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS));
        .
        .
        .
}
```

UIW_POP_UP_ITEM::~UIW_POP_UP_ITEM

Syntax

```
#include <ui_win.hpp>

virtual UIW_POP_UP_ITEM::~UIW_POP_UP_ITEM(void);
```

Remarks

This virtual destructor destroys the class information associated with the UIW_POP_UP_ITEM object. Care should be taken to only destroy pop-up items that are not attached to a parent pop-up menu.

Example

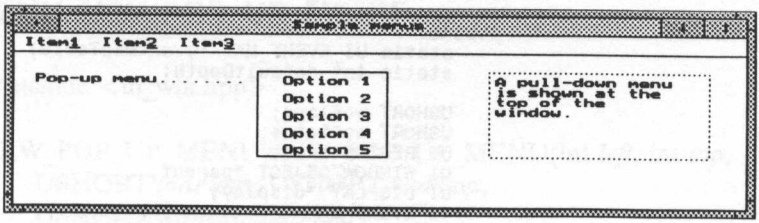
```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_MINIMIZE_BUTTON *sysButton = new UIW_SYSTEM_BUTTON;

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_DESTROY);
```

```
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + sysButton
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);

    // Add system options to the window.
    UIW_POP_UP_ITEM *dualMonitor = new UIW_POP_UP_ITEM(
        "~Switch window", 0, MNIF_NO_FLAGS, BTF_NO_TOGGLE,
        WOF_NO_FLAGS);
    *sysButton
        + dualMonitor
        + new UIW_POP_UP_ITEM("~Restore", MNIF_RESTORE,
            BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM("~Move", MNIF_MOVE,
            BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM("~Size", MNIF_SIZE,
            BTF_NO_TOGGLE, WOF_NO_FLAGS);
    .
    .
    .
    // Remove the dual Monitor option if no longer supported.
    extern int DualMonitorSupport(void);
    if (!DualMonitorSupport())
        *sysButton - dualMonitor;
    .
    .
    .
}
```

CHAPTER 36 – UIW_POP_UP_MENU

Overview

The UIW_POP_UP_MENU class is used as the control structure for selectable menu items. The figures below show graphic and textual implementations of the UIW_POP_UP_MENU class object with several pop-up menu items (shown as "Option 1" through "Option 5"):




```
[•]──────────[ Sample menus ]──────────[↓][↑]
 Item1  Item2  Item3

  Pop-up menu..... [ Option 1 ]
                   [ Option 2 ]   ┌──────────────────────┐
                   [ Option 3 ]   │A pull-down menu      │
                   [ Option 4 ]   │is shown at the       │
                   [ Option 5 ]   │top of the window.    │
                                  └──────────────────────┘
```

The public members of the UIW_POP_UP_MENU class (declared in **UI_WIN.HPP**) are:

```
class UIW_POP_UP_MENU : public UIW_WINDOW
{
public:
    USHORT mnFlags;

    UIW_POP_UP_MENU(int left, int top, USHORT mnFlags,
        USHORT woFlags, USHORT woAdvancedFlags);

    UIW_POP_UP_ITEM *First(void);
    UIW_POP_UP_ITEM *Last(void);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_WINDOW : public UI_WINDOW_OBJECT
{
public:
    void Add(UI_WINDOW_OBJECT *object);
    UI_WINDOW_OBJECT *First(void);
    UI_WINDOW_OBJECT *Last(void)
    void Subtract(UI_WINDOW_OBJECT *object);

    UIW_WINDOW &operator + (void *object);
    UIW_WINDOW &operator - (void *object);
};
class UIW_POP_UP_MENU : public UIW_WINDOW;
```

See also The example file **XWMENU.CPP**, which gives a complete example of the UIW_POP_UP_MENU class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 35—UIW_POP_UP_ITEM" of this manual, which describes the menu items used with the pop-up menu.

"Chapter 39—UIW_PULL_DOWN_ITEM" of this manual, which describes a menu item that uses a pop-up menu to display its sub-options.

"Chapter 42—UIW_SYSTEM_BUTTON" of this manual, which uses a pop-up menu to display system options.

UIW_POP_UP_MENU::UIW_POP_UP_MENU

Syntax #include <ui_win.hpp>

UIW_POP_UP_MENU::UIW_POP_UP_MENU(int *left*, int *top*,
USHORT *mnFlags*, USHORT *woFlags*,
USHORT *woAdvancedFlags*);

Remarks This constructor returns a pointer to a new UIW_POP_UP_MENU class object.

NOTE: If the pop-up menu is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- $left_{in}$ and top_{in} is 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.)

- $mnFlags_{in}$ 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:

  MNF_NO_FLAGS—Does not associate any special flags with the pop-up menu. This flag should not be used in conjunction with any other MNF flag.

  MNF_SELECT_ONE—Prevents more than one menu item from being selected from the menu.

  MNF_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.

  WOF_NO_FLAGS—Does not associate any special flags with the menu. This flag should not be used in conjunction with any other WOF flag.

  WOF_NON_FIELD_REGION—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. Otherwise, this advanced flag should only be used when attaching a pop-up menu directly to the screen display.

  WOF_NON_SELECTABLE—The menu 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 menu items.

- *woAdvancedFlags*$_{in}$ are flags that determine the advanced operation of the pop-up menu.

  WOAF_NO_FLAGS—Does not associated any special advanced flags with the menu. Setting this flag allows the user to move, size and interact with the menu in a normal fashion. This flag should not be used in conjunction with any other WOAF flag.

  WOAF_TEMPORARY—The menu only occupies the screen temporarily. Once another window is selected from the screen, 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.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a pull-down menu with menu items.
    UIW_PULL_DOWN_MENU *menu = UIW_POP_UP_MENU(22, 1,
        MNF_SELECT_ONE, WOF_BORDER, WOAF_NO_FLAGS);
    menu
        + new UIW_POP_UP_ITEM(" Option 1 ", MNIF_NO_FLAGS,
          BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM(" Option 2 ", MNIF_NO_FLAGS,
          BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM(" Option 3 ", MNIF_NO_FLAGS,
          BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM(" Option 4 ", MNIF_NO_FLAGS,
          BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM(" Option 5 ", MNIF_NO_FLAGS,
          BTF_NO_TOGGLE, WOF_NO_FLAGS));

    // Add the pop-up menu field to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_PROMPT(2, 1, "Pop-up menu.....", WOF_NO_FLAGS)
        + menu;
    .
    .
    .
}
```

UIW_POP_UP_MENU::~UIW_POP_UP_MENU

Syntax

#include <ui_win.hpp>

virtual UIW_POP_UP_MENU::~UIW_POP_UP_MENU(void);

Remarks

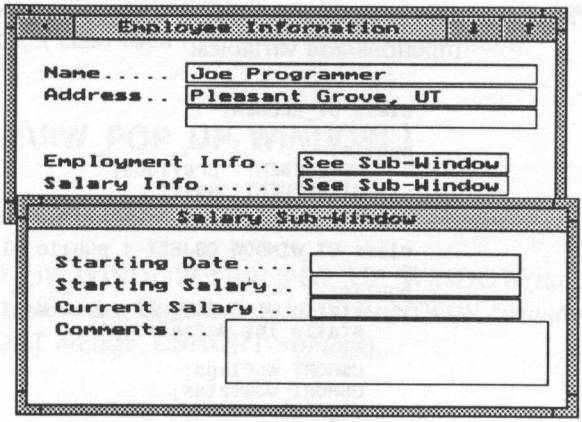
This virtual destructor destroys the class information associated with the UIW_POP_UP_MENU object. Care should be taken to only destroy those pop-up menus that are not attached to a parent window.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a pull-down menu with menu items.
    UIW_PULL_DOWN_MENU *menu = UIW_POP_UP_MENU(22, 1,
        MNF_SELECT_ONE, WOF_BORDER, WOAF_NO_FLAGS);
    menu
        + new UIW_POP_UP_ITEM(" Option 1 ", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM(" Option 2 ", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM(" Option 3 ", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM(" Option 4 ", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM(" Option 5 ", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS));

    // Add the pop-up menu field to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_PROMPT(2, 1, "Pop-up menu.....", WOF_NO_FLAGS)
        + menu;
    .
    .
    .
    // Manually remove the menu from the window.
    *window - menu;
    delete menu;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the pop-up menu destructor.
}
```

CHAPTER 37 – UIW_POP_UP_WINDOW

Overview The UIW_POP_UP_WINDOW class is used to show additional information that cannot fit in a parent window, or when the presentation of the information is enhanced by a separate window. The figures below show the graphic and textual implementations of a UIW_POP_UP_WINDOW class object (shown as the "Salary Sub-window"):




```
[•]—[Employee Information]———[↓][↑]
  Name..... [Joe Programmer      ]
  Address.. [Pleasant Grove, UT  ]
            [                    ]

  Employment Info.. [See Sub-Window]
  Salary Info..... [See Sub-Window]
        ———[Salary Sub-Window]———
    Starting Date.... [          ]
    Starting Salary.. [          ]
    Current Salary... [          ]
    Comments...
```

The public members of the UIW_POP_UP_WINDOW class (declared in **UI_WIN.HPP**) are:

```
class UIW_POP_UP_WINDOW : public UIW_BUTTON
{
public:
    UIW_POP_UP_WINDOW(int left, int top, int width,
        char *string, UIW_WINDOW *window, USHORT btFlags,
        USHORT woFlags);
    virtual ~UIW_POP_UP_WINDOW(void);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_BUTTON : public UI_WINDOW_OBJECT
{
public:
    int depth;
    USHORT btFlags;

    const char *DataGet(void);
    void DataSet(const char *string);
};

class UIW_POP_UP_WINDOW : public UIW_BUTTON;
```

See also The example file **XWMISC.CPP**, which gives a complete example of the UIW_POP_UP_WINDOW class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 27—UIW_BUTTON" of this manual, which describes the base class from which the UIW_POP_UP_WINDOW class is derived.

"Chapter 46—UIW_WINDOW" of this manual, which describes the window object class used to display the pop-up window.

UIW_POP_UP_WINDOW::UIW_POP_UP_WINDOW

Syntax #include <ui_win.hpp>

UIW_POP_UP_WINDOW::UIW_POP_UP_WINDOW(int *left*, int *top*, int *width*, char **string*, UIW_WINDOW **window*, USHORT *btFlags*, USHORT *woFlags*);

Remarks This constructor returns a pointer to a new UIW_POP_UP_WINDOW class object. The class object is not the window but an identifying string that fits on one line in the parent window. The arguments described below apply to the string information associated with the pop-up window. (The arguments for the window are specified when the *window* argument is constructed.)

NOTE: If the pop-up window is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- $left_{in}$ and top_{in} is the starting position of the pop-up window's string description within the parent window.

- $width_{in}$ is the width of the string description. (The height of the string is determined automatically by the UIW_POP_UP_WINDOW class object.)

- $string_{in}$ is a pointer to the associated string information. This pointer is used by the pop-up window object if the WOF_NO_-

ALLOCATE_DATA flag is set. Otherwise, the string is copied into a buffer allocated by the UIW_POP_UP_WINDOW class object.

- *btFlags*$_{in}$ gives information on how to display and select the pop-up window. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of a UIW_POP_UP_WINDOW class object:

  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_POP_UP_ITEM 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.

- *woFlags*$_{in}$ are flags (common to all window objects) that determine the general operation of the pop-up window selection item. The following flags (declared in **UI_WIN.HPP**) control the general presentation of, and interaction with, a UIW_POP_UP_WINDOW class object:

  WOF_BORDER—Draws a border around the string information. In graphics mode, setting this option draws a single line border around the object. In text mode, setting this option draws display braces (i.e., '[' ']') around the object.

  WOF_JUSTIFY_CENTER—Center-justifies the *string* information associated with the pop-up window.

  WOF_JUSTIFY_RIGHT—Right-justifies the *string* information associated with the pop-up window.

  WOF_NO_ALLOCATE_DATA—Prevents the pop-up window from allocating a string buffer that stores the item's string information. If this flag is set, the programmer must allocate

the string buffer (passed as the *string* parameter) that is used by the pop-up window object.

WOF_NO_FLAGS—Does not associate any special flags with the pop-up window object. In this case, the string information will be left-justified. This flag should not be used in conjunction with any other WOF flag.

WOF_NON_SELECTABLE—The pop-up window cannot be selected.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create the pop-up window.
    UIW_WINDOW *popup2 = new UIW_WINDOW(0, 0, 40, 10,
       WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *popup2
        + new UIW_BORDER
        + new UIW_TITLE("Salary Sub-Window", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Starting Date....", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 2, "Starting Salary..", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 3, "Current Salary...", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 4, "Comments...", WOF_NO_FLAGS)
        + new UIW_DATE(20, 1, 15, &UI_DATE(), "",
          DTF_NO_FLAGS, WOF_BORDER)
        + new UIW_STRING(20, 2, 15, "", 64,
          STF_NO_FLAGS, WOF_BORDER)
        + new UIW_STRING(20, 3, 15, "", 64,
          STF_NO_FLAGS, WOF_BORDER)
        + new UIW_TEXT(14, 4, 23, 3, "",
          1024, TXF_NO_FLAGS, WOF_BORDER);

    // Attach the pop-up window to a parent window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
       WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_TITLE("Employee Information", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Name.....", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 2, "Address..", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 5, "Employment Info..", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 6, "Salary Info......", WOF_NO_FLAGS)
    + new UIW_STRING(12, 1, 25, "Joe Programmer", 64,
          STF_NO_FLAGS, WOF_BORDER)
        + new UIW_STRING(12, 2, 25, "Pleasant Grove, UT", 64,
          STF_NO_FLAGS, WOF_BORDER)
        + new UIW_STRING(12, 3, 25, "", 64, STF_NO_FLAGS,
          WOF_BORDER)
        + new UIW_POP_UP_WINDOW(20, 5, 15, "See Sub-Window",
          popup1, BTF_NO_FLAGS, WOF_NO_FLAGS);
        + new UIW_POP_UP_WINDOW(20, 6, 15, "See Sub-Window",
          popup2, BTF_NO_FLAGS, WOF_NO_FLAGS);
        .
        .
        .
}
```

