

Zinc Application Framework

5

Reference Manual
(Beta Documentation)

Friday, February 07, 1997

Table of Contents

Introduction. 4

Beta Documentation Introduction 5

Class Reference 6

ZafApplication	7
ZafBignum	12
ZafBignumData	15
ZafBorder	25
ZafButton	28
ZafData	41
ZafDate	48
ZafDevice	51
ZafDisplay	55
ZafElement	75
ZafEventManager	84
ZafFormatData	88
ZafHelpTips	91
ZafHzList	96
ZafImage	101
ZafInteger	105
ZafIntegerData	110
ZafMouse	115
ZafNotification	118
ZafProgressBar	126
ZafPrompt	132
ZafPullDownItem	137
ZafPullDownMenu	143
ZafPopUpItem	146
ZafPopUpMenu	153
ZafReal	156
ZafRealData	159

ZafScrollBar	164
ZafStatusBar	168
ZafString	171
ZafStringData	181
ZafText.	191
ZafTime	198
ZafToolBar	201
ZafTreeItem.	205
ZafTreeList	213
ZafUTime.	220
ZafVtList	223
ZafWindow	227
ZafWindowManager	246
ZafWindowObject	251

Appendices 344

Event Definitions	345
Zinc Coding Standards.	362

Index 371

Introduction

Beta Documentation

Introduction

During the beta testing period, this reference manual will be the primary source of information on the use of the Zinc Application Framework 5 class library.

Since no tutorial or “Getting Started” information is yet available, developers should familiarize themselves with the internal functions of important “fundamental” classes of ZAF 5. These fundamental classes are all base classes (though not necessarily root classes) that define core functionality used by other objects throughout the application framework.

Developers are strongly urged to completely read and understand the reference chapters for the following classes (listed roughly in order of importance):

- Attribute Matrices (found separately in “z5matrix.pdf”)
- ZafWindowObject
- ZafWindow
- Event Definitions (Appendix A)
- ZafElement
- ZafData
- ZafApplication

As additional “fundamental” classes are documented they will be added to this list. Documentation updates are expected approximately weekly throughout the beta test period.

For information not found in this manual, please contact Zinc technical support via email at “zincbugs@zinc.com”.

Class Reference

ZafApplication

AddStaticModule	argc	argv
Control	LinkMain	Main

ZafApplication provides a platform independent entry point for ZAF applications called `ZafApplication::Main()` and a mechanism for managing global objects used by these applications.

Execution of a C/C++ application begins with a call to `main()` or `WinMain()` (for Microsoft Windows applications). ZAF includes `main()` or `WinMain()` in its interface library so that it need not be provided by the ZAF developer. Instead, the ZAF developer provides `ZafApplication::Main()`.

A `ZafApplication` class instance is created and destroyed within the `main()` or `WinMain()` function provided in the Zinc Application Framework library. This allows `ZafApplication` to perform initialization of global objects before `ZafApplication::Main()` is called, and to perform clean-up after. The following global pointers are initialized by the `ZafApplication` constructor: `zafDisplay`, `zafEventManager`, `zafWindowManager`, `zafLanguage`, `zafLocale`, and `zafCodeSet`.

`ZafApplication` also provides an event processing loop, `ZafApplication::Control()`. This loop controls the event flow for an application by getting events from the ZAF event manager and passing them to the window manager. It does not exit until program termination.

Declaration `#include <z_app.hpp>`

Inheritance Root Class

Constructor The `ZafApplication` constructor initializes global objects for use by ZAF applications. The constructor is called when a `ZafApplication` class instance is created inside of the `main()` or `WinMain()` function found in the library. The `ZafApplication` constructor should not be called by the ZAF Developer.

`ZafApplication`(int argc, char **argv);

The *argc* and *argv* parameters are equivalent to the parameters passed to a program's `main()` function. *argc* is an integer containing the number of arguments passed to the program, and *argv* is an array of strings representing the arguments passed to the program. *argv[0]* always contains the full path name of the program being executed. Note: Under Microsoft Windows, *argc* and *argv* are synthesized from the parameters passed to `WinMain()`.

Several global pointers are initialized by the constructor:

- `zafDisplay`, a pointer to the display;
- `zafEventManager`, a pointer to the event manager;
- `zafWindowManager`, a pointer to the window manager;
- `zafLanguage`, a pointer to a `ZafLanguageData` object containing information about active language;
- `zafLocale`, a pointer to a `ZafLocaleData` object containing information about the active locale;
- `zafCodeSet`, a pointer to a `ZafCodeSetData` object containing information about the active code page.

In addition, `ZafKeyboard`, `ZafMouse`, and `ZafCursor` devices are added to the event manager. See `ZafEventManager` for additional information.