UIW_POP_UP_WINDOW::~UIW_POP_UP_WINDOW

Syntax

```
#include <ui_win.hpp>

virtual UIW_POP_UP_WINDOW::
    ~UIW_POP_UP_WINDOW(void);
```

Remarks

This virtual destructor destroys the class information associated with the UIW_POP_UP_WINDOW object. Care should be taken to only destroy pop-up window objects that are not attached to a parent window.

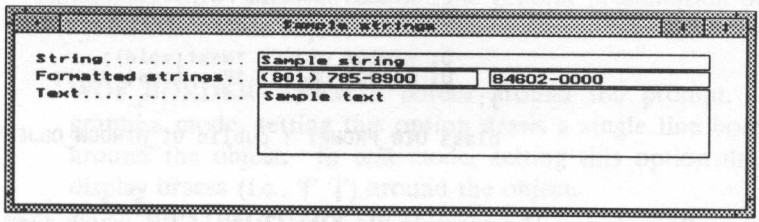
Example

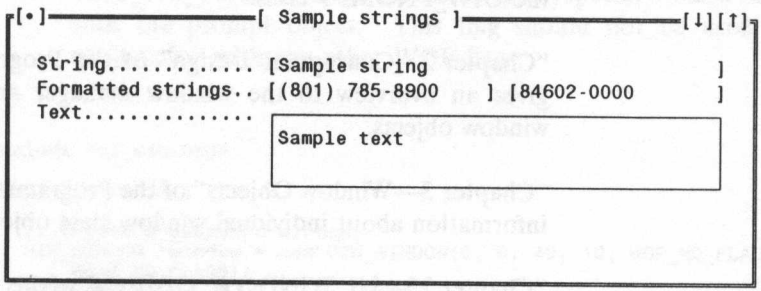
```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create the pop-up window.
    UIW_WINDOW *popup2 = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *popup2
        + new UIW_BORDER
        + new UIW_TITLE("Salary Sub-Window", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Starting Date....", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 2, "Starting Salary..", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 3, "Current Salary...", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 4, "Comments...", WOF_NO_FLAGS)
        + new UIW_DATE(20, 1, 15, &UI_DATE(), "",
            DTF_NO_FLAGS, WOF_BORDER)
        + new UIW_STRING(20, 2, 15, "", 64,
            STF_NO_FLAGS, WOF_BORDER)
        + new UIW_STRING(20, 3, 15, "", 64,
            STF_NO_FLAGS, WOF_BORDER)
        + new UIW_TEXT(14, 4, 23, 3, "",
            1024, TXF_NO_FLAGS, WOF_BORDER);

    // Attach the pop-up window to a parent window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_TITLE("Employee Information", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Name.....", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 2, "Address..", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 5, "Employment Info..", WOF_NO_FLAGS)
        + new UIW_PROMPT(2, 6, "Salary Info......", WOF_NO_FLAGS)
        + new UIW_STRING(12, 1, 25, "Joe Programmer", 64,
            STF_NO_FLAGS, WOF_BORDER)
        + new UIW_STRING(12, 2, 25, "Pleasant Grove, UT", 64,
            STF_NO_FLAGS, WOF_BORDER)
        + new UIW_STRING(12, 3, 25, "", 64, STF_NO_FLAGS,
            WOF_BORDER)
        + new UIW_POP_UP_WINDOW(20, 5, 15, "See Sub-Window",
            popup1, BTF_NO_FLAGS, WOF_NO_FLAGS);
        + new UIW_POP_UP_WINDOW(20, 6, 15, "See Sub-Window",
            popup2, BTF_NO_FLAGS, WOF_NO_FLAGS);
    .
    .
    .
```

```
    // Manually destroy the popup2 window and its parent window.
    // The first popup window is destroyed with "delete window."
    *window - popup2;
    delete popup2;
    delete window.
    // We could have just called "delete window."  Its destructor
    // would have automatically called the second pop-up window
    // destructor also.
}
```

CHAPTER 38 – UIW_PROMPT

Overview The UIW_PROMPT class is used to provide lead information about another window object. The pictures below show graphic and textual implementations of UIW_PROMPT objects (the fields with the ".." characters):





The public members of the UIW_PROMPT class (declared in **UI_WIN.HPP**) are:

```
class UIW_PROMPT : public UI_WINDOW_OBJECT
{
public:
    UIW_PROMPT(int left, int top, const char *prompt,
        USHORT woFlags);
    virtual ~UIW_PROMPT(void);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};
```

```
class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_PROMPT : public UI_WINDOW_OBJECT;
```

See also The example file **XWSTRING.CPP**, which gives a complete example of the UIW_PROMPT class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes the base class from which the UIW_PROMPT class is derived.

UIW_PROMPT::UIW_PROMPT

Syntax #include <ui_win.hpp>

UIW_PROMPT::UIW_PROMPT(int *left*, int *top*, const char **prompt*, USHORT *woFlags*);

Remarks This constructor returns a pointer to a new UIW_PROMPT object.

NOTE: If the prompt object is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- *left*$_{in}$ and *top*$_{in}$ is the starting position of the prompt field within its parent window.

- *prompt*$_{in}$ is the string representation of the prompt.

- *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 border around the prompt. In graphics mode, setting this option draws a single line border around the object. In text mode, setting this option draws display braces (i.e., '[' ']') around the 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>

ExampleFunction1()
{
    // Create a standard window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "String.............", WOF_NO_FLAGS)
        + new UIW_STRING(22, 1, 41, "Sample string", 256,
            STF_NO_FLAGS, WOF_BORDER)
        + new UIW_PROMPT(2, 2, "Formatted strings..", WOF_NO_FLAGS)
        + new UIW_FORMATTED_STRING(22, 2, 20, "8017858900",
            "LNNNLLNNNLXXXX", "(...) ...-....", WOF_BORDER)
        + new UIW_FORMATTED_STRING(43, 2, 20, "846020000",
            "NNNNNLNNNN", "....-....", WOF_BORDER)
        + new UIW_PROMPT(2, 3, "Text.............", WOF_NO_FLAGS)
        + new UIW_TEXT(22, 3, 41, 4, "Sample text", 1028,
            TXF_NO_FLAGS, WOF_BORDER);
    .
    .
    .
}
```

UIW_PROMPT::~UIW_PROMPT

Syntax #include <ui_win.hpp>

virtual UIW_PROMPT::~UIW_PROMPT(void);

Remarks This virtual destructor destroys the class information associated with the UIW_PROMPT object. Care should be taken to only destroy those prompt objects that are not attached to a parent window.

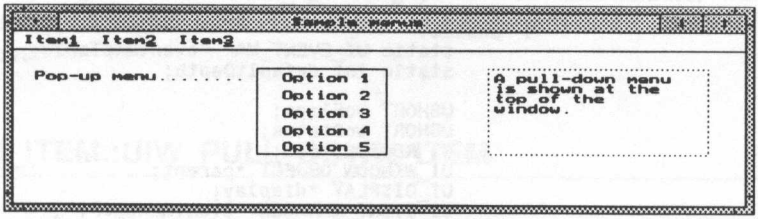
Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a prompt to the window.
    UIW_PROMPT *prompt = new UIW_PROMPT(2, 3,
        "Text.............", WOF_NO_FLAGS);
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + prompt
        + new UIW_TEXT(22, 3, 41, 4, "Sample text", 1028,
            TXF_NO_FLAGS, WOF_BORDER);
    .
    .
    .
    // Manually destroy the prompt object and its parent window.
    *window - prompt;
    delete prompt;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the prompt object
    // destructor.
}
```

CHAPTER 39 – UIW_PULL_DOWN_ITEM

Overview

The UIW_PULL_DOWN_ITEM class is used as the first-level selection within a pull-down menu. The figures below show graphic and textual implementations of UIW_PULL_DOWN_ITEM objects within a pull-down menu (shown as "Item1," "Item2" and "Item3"):




```
[•]──────────[ Sample menus ]──────────[↓][↑]
 Item1  Item2  Item3

  Pop-up menu..... [ Option 1 ]
                   [ Option 2 ]
                   [ Option 3 ]
                   [ Option 4 ]
                   [ Option 5 ]
```

A pull-down menu is shown at the top of the window.

The public members of the UIW_PULL_DOWN_ITEM class (declared in **UI_WIN.HPP**) are:

```
class UIW_PULL_DOWN_ITEM : public UIW_BUTTON
{
public:
    UIW_PULL_DOWN_ITEM(char *string, USHORT mnFlags,
        void (*userFunction)(void *object, UI_EVENT &event));
    virtual ~UIW_PULL_DOWN_ITEM(void);

    void Add(UIW_POP_UP_ITEM *item);
    void Subtract(UIW_POP_UP_ITEM *item);

    UIW_PULL_DOWN_ITEM &operator + (void *object);
    UIW_PULL_DOWN_ITEM &operator - (void *object);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_BUTTON : public UI_WINDOW_OBJECT
{
public:
    int depth;
    USHORT btFlags;

    const char *DataGet(void);
    void DataSet(const char *string);
};

class UIW_PULL_DOWN_ITEM : public UIW_BUTTON;
```

See also The example file **XWMENU.CPP**, which gives a complete example of the UIW_PULL_DOWN_ITEM class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 27—UIW_BUTTON" of this manual, which describes the base class from which the UIW_PULL_DOWN_ITEM class is derived.

"Chapter 35—UIW_POP_UP_ITEM" of this manual, which describes a menu item class used by the pull-down item.

"Chapter 37—UIW_POP_UP_MENU" of this manual, which describes the pop-up menu class used by the pull-down item to display additional control options.

"Chapter 36—UIW_PULL_DOWN_MENU" of this manual, which describes a higher-level class object that uses pull-down items in its menu.

UIW_PULL_DOWN_ITEM::UIW_PULL_DOWN_ITEM

Syntax

```
#include <ui_win.hpp>

UIW_PULL_DOWN_ITEM::UIW_PULL_DOWN_ITEM(
    char *string, USHORT mnFlags,
    void (*userFunction)(void *item, UI_EVENT &event) = NULL);
```

Remarks This constructor returns a pointer to a new UIW_PULL_DOWN_ITEM class object.

NOTE: If the pull-down item is attached to a parent menu, it will automatically be destroyed when the parent menu is destroyed.

- *string*$_{in}$ is a pointer to the string information associated with the pull-down item. This pointer is used by the pull-down item if the WOF_NO_ALLOCATE_DATA flag is set. Otherwise, the string is copied into a buffer allocated by the UIW_PULL_DOWN_ITEM class object.

- *mnFlags*$_{in}$ gives information on how to display the item's pull-down menu. The following flags (declared in **UI_WIN.HPP**) control the general presentation and operation of the item's pull-down menu:

  MNF_NO_FLAGS—Does not associate any special flags with the pull-down menu. This flag should not be used in conjunction with any other MNF flag.

MNF_SELECT_ONE—Prevents more than one menu item from being selected.

MNF_AUTO_SORT—Automatically sorts the menu items in alphabetical order.

- *userFunction*$_{in}$ is a programmer-defined function that is called whenever the pull-down item is selected.

  A default function is provided if this argument is NULL. This function brings up the pop-up window information associated with the pull-down item. This argument, therefore, should only be set if you want 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:

  item$_{in}$ is a pointer to the UIW_PULL_DOWN_ITEM class object or 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.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a pull-down menu with menu items.
    UIW_PULL_DOWN_MENU *menu = new UIW_PULL_DOWN_MENU(0,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    menu
        + &(*new UIW_PULL_DOWN_ITEM(" Item~1 ", MNF_NO_FLAGS,0)
            + new UIW_POP_UP_ITEM("Option 1.1", MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS)
            + new UIW_POP_UP_ITEM("Option 1.2", MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS))
        + &(*new UIW_PULL_DOWN_ITEM(" Item~2 ", MNF_NO_FLAGS,0)
            + new UIW_POP_UP_ITEM("Option 2.1",
                MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS))
        + new UIW_PULL_DOWN_ITEM(" Item~3 ", MNF_NO_FLAGS,0);
```

```
    // Attach the menu to a window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample menus ", WOF_JUSTIFY_CENTER)
        + menu;
    .
    .
    .
}
```

UIW_PULL_DOWN_ITEM::~UIW_PULL_DOWN_ITEM

Syntax

```
#include <ui_win.hpp>

virtual UIW_PULL_DOWN_ITEM::
    ~UIW_PULL_DOWN_ITEM(void);
```

Remarks

This virtual destructor destroys the class information associated with the UIW_PULL_DOWN_ITEM object. Care should be taken to only destroy those pull-down items that are not attached to a parent pull-down menu.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a pull-down menu with menu items.
    UIW_PULL_DOWN_MENU *menu = new UIW_PULL_DOWN_MENU(0,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);

    UIW_PULL_DOWN_ITEM *item1 = new UIW_PULL_DOWN_ITEM(" Item~1 ",
        MNF_NO_FLAGS, 0);
    *item1
        + new UIW_POP_UP_ITEM("Option 1.1", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS);
        + new UIW_POP_UP_ITEM("Option 1.2", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS);

    UIW_PULL_DOWN_ITEM *item2 = new UIW_PULL_DOWN_ITEM(" Item~2 ",
        MNF_NO_FLAGS, 0);
    *item2
        + new UIW_POP_UP_ITEM("Option 2.1", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS);

    // Attach the menu to a window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_TITLE(" Sample menus ", WOF_JUSTIFY_CENTER)
        + &(*menu + item1 + item2);
    .
    .
    .
```

```
    // Manually destroy the menu items, being careful to destroy
    // the objects in the proper order.
    *menu - item1 - item2;
    *window - menu;
    delete menu;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the text menu item
    // destructors.
}
```

UIW_PULL_DOWN_ITEM::Add

Syntax

```
#include <ui_win.hpp>

void UIW_PULL_DOWN_ITEM::Add(
    UIW_POP_UP_ITEM *item);
```

Remarks

This function adds a new pop-up menu item to the UIW_PULL_-DOWN_ITEM class object. (The pop-up menu item is displayed when the pull-down item is selected.)

- $item_{in}$ is a pointer to the object to be added to the pull-down items associated pop-up menu. The new item must be a UIW_POP_UP_-ITEM class object or a class object derived from the UIW_POP_-UP_ITEM base class.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a pull-down menu with menu items.
    UIW_PULL_DOWN_MENU *menu = new UIW_PULL_DOWN_MENU(0,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);

    UIW_PULL_DOWN_ITEM *item1 = new UIW_PULL_DOWN_ITEM(" Item~1 ",
        MNF_NO_FLAGS, 0);
    item1->Add(new UIW_POP_UP_ITEM("Option 1.1", MNIF_NO_FLAGS,
        BTF_NO_TOGGLE, WOF_NO_FLAGS));
    item1->Add(new UIW_POP_UP_ITEM("Option 1.2", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS));

    UIW_PULL_DOWN_ITEM *item2 = new UIW_PULL_DOWN_ITEM(" Item~2 ",
        MNF_NO_FLAGS, 0);
    UIW_POP_UP_ITEM *item21 = new UIW_POP_UP_ITEM("Option 2.1",
        MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS);
    item2->Add(item21);
```

```
    // Attach the menu to a window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_TITLE(" Sample menus ", WOF_JUSTIFY_CENTER)
        + &(*menu + item1 + item2);
    .
    .
    .
}
```

UIW_PULL_DOWN_ITEM::Subtract

Syntax

```
#include <ui_win.hpp>

void UIW_PULL_DOWN_ITEM::Subtract(
    UIW_POP_UP_ITEM *object);
```

Remarks

This function removes a pop-up menu item from the UIW_PULL_-DOWN_ITEM object. This function does not call the destructor associated with the menu item.

- *item*$_{in}$ is a pointer to the item to be removed from the current menu item's list of pop-up menu items. This argument must be a UIW_-POP_UP_ITEM class object or a class object derived from the UIW_POP_UP_ITEM base class.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a pull-down menu with menu items.
    UIW_PULL_DOWN_MENU *menu = new UIW_PULL_DOWN_MENU(0,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);

    UIW_PULL_DOWN_ITEM *item1 = new UIW_PULL_DOWN_ITEM(" Item 1 ",
        MNF_NO_FLAGS, 0);
    item1->Add(new UIW_POP_UP_ITEM("Option 1.1", MNIF_NO_FLAGS,
        BTF_NO_TOGGLE, WOF_NO_FLAGS));
    item1->Add(new UIW_POP_UP_ITEM("Option 1.2", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS));

    UIW_PULL_DOWN_ITEM *item2 = new UIW_PULL_DOWN_ITEM(" Item 2 ",
        MNF_NO_FLAGS, 0);
    UIW_POP_UP_ITEM *item21 = new UIW_POP_UP_ITEM("Option 2.1",
        MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS);
    item2->Add(item21);
```

```
    // Attach the menu to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_TITLE(" Sample menus ", WOF_JUSTIFY_CENTER)
        + &(*menu + item1 + item2);
    .
    .
    .
    // Remove the "Option 2.1" item.
    item2->Subtract(item21);
    .
    .
    .
}
```

UIW_PULL_DOWN_ITEM::operator +

Syntax

```
#include <ui_win.hpp>

UIW_PULL_DOWN_ITEM &UIW_PULL_DOWN_ITEM::
    operator + (UIW_POP_UP_ITEM *item);
```

Remarks

This overload operator adds a pop-up menu item to the UIW_PULL_DOWN_ITEM class object. This operator overload is equivalent to calling the **UIW_PULL_DOWN_ITEM::Add** routine, except that it allows the chaining of pop-up menu item additions to the UIW_PULL_DOWN_ITEM object.

- *returnValue*$_{out}$ is the UIW_PULL_DOWN_ITEM reference. Returning the reference to the UIW_PULL_DOWN_ITEM object allows chaining of the **UIW_PULL_DOWN_ITEM::operator+** overload operator.

- *item*$_{in}$ is a pointer to the UIW_POP_UP_ITEM object or the object derived from the UIW_POP_UP_ITEM base class that is to be added to the pull-down item's list of pop-up items.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a pull-down menu with menu items.
    UIW_PULL_DOWN_MENU *menu = new UIW_PULL_DOWN_MENU(0,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    menu
        + &(*new UIW_PULL_DOWN_ITEM(" Item~1 ", MNF_NO_FLAGS,0)
            + new UIW_POP_UP_ITEM("Option 1.1", MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS)
```

```
            + new UIW_POP_UP_ITEM("Option 1.2", MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS))
          + &(*new UIW_PULL_DOWN_ITEM(" Item 2 ", MNF_NO_FLAGS,0)
            + new UIW_POP_UP_ITEM("Option 2.1",
                MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS))
          + new UIW_PULL_DOWN_ITEM(" Item 3 ", MNF_NO_FLAGS,0)

      // Attach the menu to a window.
      UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
          WOAF_NO_FLAGS);
      *window
          + new UIW_TITLE(" Sample menus ", WOF_JUSTIFY_CENTER)
          + menu;
          .
          .
          .
}
```

UIW_PULL_DOWN_ITEM::operator –

Syntax #include <ui_win.hpp>

UIW_PULL_DOWN_ITEM &UIW_PULL_DOWN_ITEM::
operator – (UIW_POP_UP_ITEM *item);

Remarks This overload operator removes a pop-up menu item from the UIW_-PULL_DOWN_ITEM class object. This operator overload is equivalent to calling the **UIW_PULL_DOWN_ITEM::Subtract** routine, except that it allows the chaining of pop-up menu item removal from the UIW_-PULL_DOWN_ITEM object.

- *returnValue*$_{out}$ is the UIW_PULL_DOWN_ITEM reference. Returning the reference to the UIW_PULL_DOWN_ITEM object allows chaining of the **UIW_PULL_DOWN_ITEM::operator–** overload operator.

- *item*$_{in}$ is a pointer to the UIW_POP_UP_ITEM object or the object derived from the UIW_POP_UP_ITEM base class that is to be removed from the pull-down item's list of pop-up items.

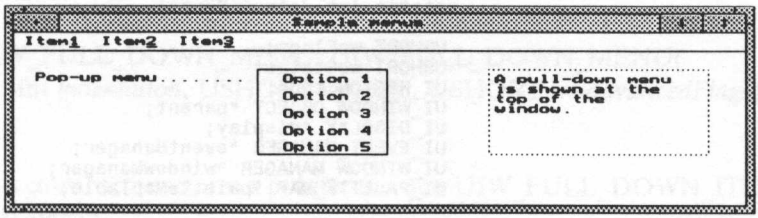
Example

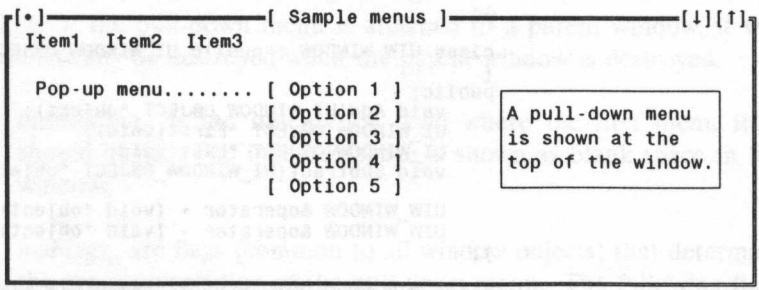
```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a pull-down menu with menu items.
    UIW_PULL_DOWN_MENU *menu = new UIW_PULL_DOWN_MENU(0,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    UIW_PULL_DOWN_ITEM *item1 = new UIW_PULL_DOWN_ITEM(" Item~1 ",
        MNF_NO_FLAGS, 0);
    *item1
        + new UIW_POP_UP_ITEM("Option 1.1", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM("Option 1.2", MNIF_NO_FLAGS,
            BTF_NO_TOGGLE, WOF_NO_FLAGS);

    UIW_PULL_DOWN_ITEM *item2 = new UIW_PULL_DOWN_ITEM(" Item~2 ",
        MNF_NO_FLAGS, 0);
    UIW_POP_UP_ITEM *item21 = new UIW_POP_UP_ITEM("Option 2.1",
        MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS);
    item2 + item21;

    // Attach the menu to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_TITLE(" Sample menus ", WOF_JUSTIFY_CENTER)
        + &(*menu + item1 + item2);
    .
    .
    .
    // Remove the "Option 2.1" item.
    *item2 - item21;
    .
    .
    .
}
```

CHAPTER 40 – UIW_PULL_DOWN_MENU

Overview

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. The figures below show graphic and textual implementations of a UIW_PULL_DOWN_MENU class object with three pull-down items (shown as "Item1," "Item2" and "Item3"):





The public members of the UIW_PULL_DOWN_MENU class (declared in UI_WIN.HPP) are:

```
class UIW_PULL_DOWN_MENU : public UIW_WINDOW
{
public:
    UIW_PULL_DOWN_MENU(int indentation, USHORT woFlags,
      USHORT woAdvancedFlags);
    virtual ~UIW_PULL_DOWN_MENU(void);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_WINDOW : public UI_WINDOW_OBJECT
{
public:
    void Add(UI_WINDOW_OBJECT *object);
    UI_WINDOW_OBJECT *First(void);
    UI_WINDOW_OBJECT *Last(void)
    void Subtract(UI_WINDOW_OBJECT *object);

    UIW_WINDOW &operator + (void *object);
    UIW_WINDOW &operator - (void *object);
};

class UIW_PULL_DOWN_MENU : public UIW_WINDOW;
```

See also The example file **XWMENU.CPP**, which gives a complete example of the UIW_PULL_DOWN_MENU class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 27—UIW_WINDOW" of this manual, which describes the base class from which the UIW_PULL_DOWN_ITEM class is derived.

"Chapter 37—UIW_POP_UP_MENU" of this manual, which describes a similar menu class that displays menu items vertically on the screen.

"Chapter 35—UIW_PULL_DOWN_ITEM" of this manual, which describes the menu item class used by the pull-down menu.

UIW_PULL_DOWN_MENU::UIW_PULL_DOWN_MENU

Syntax #include <ui_win.hpp>

UIW_PULL_DOWN_MENU::UIW_PULL_DOWN_MENU(
int *indentation*, USHORT *woFlags*, USHORT *woAdvancedFlags*);

Remarks This constructor returns a pointer to a new UIW_PULL_DOWN_ITEM class object.

NOTE: If the pull-down menu is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- *indentation*$_{in}$ is the indentation level where the first menu item should begin. The indented space is shown as blank space in the window.

- *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.

  WOF_NO_FLAGS—Does not associate any special flags with the menu. This flag should not be used in conjunction with any other WOF flag.

  WOF_NON_SELECTABLE—Prevents the menu from being selected. If this flag is set, the user will not be able to edit or move within the menu.

- *woAdvancedFlags*$_{in}$ are flags that determine the advanced operation of the menu.

  WOAF_NO_FLAGS—Does not associated any special advanced flags with the menu. Setting this flag allows the user to move, size and interact with the menu in a normal fashion. This flag should not be used in conjunction with any other WOAF flag.

  WOAF_TEMPORARY—The pull-down menu only occupies the screen temporarily. Once another window is selected from the screen, the temporary menu is destroyed.

  WOAF_NO_DESTROY—Prevents the window manager from calling the pull-down menu's destructor. If this flag is set, the menu can be removed from the screen display, but the programmer must call the associated destructor.

  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 menu from the screen display.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a pull-down menu with menu items.
    UIW_PULL_DOWN_MENU *menu = new UIW_PULL_DOWN_MENU(0,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    menu
        + &(*new UIW_PULL_DOWN_ITEM(" Item~1 ", MNF_NO_FLAGS,0)
            + new UIW_POP_UP_ITEM("Option 1.1", MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS)
            + new UIW_POP_UP_ITEM("Option 1.2", MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS))
        + &(*new UIW_PULL_DOWN_ITEM(" Item~2 ", MNF_NO_FLAGS,0)
            + new UIW_POP_UP_ITEM("Option 2.1",
                MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS))
        + new UIW_PULL_DOWN_ITEM(" Item~3 ", MNF_NO_FLAGS,0)
```

```
    // Attach the menu to a window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample menus ", WOF_JUSTIFY_CENTER)
        + menu;
    .
    .
    .
}
```

UIW_PULL_DOWN_MENU::~UIW_PULL_DOWN_MENU

Syntax

```
#include <ui_win.hpp>

virtual UIW_PULL_DOWN_MENU::
    ~UIW_PULL_DOWN_MENU(void);
```

Remarks

This virtual destructor destroys the class information associated with the UIW_PULL_DOWN_MENU class object. Care should be taken to only destroy those pull-down menus that are not automatically destroyed by the parent window or the window manager.

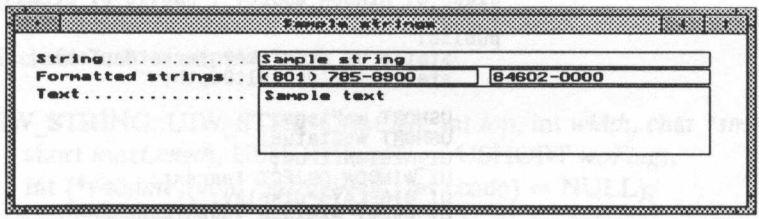
Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a pull-down menu with menu items.
    UIW_PULL_DOWN_MENU *menu = new UIW_PULL_DOWN_MENU(0,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    menu
        + &(*new UIW_PULL_DOWN_ITEM(" Item~1 ", MNF_NO_FLAGS,0)
            + new UIW_POP_UP_ITEM("Option 1.1", MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS)
            + new UIW_POP_UP_ITEM("Option 1.2", MNIF_NO_FLAGS,
                BTF_NO_TOGGLE, WOF_NO_FLAGS))
        + &(*new UIW_PULL_DOWN_ITEM(" Item~2 ", MNF_NO_FLAGS,0)
            + new UIW_POP_UP_ITEM("Option 2.1",
                MNIF_NO_FLAGS, BTF_NO_TOGGLE, WOF_NO_FLAGS))
        + new UIW_PULL_DOWN_ITEM(" Item~3 ", MNF_NO_FLAGS,0)

    // Attach the menu to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new BORDER
        + new UIW_TITLE(" Sample menus ", WOF_JUSTIFY_CENTER)
        + menu;
    .
    .
    .
```

```
    // Manually destroy the menu and its parent window.
    *window - menu;
    delete menu;
    delete window;
    // We could have just called "delete  window."  Its destructor
    // would have automatically called the menu object destructor.
}
```

CHAPTER 41 – UIW_STRING

Overview 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 figures below show graphic and textual implementations of a UIW_STRING object:




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

 String..... [Sample string                    ]
 Formatted strings.. [(801) 785-8900   ] [84602-0000   ]
 Text.....
                     Sample text
```

The public members of the UIW_STRING class (declared in **UI_WIN.HPP**) are:

```
class UIW_STRING : public UI_WINDOW_OBJECT, public UI_EDIT_INFO
{
public:
    USHORT strFlags;

    UIW_STRING(int left, int top, int width, char *string,
        short maxLength, USHORT strFlags, USHORT woFlags,
        int (*validate)(void *stringField, int ccode) = NULL);
    virtual ~UIW_STRING(void);

    const char *DataGet(void);
    void DataSet(char *string, int maxLength = -1);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UI_EDIT_INFO
{
public:
    static void UndoStrategy(short maxObjects, long maxBytes,
        short maxUndos, long maxBytesPerObject,
        short maxUndosPerObject);
};

class UIW_STRING :
    public UI_WINDOW_OBJECT, public UI_EDIT_INFO;
```

See also The example file **XWSTRING.CPP**, which gives a complete example of the UIW_STRING class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes the base class from which the UIW_STRING class is derived.

"Chapter 28—UIW_DATE" of this manual, which describes a class derived from the UIW_STRING class.

"Chapter 43—UIW_TEXT" of this manual, which describes a class derived from the UIW_STRING class.

"Chapter 44—UIW_TIME" of this manual, which describes a class derived from the UIW_STRING class.

UIW_STRING::UIW_STRING

Syntax #include <ui_win.hpp>

UIW_STRING::UIW_STRING(int *left*, int *top*, int *width*, char **string*, short *maxLength*, USHORT *strFlags*, USHORT *woFlags*, int (**validate*)(void *stringField, int ccode) = NULL);

Remarks This constructor returns a pointer to a new UIW_STRING class object.

NOTE: If the string window object is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- $left_{in}$ and top_{in} is 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.)
- $string_{in/out}$ is a pointer to the initial string to be displayed to the screen. This pointer is used by the string object if the WOF_NO_-ALLOCATE_DATA flag is set for the field. Otherwise, the string is copied into a buffer, allocated by the UIW_STRING object, which is *maxLength* in length.
- $maxLength_{in}$ is the maximum length of the string buffer.
- $strFlags_{in}$ gives information on how to display the string information. At present, only **STF_NO_FLAGS** is supported. This flag does not associate any special flags with the UIW_STRING class object.
- $woFlags_{in}$ 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 positions on the first character of the string field (from another window field) then presses a key (without having previously pressed any movement or editing keys).

WOF_BORDER—Draws a border around the string object. In graphics mode, setting this option draws a single line border around the object. In text mode, setting this option draws display brackets (i.e., '[' ']') around the object.

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 *validate* function (discussed below). For example, a string field may initially be set to 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_NO_ALLOCATE_DATA—Prevents the string object from allocating a string buffer that stores the string information. If this flag is set, the programmer must allocate the string buffer (passed as the *string* parameter) that is used by the string object.

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_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 flag.

WOF_NO_INVALID—Prevents the "Leave invalid" option from being selectable by the end-user when an incomplete or invalid string is entered.

WOF_NO_UNANSWERED—Prevents the "Leave unanswered" option from being selectable by the end-user when an incomplete or invalid string is entered.

WOF_NON_SELECTABLE—Prevents the string object from being selected. If this flag is set, the end-user will not be able to edit 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.

- *validate*$_{in}$ is a programmer defined function that is called whenever:

  1—a string is entered and the user moves to a different field in the form, or

  2—the user moves to a different window on the screen.

  The following arguments are passed to *validate* when string information is entered:

  stringField$_{in}$—A pointer to the UIW_STRING class object or the class object derived from the UIW_STRING object base class. This argument must be typecast by the programmer.

  ccode$_{in}$—The logical or system code that caused the validate function to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

  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 validate 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>

ExampleFunction1()
{
    // Add a string field to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "String.............", WOF_NO_FLAGS)
        + new UIW_STRING(22, 1, 41, "Sample string", 256,
            STF_NO_FLAGS, WOF_BORDER);
    .
    .
    .
}
```

UIW_STRING::~UIW_STRING

Syntax

#include <ui_win.hpp>

virtual UIW_STRING::~UIW_STRING(void);

Remarks

This virtual destructor destroys the class information associated with the UIW_STRING object. Care should be taken to only destroy those string objects that are not attached to a parent window.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a string field to the window.
    UIW_STRING *stringField = new UIW_STRING(22, 1, 41,
        "Sample string", 256, STF_NO_FLAGS, WOF_BORDER);

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "String.............", WOF_NO_FLAGS)
        + stringField;
    .
    .
    .
```

```
    // Manually destroy the string field and its parent window.
    *window - stringField;
    delete stringField;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the string object
    // destructor.
}
```

UIW_STRING::DataGet

Syntax #include <ui_win.hpp>

const char *UIW_STRING::DataGet(void);

Remarks This function gets the current string information associated with the UIW_STRING class object. This function returns a pointer to a constant character array. Thus, the contents of the array cannot be directly modified by the programmer.

- *returnValue*$_{out}$ is a constant pointer to the string buffer.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a string field to the window.
    UIW_STRING *stringField = new UIW_STRING(22, 1, 41,
        "Sample string", 256, STF_NO_FLAGS, WOF_BORDER);

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "String.............", WOF_NO_FLAGS)
        + stringField;
    .
    .
    .
    // Get the contents of the string buffer.
    const char *buffer = stringField->DataGet();
    .
    .
    .
}
```

UIW_STRING::DataSet

Syntax #include <ui_win.hpp>

void UIW_STRING::DataSet(char **string*, int *maxLength* = -1);

Remarks This function resets the current string information associated with the UIW_STRING class object or tells the class object that key flags, associated with the string object, have been changed.

- $string_{in/out}$ is a pointer to the new string buffer. 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_STRING class object is destroyed. Otherwise, the information associated with this argument is copied by the UIW_STRING class object. If this argument is NULL, no string information is changed, but the string field is re-displayed.

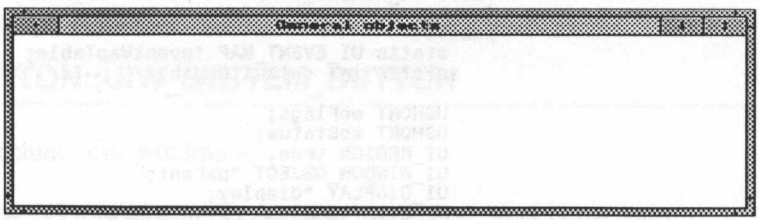
- $maxLength_{in}$ is the new maximum length of the string buffer. If this value is -1, the new value is ignored. If the new length is greater than the old length and the WOF_NO_ALLOCATE_DATA flag is not set, a new string buffer is allocated by the UIW_STRING class object.

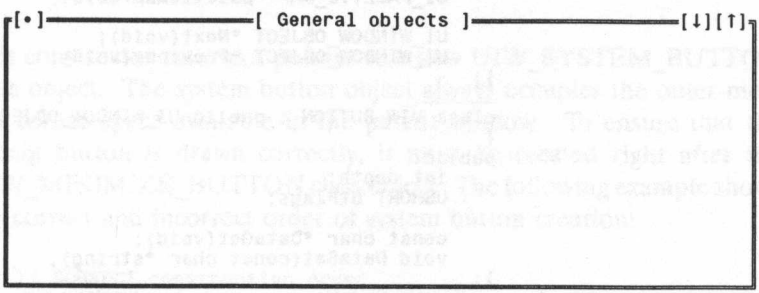
Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a string field to the window.
    UIW_STRING *stringField = new UIW_STRING(22, 1, 41,
        "Sample string", 256, STF_NO_FLAGS, WOF_BORDER);

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE(" Sample strings ", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "String.............", WOF_NO_FLAGS)
        + stringField;
    .
    .
    .
    // Reset the string field buffer but leave the old length.
    stringField->DataSet("This is a new sample string.", -1);
}
```

CHAPTER 42 – UIW_SYSTEM_BUTTON

Overview The UIW_SYSTEM_BUTTON class is used to select general operations on a window (e.g., size, move, maximize, minimize). The figures below show graphic and textual implementations of a UIW_SYSTEM_-BUTTON class object (the button with the '•' character):





The public members of the UIW_SYSTEM_BUTTON class (declared in **UI_WIN.HPP**) are:

```
class UIW_SYSTEM_BUTTON : public UIW_BUTTON
{
public:
    UIW_SYSTEM_BUTTON(void);
    virtual ~UIW_SYSTEM_BUTTON(void);

    void Add(UIW_POP_UP_ITEM *item);
    void Subtract(UIW_POP_UP_ITEM *item);

    UIW_SYSTEM_BUTTON &operator + (UIW_POP_UP_ITEM *object);
    UIW_SYSTEM_BUTTON &operator - (UIW_POP_UP_ITEM *object);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_BUTTON : public UI_WINDOW_OBJECT
{
public:
    int depth;
    USHORT btFlags;

    const char *DataGet(void);
    void DataSet(const char *string);
};

class UIW_SYSTEM_BUTTON : public UIW_BUTTON;
```

See also The example file **XWGEN.CPP**, which gives a complete example of the UIW_SYSTEM_BUTTON class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 27—UIW_BUTTON" of this manual, which describes the base class from which the UIW_SYSTEM_BUTTON class is derived.

"Chapter 32—UIW_MAXIMIZE_BUTTON" of this manual, which describes an additional class derived from the UIW_BUTTON class.

"Chapter 33—UIW_MINIMIZE_BUTTON" of this manual, which describes an additional class derived from the UIW_BUTTON class.

"Chapter 35—UIW_POP_UP_ITEM" of this manual, which describes the class used in the system button's pop-up menu.

UIW_SYSTEM_BUTTON::UIW_SYSTEM_BUTTON

Syntax #include <ui_win.hpp>

UIW_SYSTEM_BUTTON::UIW_SYSTEM_BUTTON(void);

Remarks This constructor returns a pointer to a new UIW_SYSTEM_BUTTON class object. 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 created 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,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        .
        .
        .

2)  // INCORRECT construction order.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_SYSTEM_BUTTON
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_BORDER
        .
        .
        .
```

NOTE: If the system button object is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
}
```

UIW_SYSTEM_BUTTON::~UIW_SYSTEM_BUTTON

Syntax

```
#include <ui_win.hpp>

virtual UIW_SYSTEM_BUTTON::
    ~UIW_SYSTEM_BUTTON(void);
```

Remarks

This virtual destructor destroys the class information associated with the UIW_SYSTEM_BUTTON object. Care should be taken to only destroy system button objects that are not attached to a parent window.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    UIW_MAXIMIZE_BUTTON *sysButton = new UIW_SYSTEM_BUTTON;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + sysButton
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
```

```
    // Manually destroy the system button and its parent window.
    *window - sysButton;
    delete sysButton;
    delete window;
    // We could have just called "delete window." Its destructor
    // would have automatically called the system button
    // destructor.
}
```

UIW_SYSTEM_BUTTON::Add

Syntax #include <ui_win.hpp>

void UIW_SYSTEM_BUTTON::Add(UIW_POP_UP_ITEM *item*);

Remarks This function adds a new pop-up menu item to the UIW_SYSTEM_-BUTTON class object.

- $item_{in}$ is a pointer to the item to be added to the system button's list of menu items. The new item must be a UIW_POP_UP_ITEM class object or a class object derived from the UIW_POP_UP_-ITEM base class.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with the basic window objects.
    UIW_SYSTEM_BUTTON *sysButton = new UIW_SYSTEM_BUTTON;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + sysButton
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
```

```
    // Add system button menu options.
    UIW_POP_UP_ITEM *dualMonitor = new UIW_POP_UP_ITEM(
        "~Switch window", MNIF_NO_FLAGS, BTF_NO_TOGGLE,
        WOF_NO_FLAGS);
    sysButton->Add(dualMonitor);
    sysButton->Add(new UIW_POP_UP_ITEM("~Restore", MNIF_RESTORE,
        BTF_NO_TOGGLE, WOF_NO_FLAGS);
    sysButton->Add(new UIW_POP_UP_ITEM("~Move", MNIF_MOVE,
        BTF_NO_TOGGLE, WOF_NO_FLAGS);
    sysButton->Add(new UIW_POP_UP_ITEM("~Size", MNIF_SIZE,
        BTF_NO_TOGGLE, WOF_NO_FLAGS);
    .
    .
    .
}
```

UIW_SYSTEM_BUTTON::Subtract

Syntax

```
#include <ui_win.hpp>

void UIW_SYSTEM_BUTTON::Subtract(
    UIW_POP_UP_ITEM *item);
```

Remarks

This function removes a pop-up menu item from the UIW_SYSTEM_-BUTTON class object. This function does not call the destructor associated with the menu item.

- $item_{in}$ is a pointer to the item to be removed from the current system button's list of pop-up menu items. This argument must be a UIW_POP_UP_ITEM class object or a class object derived from the UIW_POP_UP_ITEM base class.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with the basic window objects.
    UIW_SYSTEM_BUTTON *sysButton = new UIW_SYSTEM_BUTTON;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + sysButton
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);

    // Add system button menu options.
    UIW_POP_UP_ITEM *dualMonitor = new UIW_POP_UP_ITEM(
        "~Switch window", MNIF_NO_FLAGS, BTF_NO_TOGGLE,
        WOF_NO_FLAGS);
    sysButton->Add(dualMonitor);
```

```
    sysButton->Add(new UIW_POP_UP_ITEM("~Restore", MNIF_RESTORE,
        BTF_NO_TOGGLE, WOF_NO_FLAGS);
    sysButton->Add(new UIW_POP_UP_ITEM("~Move", MNIF_MOVE,
        BTF_NO_TOGGLE, WOF_NO_FLAGS);
    sysButton->Add(new UIW_POP_UP_ITEM("~Size", MNIF_SIZE,
        BTF_NO_TOGGLE, WOF_NO_FLAGS);
    .
    .
    .
    // Remove the dual Monitor option if no longer supported.
    extern int DualMonitorSupport(void);
    if (!DualMonitorSupport())
        sysButton->Subtract(dualMonitor);
    .
    .
    .
}
```

UIW_SYSTEM_BUTTON::operator +

Syntax

```
#include <ui_win.hpp>

UIW_SYSTEM_BUTTON &UIW_SYSTEM_BUTTON::
    operator + (UIW_POP_UP_ITEM *item);
```

Remarks This overload operator is used to add menu items to the associated system button menu. This operator overload is equivalent to calling the **UIW_SYSTEM_BUTTON::Add** function, except that it allows the chaining of menu item additions to the UIW_SYSTEM_BUTTON object.

- *returnValue*$_{out}$ is the UIW_SYSTEM_BUTTON reference. Returning the reference to the list allows chaining of the **UIW_SYSTEM_BUTTON::operator+** overload operator.
- *item*$_{in}$ is a pointer to the UI_WINDOW_OBJECT class or the object derived from the UI_WINDOW_OBJECT class that is to be added to the system menu.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
```

```
                + &(*new UIW_SYSTEM_BUTTON
                    + new UIW_POP_UP_ITEM("~Restore", MNIF_RESTORE,
                        BTF_NO_TOGGLE, WOF_NO_FLAGS)
                    + new UIW_POP_UP_ITEM("~Move", MNIF_MOVE,
                        BTF_NO_TOGGLE, WOF_NO_FLAGS)
                    + new UIW_POP_UP_ITEM("~Size", MNIF_SIZE,
                        BTF_NO_TOGGLE, WOF_NO_FLAGS)
                    + new UIW_POP_UP_ITEM("Mi~nimize", MNIF_MINIMIZE,
                        BTF_NO_TOGGLE, WOF_NO_FLAGS)
                    + new UIW_POP_UP_ITEM("Ma~ximize", MNIF_MAXIMIZE,
                        BTF_NO_TOGGLE, WOF_NO_FLAGS)
                    + new UIW_POP_UP_ITEM
                    + new UIW_POP_UP_ITEM("~Close", MNIF_CLOSE,
                        BTF_NO_TOGGLE, WOF_NO_FLAGS))
                + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
            .
            .
            .
}
```

UIW_SYSTEM_BUTTON::operator -

Syntax

```
#include <ui_win.hpp>

UIW_SYSTEM_BUTTON &UIW_SYSTEM_BUTTON::
    operator - (UIW_POP_UP_ITEM *item);
```

Remarks

This overload operator removes a menu item object from the system button's list of menu items. This operator overload is equivalent to calling the **UIW_SYSTEM_BUTTON::Subtract** function, except that it allows the chaining of menu item subtractions from the UIW_SYSTEM_BUTTON class object.

- *returnValue*$_{out}$ is the UIW_SYSTEM_BUTTON reference. Returning the reference to the list allows chaining of the **UIW_SYSTEM_BUTTON::operator-** overload operator.
- *object*$_{in}$ is a pointer to the UI_WINDOW_OBJECT class or the object derived from the UI_WINDOW_OBJECT class that is to be removed from the system menu.

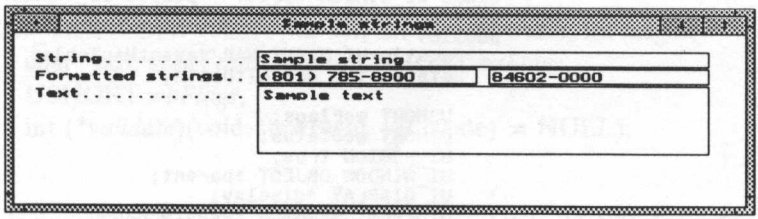
Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_MINIMIZE_BUTTON *sysButton = new UIW_SYSTEM_BUTTON;
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
```

```
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + sysButton
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);

    // Create the system button menu options.
    UIW_POP_UP_ITEM *dualMonitor = new UIW_POP_UP_ITEM(
        "~Switch window", MNIF_NO_FLAGS, BTF_NO_TOGGLE,
        WOF_NO_FLAGS);
    *sysButton
        + dualMonitor
        + new UIW_POP_UP_ITEM("~Restore", MNIF_RESTORE,
            BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM("~Move", MNIF_MOVE,
            BTF_NO_TOGGLE, WOF_NO_FLAGS)
        + new UIW_POP_UP_ITEM("~Size", MNIF_SIZE,
            BTF_NO_TOGGLE, WOF_NO_FLAGS);
    .
    .
    .
    // Remove the dual Monitor option if no longer supported.
    extern int DualMonitorSupport(void);
    if (!DualMonitorSupport())
        *sysButton - dualMonitor;
    .
    .
    .
}
```

CHAPTER 43 – UIW_TEXT

Overview The UIW_TEXT class is used to display text information to the screen and to collect information, in alphanumeric form, from an end user. The figures below show graphic and textual implementations of a UIW_TEXT class object:




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

String..... [Sample string                    ]
Formatted strings.. [(801) 785-8900    ] [84602-0000      ]
Text.....
                    Sample text
```

The public members of the UIW_TEXT class (declared in **UI_WIN.HPP**) are:

```
class UIW_TEXT : public UIW_STRING
{
public:
    USHORT txtFlags;

    UIW_TEXT(int left, int top, int width, int height,
        char *text, short maxLength, USHORT txtFlags,
        USHORT woFlags,
        int (*validate)(void *textField, int ccode) = NULL);
    virtual ~UIW_TEXT(void);

    const char *DataGet(void);
    void DataSet(char *text, short maxLength = -1);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UI_EDIT_INFO
{
public:
    static void UndoStrategy(short maxObjects, long maxBytes,
        short maxUndos, long maxBytesPerObject,
        short maxUndosPerObject);
};

class UIW_STRING : public UI_WINDOW_OBJECT, public UI_EDIT_INFO
{
public:
    USHORT strFlags;

    const char *DataGet(void);
    void DataSet(char *string, int maxLength = -1);
};

class UIW_TEXT : public UIW_STRING;
```

See also The example file **XWSTRING.CPP**, which gives a complete example of the UIW_TEXT class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 41—UIW_STRING" of this manual, which describes the base class from which the UIW_TEXT class is derived.

UIW_TEXT::UIW_TEXT

Syntax #include <ui_win.hpp>

UIW_TEXT::UIW_TEXT(int *left*, int *top*, int *width*, int *height*,
char **text*, short *maxLength*, USHORT *txtFlags*,
USHORT *woFlags*,
int (**validate*)(void *textField, int ccode) = NULL);

Remarks This constructor returns a pointer to a new UIW_TEXT class object.

NOTE: If the text object is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- $left_{in}$ and top_{in} is the starting position of the text field within its parent window.
- $width_{in}$ is the width of the text field.
- $height_{in}$ is the height of the text field.
- $text_{in/out}$ is a pointer to the initial text to be displayed to the screen. This pointer is used by the text object if the WOF_NO_ALLOCATE_DATA flag is set for the field. Otherwise, the text is copied into a buffer, allocated by the UIW_TEXT object, which is *maxLength* in length.
- $maxLength_{in}$ is the maximum length of the text buffer.
- $txtFlags_{in}$ gives information on how to display the text information. Currently, only **TXF_NO_FLAGS** is supported. This flag does not associate any special flags with the UIW_TEXT class object.

- *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 positions on the first character of 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.

  WOF_INVALID—Sets the initial status of the text field to be "invalid." By default, all text information is valid. A programmer may specify a text field as invalid by setting this flag upon creation of the text object or by re-setting the flag through the *validate* function (discussed below). For example, a text field may initially be set to 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 that stores 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 flag.

  WOF_NON_FIELD_REGION—The text object is not a form field. If this flag is set and the text is attached to a higher-level window, then the *left*, *top*, *height* and *width* arguments are ignored and the text occupies any remaining space within the parent window.

  WOF_NO_INVALID—Prevents the "Leave invalid" option from being selectable by the end-user when incomplete or invalid text is entered.

WOF_NO_UNANSWERED—Prevents the "Leave unanswered" option from being selectable by the end-user when incomplete or invalid text is entered.

WOF_NON_SELECTABLE—Prevents the text object from being selected. If this flag is set, the user will not be able to edit or 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.

- *validate*$_{in}$ is a programmer defined function that is called whenever:

  1—a text string is entered and the user moves to a different field in the form, or

  2—the user moves to a different window on the screen.

  The following arguments are passed to *validate* when text information is entered:

  textField$_{in}$—A pointer to the UIW_TEXT class object or the class object derived from the UIW_TEXT object base class. This argument must be typecast by the programmer.

  ccode$_{in}$—The logical or system code that caused the validate 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 validate function'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>

ExampleFunction1()
{
    // Add a text field to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 3, "Text.............", WOF_NO_FLAGS)
        + new UIW_TEXT(22, 3, 41, 4, "Sample text", 1028,
            TXF_NO_FLAGS, WOF_BORDER);
        .
        .
        .
}

ExampleFunction2()
{
    // Add a text object to the window.  Unlike ExampleFunction1(),
    // this text field occupies the remaining part of the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 3, "Text.............", WOF_NO_FLAGS)
        + new UIW_TEXT(0, 0, 0, 0, "Sample text", 1028,
            TXF_NO_FLAGS, WOF_NON_FIELD_REGION);
        .
        .
        .
}
```

UIW_TEXT::~UIW_TEXT

Syntax #include <ui_win.hpp>

virtual UIW_TEXT::~UIW_TEXT(void);

Remarks This virtual destructor destroys the class information associated with the UIW_TEXT object. Care should be taken to only destroy those text objects that are not attached to a parent window.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a text field to the window.
    UIW_TEXT *text = new UIW_TEXT(22, 3, 41, 4, "Sample text",
        1028, TXF_NO_FLAGS, WOF_BORDER);

    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 3, "Text.............", WOF_NO_FLAGS)
        + text;
    .
    .
    .
    // Manually destroy the text object and its parent window.
    *window - text;
    delete text;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the text object destructor.
}
```

UIW_TEXT::DataGet

Syntax

#include <ui_win.hpp>

const char *UIW_TEXT::DataGet(void);

Remarks

This function gets the current text information associated with the UIW_TEXT class object. This function returns a pointer to a constant character array. Thus, the contents of the array cannot be directly modified by the programmer.

- *returnValue*$_{out}$ is a constant pointer to the text buffer.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a text field to the window.
    UIW_TEXT *text = new UIW_TEXT(22, 3, 41, 4, "Sample text",
        1028, TXF_NO_FLAGS, WOF_BORDER);
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 3, "Text.............", WOF_NO_FLAGS)
        + text;
    .
    .
    .
    // Get the contents of the buffer.
    const char *buffer = text->DataGet();
    .
    .
    .
}
```

UIW_TEXT::DataSet

Syntax

#include <ui_win.hpp>

void UIW_TEXT::DataSet(char **text*, short *maxLength* = -1);

Remarks

This function resets the current text information associated with the UIW_TEXT class object or tells the class object that key flags, associated with the text object, have been changed.

- $text_{in/out}$ is a pointer to the new text buffer. 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_TEXT class object is destroyed. Otherwise, the information associated with this argument is copied by the UIW_TEXT class object. If this argument is NULL, no text information is changed, but the text field is re-displayed.

- $maxLength_{in}$ is the new maximum length of the text buffer. If this value is -1, the new value is ignored. If the new buffer length is greater than the old length and the WOF_NO_ALLOCATE_DATA flag is not set, a new text buffer is allocated by the UIW_TEXT class object.

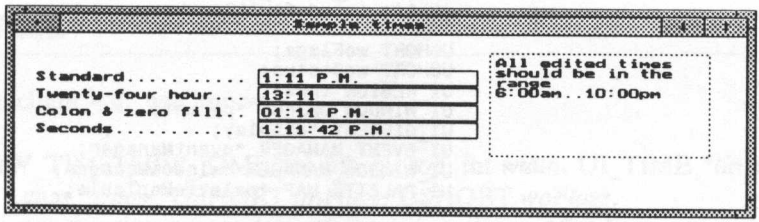
Example

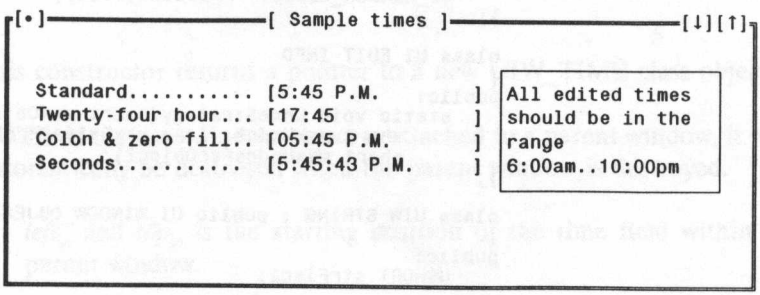
```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a text field to the window.
    UIW_TEXT *text = new UIW_TEXT(22, 3, 41, 4, "Sample text",
        1028, TXF_NO_FLAGS, WOF_BORDER);
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 3, "Text.............", WOF_NO_FLAGS)
        + text;
    .
    .
    .
    // Reset the text field buffer but leave the old length.
    text->DataSet("This is new sample text.", -1);
    .
    .
    .
}
```

CHAPTER 44 – UIW_TIME

Overview

The UIW_TIME class is used to display time information to the screen and to collect information, in time form, from an end user. It is not the low-level time storage object. (See "Chapter 23—UI_TIME" of this manual for information about the low-level time storage object.) The pictures below show graphic and textual implementations of UIW_TIME objects:





The public members of the UIW_TIME class (declared in **UI_WIN.HPP**) are:

```
class UIW_TIME : public UIW_STRING
{
public:
    UIW_TIME(int left, int top, int width, UI_TIME *time,
        char *range, USHORT tmFlags, USHORT woFlags,
        int (*validate)(void *timeField, int ccode) = NULL);
    virtual ~UIW_TIME(void);

    const UIW_TIME *DataGet(void);
    void DataSet(UIW_TIME *time);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UI_EDIT_INFO
{
public:
    static void UndoStrategy(short maxObjects, long maxBytes,
        short maxUndos, long maxBytesPerObject,
        short maxUndosPerObject);
};

class UIW_STRING : public UI_WINDOW_OBJECT, public UI_EDIT_INFO
{
public:
    USHORT strFlags;

    const char *DataGet(void);
    void DataSet(char *string, int maxLength = -1);
};

class UIW_TIME : public UIW_STRING;
```

See also The example file **XWTIME.CPP**, which gives a complete example of the UIW_TIME class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 23—UI_TIME" of this manual, which describes the low-level time storage class object.

"Chapter 28—UIW_DATE" of this manual, which describes a similar high-level class object that stores date information.

"Chapter 41—UIW_STRING" of this manual, which describes the base class from which the UIW_TIME class is derived.

UIW_TIME::UIW_TIME

Syntax #include <ui_win.hpp>

UIW_TIME::UIW_TIME(int *left*, int *top*, int *width*, UI_TIME **time*, char **range*, USHORT *tmFlags*, USHORT *woFlags*, int (**validate*)(void *timeField, int ccode) = NULL);

Remarks This constructor returns a pointer to a new UIW_TIME class object.

NOTE: If the time window object is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- $left_{in}$ and top_{in} is 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.)
- $time_{in/out}$ is a pointer to the initial time 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_-TIME class object is destroyed.
- $range_{in}$ is a string that gives the valid time ranges. For example, if a range of "12:01pm..11:59:59pm" were specified, the UIW_TIME class object would only accept those times whose values fell in the post-meridian time. If *range* is NULL, any time value 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 country dependant information (supplied by all DOS based systems):

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

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

  TMF_LOWER_CASE—Converts the time to lower-case. Some example times with the TMF_LOWER_CASE flag set are: "12:00 p.m." and "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. This flag should not be used in conjunction with any other TMF flag.

  TMF_NO_HOURS—Does not display or 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 the minutes and "15" would be interpreted as the seconds.

  TMF_NO_MINUTES—Does not display or interpret a minute value for the UIW_TIME class object. For example, if the end-user entered the time "12:15" and the TMF_NO_MINUTES were set, the displayed time would be interpreted as 12 seconds and 15 hundredths of seconds.

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

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

TM_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 (e.g., "1:10pm").

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

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

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

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

- *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 positions on to the first character of the time field (from another window field) and then presses a key (without having previously pressed any movement or editing keys).

  WOF_BORDER—Draws a border around the time object. In graphics mode, setting this option draws a single line border around the object. In text mode, setting this option draws display braces (i.e., '[' ']') around the object.

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., "12:00pm..11:59:59pm") but does not fulfill all the requirements specified by the program. For example, a time field may initially be set to "8:15am," but the final time, edited by the end-user, must be in the range "12:00pm..11:59:59pm." The initial time in this example fits the absolute requirements of a 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 class object to store the time information. If this flag is set, the programmer must allocate the UI_TIME (passed as the *time* parameter) that is used by the time object.

WOF_NO_FLAGS—Does not associate any special window 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_NO_INVALID—Prevents the "Leave invalid" option from being selectable by the end-user when an incomplete or invalid time value is entered.

WOF_NO_UNANSWERED—Prevents the "Leave unanswered" option from being selectable by the end-user when an incomplete or invalid time value is entered.

WOF_NON_SELECTABLE—Prevents the time object from being selected. If this flag is set, the user will not be able to edit 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.

- *validate*$_{in}$ is a programmer defined function that is called whenever:

  1—a time string is entered and the user moves to a different field in the form, or

  2—the user moves to a different window on the screen.

The *validate* function is not called if the time does not fit 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 *validate* when a new time is entered:

timeField$_{in}$—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.

ccode$_{in}$—The logical or system code that caused the validate function to be called. This code (declared in **UI_EVT.HPP**) will be one of the following constant values:

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 validate 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>

ExampleFunction1()
{
    // Add several time fields to the window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 2, "Standard.....", WOF_NO_FLAGS)
        + new UIW_TIME(22, 2, 20, &time, "6:00am..10:00pm",
            TMF_NO_FLAGS, WOF_BORDER)
        + new UIW_PROMPT(2, 3, "Twenty-four hour...", WOF_NO_FLAGS)
```

```
        + new UIW_TIME(22, 3, 20, &time, "6:00am..10:00pm",
            TMF_TWENTY_FOUR_HOUR, WOF_BORDER)
        + new UIW_PROMPT(2, 4, "Colon & zero fill..", WOF_NO_FLAGS)
        + new UIW_TIME(22, 4, 20, &time, "6:00am..10:00pm",
            TMF_COLON_SEPARATOR | TMF_ZERO_FILL, WOF_BORDER);

    .
    .
    .
}
```

UIW_TIME::~UIW_TIME

Syntax

#include <ui_win.hpp>

virtual UIW_TIME::~UIW_TIME(void);

Remarks

This virtual destructor destroys the class information associated with the UIW_TIME object. Care should be taken to only destroy those time objects that are not attached to a parent window.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    UI_DATE date; // system date
    UIW_DATE *dateField = new UIW_DATE(9, 1, 20, &date,
        "", DTF_ALPHA_MONTH | DTF_SYSTEM, WOF_BORDER);
    UI_TIME time; // system time
    UIW_TIME *timeField = new UIW_TIME(9, 1, 20, &time,
        "", TMF_SECONDS, WOF_BORDER);

    // Create a window with system date and time information.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11, WOF_NO_FLAGS,
        WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Date..", WOF_NO_FLAGS)
        + dateField
        + new UIW_PROMPT(2, 1, "Time..", WOF_NO_FLAGS)
        + timeField;

    .
    .
    .
    // Manually destroy the time field and its parent window.
    *window - timeField;
    delete timeField;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the time object destructor.
}
```

UIW_TIME::DataGet

Syntax

```
#include <ui_win.hpp>

const UI_TIME *UIW_TIME::DataGet(void);
```

Remarks

This function gets the current time information associated with the UIW_TIME class object. This function returns a pointer to a constant UI_TIME variable. Thus, the contents of this variable cannot be directly modified by the programmer.

- *returnValue*$_{out}$ is a constant pointer to the UI_TIME variable.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    UI_DATE date; // system date
    UIW_DATE *dateField = new UIW_DATE(9, 1, 20, &date,
        "", DTF_ALPHA_MONTH | DTF_SYSTEM, WOF_BORDER);
    UI_TIME time; // system time
    UIW_TIME *timeField = new UIW_TIME(9, 2, 20, &time,
        "", TMF_SECONDS, WOF_BORDER);

    // Create a window with system date and time information.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Date..", WOF_NO_FLAGS)
        + dateField
        + new UIW_PROMPT(2, 2, "Time..", WOF_NO_FLAGS)
        + timeField;
    .
    .
    .
    // Reset the date and time information.
    extern int ResetDateTime(void);
    if (ResetDateTime())
    {
        date.Import();  // Get the new system date.
        time.Import();  // Get the new system time.
        dateField->DataSet(&date);
        timeField->DataSet(&time);
    }
    else
    {
        date = *dateField->DataGet();
        time = *timeField->DataGet();
    }
    .
    .
    .
}
```

UIW_TIME::DataSet

Syntax

#include <ui_win.hpp>

void UIW_TIME::DataSet(UI_TIME **time*);

Remarks

This function resets the current time information associated with the UIW_TIME class object or is used to tell the class object that key flags, associated with the time object, have been changed.

- $time_{in/out}$ is a pointer to the new time information. 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_TIME class object is destroyed. Otherwise, the information associated with this argument is copied by the UIW_TIME class object. If this argument is NULL, no time information is changed, but the time field is re-displayed.

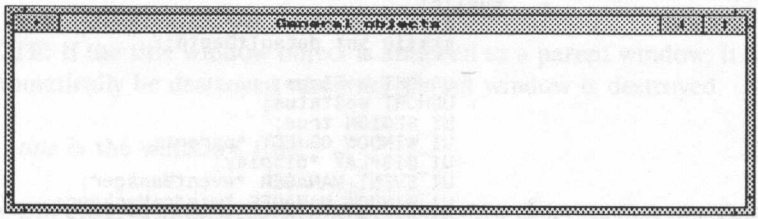
Example

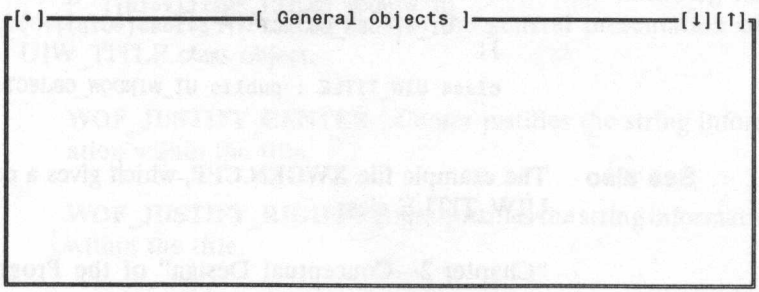
```
#include <ui_win.hpp>

ExampleFunction1()
{
    UI_DATE date; // system date
    UIW_DATE *dateField = new UIW_DATE(9, 1, 20, &date,
        "", DTF_ALPHA_MONTH | DTF_SYSTEM, WOF_BORDER);
    UI_TIME time; // system time
    UIW_TIME *timeField = new UIW_TIME(9, 2, 20, &time,
        "", TMF_SECONDS, WOF_BORDER);

    // Create a window with system date and time information.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER)
        + new UIW_PROMPT(2, 1, "Date..", WOF_NO_FLAGS)
        + dateField
        + new UIW_PROMPT(2, 2, "Time..", WOF_NO_FLAGS)
        + timeField;
    .
    .
    .
```

```
        // Reset the date and time information.
        extern int ResetDateTime(void);
        if (ResetDateTime())
        {
            date.Import();  // Get the new system date.
            time.Import();  // Get the new system time.
            dateField->DataSet(&date);
            timeField->DataSet(&time);
        }
        else
        {
            date = *dateField->DataGet();
            time = *timeField->DataGet();
        }
        .
        .
        .
    }
```

CHAPTER 45 – UIW_TITLE

Overview The UIW_TITLE class is used to display short textual information about the parent window and, when clicked on with a mouse, to move the position of the parent window. The figures below show graphic and textual implementations of a window with a UIW_TITLE class object (shown with the "General objects" string):





The public members of the UIW_TITLE class (declared in **UI_WIN.HPP**) are:

```
class UIW_TITLE : public UI_WINDOW_OBJECT
{
public:
    UIW_TITLE(char *title,
        USHORT woFlags = WOF_JUSTIFY_CENTER);
    virtual ~UIW_TITLE(void);

    const char *DataGet(void);
    void DataSet(char *title);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_TITLE : public UI_WINDOW_OBJECT;
```

See also The example file **XWGEN.CPP**, which gives a complete example of the UIW_TITLE class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes the base class from which the UIW_TITLE class is derived.

UIW_TITLE::UIW_TITLE

Syntax

```
#include <ui_win.hpp>

UIW_TITLE::UIW_TITLE(char *title,
    USHORT woFlags = WOF_JUSTIFY_CENTER);
```

Remarks

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

NOTE: If the title window object is attached to a parent window, it will automatically be destroyed when the parent window is destroyed.

- *title* 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.

  WOF_JUSTIFY_RIGHT—Right-justifies the string information within the title.

  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 *title* 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>

ExampleFunction1()
{
    // Create a basic window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
}
```

UIW_TITLE::~UIW_TITLE

Syntax

#include <ui_win.hpp>

virtual UIW_TITLE::~UIW_TITLE(void);

Remarks

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

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a title to the window.
    UIW_TITLE *title = new UIW_TITLE("Window 1",
        WOF_JUSTIFY_CENTER);
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_DESTROY);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + title;
    .
    .
    .
    // Manually destroy the title and its parent window.
    *window - title;
    delete title;
    delete window;
    // We could have just called "delete window."  Its destructor
    // would have automatically called the title object destructor.
}
```

UIW_TITLE::DataGet

Syntax #include <ui_win.hpp>

const char *UIW_TITLE::DataGet(void);

Remarks This function is used to get the current title information associated with the UIW_TITLE class object. This function returns a pointer to a constant character buffer. Thus, the contents of this buffer cannot be directly modified by the programmer.

- *returnValue*$_{out}$ is a constant pointer to the title's character buffer.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a title to the window.
    UIW_TITLE *titleField = new UIW_TITLE("Window 1",
        WOF_JUSTIFY_CENTER);
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + titleField;
    .
    .
    .
    // Check the window's title.
    char title[128];
    extern int ModifyTitle(char *title);
    if (ModifyTitle(title))
        titleField->dataSet(title);
    else
        strcpy(title, titleField->dataGet());
}
```

UIW_TITLE::DataSet

Syntax #include <ui_win.hpp>

void UIW_TITLE::DataSet(const char *title);

Remarks This function is used to reset the current title information associated

with the UIW_TITLE class object or to tell the class object that key flags, associated with the title object, have been changed.

- *title*$_{in/out}$ is a pointer to the new title. 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_-TITLE class object is destroyed. Otherwise, the information associated with this argument is copied by the UIW_TITLE class object. If this argument is NULL, no title information is changed, but the title is re-displayed.

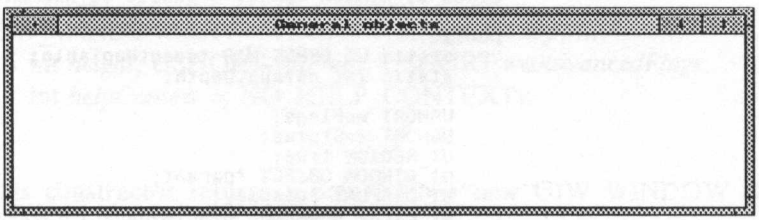
Example

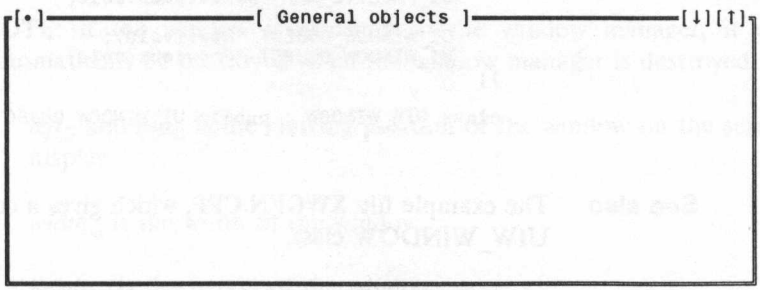
```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Manually add a title to the window.
    UIW_TITLE *titleField = new UIW_TITLE("Window 1",
        WOF_JUSTIFY_CENTER);
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + titleField;
    .
    .
    .
    // Check the window's title.
    char title[128];
    extern int ModifyTitle(char *title);
    if (ModifyTitle(title))
        titleField->dataSet(title);
    else
        strcpy(title, titleField->dataGet());
}
```

CHAPTER 46 – UIW_WINDOW

Overview

The UIW_WINDOW class is used as the controlling object for window fields that are to be displayed on the screen. The figures below show graphic and textual implementations of a UIW_WINDOW class object with several attached window objects (i.e., border, buttons, title):





The public members of the UIW_WINDOW class (declared in **UI_WIN.HPP**) are:

```
class UIW_WINDOW : public UI_WINDOW_OBJECT
{
public:
    UIW_WINDOW(int left, int top, int width, int height,
        USHORT woFlags, USHORT woAdvancedFlags,
        int helpContext = NO_HELP_CONTEXT);
    virtual ~UIW_WINDOW(void);

    void Add(UI_WINDOW_OBJECT *object);
    UI_WINDOW_OBJECT *First(void);
    UI_WINDOW_OBJECT *Last(void)
    void Subtract(UI_WINDOW_OBJECT *object);

    UIW_WINDOW &operator + (void *object);
    UIW_WINDOW &operator - (void *object);
};
```

Inheritance The programmer should be aware of the following inherited member functions and variables:

```
class UI_ELEMENT
{
public:
    UI_ELEMENT *previous;
    UI_ELEMENT *next;
};

class UI_WINDOW_OBJECT : public UI_ELEMENT
{
public:
    static UI_EVENT_MAP *eventMapTable;
    static int defaultDepth;

    USHORT woFlags;
    USHORT woStatus;
    UI_REGION true;
    UI_WINDOW_OBJECT *parent;
    UI_DISPLAY *display;
    UI_EVENT_MANAGER *eventManager;
    UI_WINDOW_MANAGER *windowManager;
    UI_PALETTE_MAP *paletteMapTable;

    UI_WINDOW_OBJECT *Next(void);
    UI_WINDOW_OBJECT *Previous(void);
};

class UIW_WINDOW : public UI_WINDOW_OBJECT;
```

See also The example file **XWGEN.CPP**, which gives a complete example of the UIW_WINDOW class.

"Chapter 2—Conceptual Design" of the Programmer's Guide, which gives an overview to the window manager and its interaction with window objects.

"Chapter 3—Window Objects" of the Programmer's Guide, which gives information about individual window class objects.

"Chapter 25—UI_WINDOW_OBJECT" of this manual, which describes the base class from which the UIW_WINDOW class is derived.

"Chapter 25—UIW_LIST" of this manual, which describes a class object derived from the UIW_WINDOW class.

"Chapter 31—UIW_MATRIX" of this manual, which describes a class object derived from the UIW_WINDOW class.

"Chapter 36—UIW_POP_UP_MENU" of this manual, which describes a class object derived from the UIW_WINDOW class.

"Chapter 40—UIW_PULL_DOWN_MENU" of this manual, which describes a class object derived from the UIW_WINDOW class.

UIW_WINDOW::UIW_WINDOW

Syntax #include <ui_win.hpp>

UIW_WINDOW::UIW_WINDOW(int *left*, int *top*, int *width*, int *height*, USHORT *woFlags*, USHORT *woAdvancedFlags*, int *helpContext* = NO_HELP_CONTEXT);

Remarks This constructor returns a pointer to a new UIW_WINDOW class object.

NOTE: If the window is attached to the window manager, it will automatically be destroyed when the window manager is destroyed.

- $left_{in}$ and top_{in} is the starting position of the window on the screen display.
- $width_{in}$ is the width of the window.
- $height_{in}$ is the height 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—Draws a single line border around the window.

  WOF_NO_FLAGS—Does not associate any special flags with the window. This flag should not be used in conjunction with any other WOF flag.

  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_MATRIX, UIW_PULL_DOWN_MENU):

  WOAF_NO_FLAGS—Does not associated 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 flag.

  WOAF_TEMPORARY—The window only occupies the screen temporarily. Once another window is selected from the screen, the temporary window is destroyed.

  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_SIZE—Prevents the end-user from changing the size of the window 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 window from the screen display.

- *helpContext*$_{in}$ is the help information associated with the window. This specifies the type of help that will be presented by the help system when the help key is pressed and the end-user is viewing the specified window. (For more information about the help system and help context information see"Chapter 15—UI_HELP_-SYSTEM" of this manual.)

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    extern UI_WINDOW_MANAGER *_windowManager;
    *_windowManager + window;
    .
    .
    .
}

ExampleFunction2()
{
    // Create a window with basic window objects.  This is the same
    // code as above, but the window is attached directly to the
    // screen display.
    extern UI_WINDOW_MANAGER *_windowManager;
    *_windowManager
        + &(*new UIW_WINDOW(0, 1, 67, 11, WOF_NO_FLAGS,
              WOAF_NO_FLAGS)
            + new UIW_BORDER
            + new UIW_MAXIMIZE_BUTTON
            + new UIW_MINIMIZE_BUTTON
            + new UIW_SYSTEM_BUTTON
            + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER));
    .
    .
    .
}
```

UIW_WINDOW::~UIW_WINDOW

Syntax #include <ui_win.hpp>

virtual UIW_WINDOW::~UIW_WINDOW(void);

Remarks This virtual destructor destroys the class information associated with the UIW_WINDOW object. Care should be taken to only destroy those windows that are not attached to the window manager.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    extern UI_WINDOW_MANAGER *_windowManager;

    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE(" General objects ", WOF_JUSTIFY_CENTER);
    *windowManager + window;
    .
    .
    .
    // Remove the window from the screen display then destroy it.
    *windowManager - window;
    delete window;
    // NOTE: If the window had been created with WOAF_NO_FLAGS, the
    // window manager would automatically call the
    // window's destructor
}
```

UIW_WINDOW::Add

Syntax

#include <ui_win.hpp>

void UIW_WINDOW::Add(UI_WINDOW_OBJECT **object*);

Remarks

This function adds a new window object to the UIW_WINDOW class object.

- $object_{in}$ is a pointer to the object to be added to the current window's list of window objects. This argument must be a class object derived from the UI_WINDOW_OBJECT base class.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    extern UI_WINDOW_MANAGER *_windowManager;

    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    UIW_BORDER *border = new UIW_BORDER;
    UIW_MAXIMIZE_BUTTON *maxButton = new UIW_MAXIMIZE_BUTTON;
    UIW_MINIMIZE_BUTTON *minButton = new UIW_MINIMIZE_BUTTON;
    UIW_SYSTEM_BUTTON *sysButton = new UIW_SYSTEM_BUTTON;
```

```
    window->Add(border);
    window->Add(maxButton);
    window->Add(minButton);
    window->Add(sysButton);
    window->Add(new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER));
    .
    .
    .
}
```

UIW_WINDOW::First

Syntax

```
#include <ui_win.hpp>

UI_WINDOW_OBJECT *UIW_WINDOW::First(void);
```

Remarks

This advanced function returns a pointer to the first window object in the window's list of window objects.

- *returnValue*$_{out}$ is a pointer to the first window object in the window's list of window objects.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a new window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
    // See if any fields are invalid.
    int invalidFields = FALSE;
    for (UI_WINDOW_OBJECT *object = window->First();
        object; object = object->Next())
        if (FlagSet(object->woStatus, WOS_INVALID))
        {
            invalidFields = TRUE;
            break;
        }
    .
    .
    .
}
```

UIW_WINDOW::Last

Syntax

#include <ui_win.hpp>

UI_WINDOW_OBJECT *UIW_WINDOW::Last(void);

Remarks

This function returns a pointer to the last window object in the window's list of window objects.

- *returnValue*$_{out}$ is a pointer to the last window object in the window's list of window objects.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a new window.
    UIW_WINDOW *window = new UIW_WINDOW(0, 0, 40, 10, WOF_NO_FLAGS,
        WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
    // See if any fields are invalid.
    int invalidFields = FALSE;
    for (UI_WINDOW_OBJECT *object = window->Last();
        object; object = object->Previous())
        if (FlagSet(object->woStatus, WOS_INVALID))
        {
            invalidFields = TRUE;
            break;
        }
    .
    .
    .
}
```

UIW_WINDOW::Subtract

Syntax

#include <ui_win.hpp>

void UIW_WINDOW::Subtract(UI_WINDOW_OBJECT **object*);

Remarks

This function removes an object from the UIW_WINDOW class object. This function does not call the destructor associated with the object.

- *object*$_{in}$ is a pointer to the object to be removed from the current window's list of window objects. This argument must be a class object derived from the UI_WINDOW_OBJECT base class.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    extern UI_WINDOW_MANAGER *_windowManager;

    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    UIW_BORDER *border = new UIW_BORDER;
    UIW_MAXIMIZE_BUTTON *maxButton = new UIW_MAXIMIZE_BUTTON;
    UIW_MINIMIZE_BUTTON *minButton = new UIW_MINIMIZE_BUTTON;
    UIW_SYSTEM_BUTTON *sysButton = new UIW_SYSTEM_BUTTON;

    window->Add(border);
    window->Add(maxButton);
    window->Add(minButton);
    window->Add(sysButton);
    window->Add(new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER));
    .
    .
    .
    // Remove the buttons if no selection can be made.
    extern int RemoveButtons(void);
    if (RemoveButtons())
    {
        window->Subtract(maxButton);
        window->Subtract(minButton);
        window->Subtract(sysButton);
    }
    .
    .
    .
}
```

UIW_WINDOW::operator +

Syntax

```
#include <ui_win.hpp>

UIW_WINDOW &UIW_WINDOW::
    operator + (UI_WINDOW_OBJECT *object);
```

Remarks This overload operator adds a window object to the UIW_WINDOW object. This operator overload is equivalent to calling the **UIW_-WINDOW::Add** function, except that it allows the chaining of window object additions to the UIW_WINDOW object.

- *returnValue*$_{out}$ is the UIW_WINDOW reference. Returning the reference to the UIW_WINDOW object allows chaining of the **UIW_WINDOW::operator+** overload operator.

- *object*$_{in}$ is a pointer to the window object derived from the UI_WINDOW_OBJECT base class that is to be added to the window's list of window objects.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    *window
        + new UIW_BORDER
        + new UIW_MAXIMIZE_BUTTON
        + new UIW_MINIMIZE_BUTTON
        + new UIW_SYSTEM_BUTTON
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);

    // Add the window to the screen display.
    extern UI_WINDOW_MANAGER *_windowManager;
    *windowManager + window;
    .
    .
    .
}
```

UIW_WINDOW::operator –

Syntax

```
#include <ui_win.hpp>

UIW_WINDOW &UIW_WINDOW::
    operator – (UI_WINDOW_OBJECT *object);
```

Remarks

This overload operator removes a window object from the window's list of objects. This operator overload is equivalent to calling the **UIW_WINDOW::Subtract** function, except that it allows the chaining of window object subtractions from the UIW_WINDOW class object.

- *returnValue*$_{out}$ is the UIW_WINDOW reference. Returning the reference to the window allows chaining of the **UIW_WINDOW:: operator–** overload operator.

- *object*$_{in}$ is a pointer to the window object derived from the UI_WINDOW_OBJECT base class that is to be removed from the window's list of objects.

Example

```
#include <ui_win.hpp>

ExampleFunction1()
{
    extern UI_WINDOW_MANAGER *_windowManager;

    // Create a window with basic window objects.
    UIW_WINDOW *window = new UIW_WINDOW(0, 1, 67, 11,
        WOF_NO_FLAGS, WOAF_NO_FLAGS);
    UIW_BORDER *border = new UIW_BORDER;
    UIW_MAXIMIZE_BUTTON *maxButton = new UIW_MAXIMIZE_BUTTON;
    UIW_MINIMIZE_BUTTON *minButton = new UIW_MINIMIZE_BUTTON;
    UIW_SYSTEM_BUTTON *sysButton = new UIW_SYSTEM_BUTTON;

    *window
        + border
        + maxButton
        + minButton
        + sysButton
        + new UIW_TITLE("Window 1", WOF_JUSTIFY_CENTER);
    .
    .
    .
    // Remove the buttons if no selection can be made.
    extern int RemoveButtons(void);
    if (RemoveButtons())
    {
        *window - maxButton;
        *window - minButton;
        *window - sysButton;
    }
    .
    .
    .
}
```

INDEX

G

H

I

K

L

M

V

W

GNU Free Documentation License
Version 1.3, 3 November 2008

0. PREAMBLE

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

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

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

1. APPLICABILITY AND DEFINITIONS

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

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

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

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

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

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input

to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

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

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

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

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

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

2. VERBATIM COPYING

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

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

3. COPYING IN QUANTITY

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

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

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy

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

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

4. MODIFICATIONS

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

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

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

You may add a section Entitled "Endorsements", provided it contains

nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

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

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

5. COMBINING DOCUMENTS

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

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

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

6. COLLECTIONS OF DOCUMENTS

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

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

7. AGGREGATION WITH INDEPENDENT WORKS

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

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

8. TRANSLATION

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

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

9. TERMINATION

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

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

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

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

10. FUTURE REVISIONS OF THIS LICENSE

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

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

11. RELICENSING

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

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit

corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

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

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

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

ADDENDUM: How to use this License for your documents

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

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

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

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

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

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