Destructor

```
~ZafApplication(void);
```

The `ZafApplication` destructor destroys all objects allocated by the constructor, and instructs any static modules attached by `AddStaticModule()` to free their data.

Members

```
static ZafStaticModule AddStaticModule(ZafStaticModule
    module);
```

`AddStaticModule()` is a mechanism for allowing objects to designate data to be deleted during program termination. It is used internally by the interface library, and is not intended to be used by the ZAF developer. *module* is a pointer to a function that is to be called during program termination. The definition of `ZafStaticModule` is shown below:

```
typedef void (*ZafStaticModule)(bool);
```

The `ZafApplication` destructor calls all functions added with `AddStaticModule()` with a parameter of `true`. At this time all data should be destroyed because program termination is imminent.

```
int argc;
```

`argc` is an integer value specifying the number of arguments passed to the program at start-up. These arguments can be found in the `argv` member array.

```
ZafIChar **argv;
```


argv is an array of strings representing the arguments passed to the program at start-up. argv[0] always contains the full path name of the program being executed. The number of arguments passed to the program is contained in argc.

```
ZafEventType Control(void);
```

Control() is an event processing loop suitable for most applications. It removes events from the event manager's event queue and passes them to the window manager. The Control() loop does not terminate until an S_EXIT is removed from the event queue or until no windows are left on the window manager.

Control() returns either S_EXIT, or S_NO_OBJECT. S_EXIT indicates that an application is terminating because an S_EXIT message was processed, while S_NO_OBJECT indicates that an application is terminating because there are no windows left to process events.

The code for ZafApplication::Control() is shown below:

```
ZafEventType ZafApplication::Control(void)
{
    // Wait for user response.
    ZafEventStruct event;
    ZafEventType ccode;
    do
    {
        zafEventManager->Get(event, Q_NORMAL);
        ccode = zafWindowManager->Event(event);
    } while (ccode != S_EXIT && ccode != S_NO_OBJECT);

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

It is sometimes desirable to process special events or to perform other special actions inside of the control loop. In such cases, a custom control loop can be written and the code above can be used as a starting point. See ZafEventManager and ZafWindowManager for details on relevant functions.

```
void LinkMain(void);
```

LinkMain() is a stub used to overcome a deficiency with some linkers. If main() (or WinMain()) is not found in an object file, most linkers will search the libraries to find it. Some, however, do not. Since main() is included in the interface library, linkers that do not look for it there may not find it and gener-

ate an error. ZAF developers who run into this error can resolve it by adding a call to `LinkMain()` inside of `ZafApplication::Main()`.

```
int Main(void);
```

`Main()` is the entry point for a ZAF application, and must be provided by the ZAF developer. Execution of a C/C++ application begins with a call to `main()` (or `WinMain()` for Microsoft Windows applications). ZAF includes `main()` or `WinMain()` in its interface library so that it need not be provided by the ZAF developer. Instead, the ZAF developer provides the `ZafApplication::Main()` function. This provides a platform independent entry point for all ZAF applications.

The code below shows the `Main()` function for a simple ZAF application:

```
int ZafApplication::Main()  
{  
    zafWindowManager->Add(new HellowWindow());  
    Control();  
    return (0);  
}
```

The `argc` and `argv` member variables can be used in place of the parameters normally passed to a C/C++ `main()` function. These variables are initialized in the `ZafApplication` constructor. (See `argc` and `argv`.)

ZafBignum

Event	BignumData
-------	------------

ZafBignum is a single-line arbitrary precision numeric object designed for financial and scientific use. ZafBignum allows user input through the keyboard. ZafBignum is fully internationalized to display and input using any format.

All ZafBignum objects refer to data contained in a ZafBignumData object (refer to this class for additional essential information).

Declaration `#include <z_bnum1.hpp>`

Inheritance `ZafBignum : ZafString : ZafWindowObject : ZafElement`

Constructors All ZafBignum constructors initialize the member variables associated with an instantiated ZafBignum object. The default values set by the ZafBignum and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafBignum. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafBignum	
<code>BignumData()</code>	<code>null</code>
ZafString	
<code>LowerCase()</code>	<code>false[†]</code>
<code>Password()</code>	<code>false[†]</code>
<code>StringData()</code>	<code>null[†]</code>
<code>UpperCase()</code>	<code>false[†]</code>
<code>VariableName()</code>	<code>false[†]</code>
ZafElement	
<code>ClassID()</code>	<code>ID_ZAF_BIGNUM</code>
<code>ClassName()</code>	<code>"ZafBignum"</code>

ZafBignum(int left, int top, int width, long value);
ZafBignum(int left, int top, int width, double value);

These constructors are useful in straight-code situations, particularly if you wish the `ZafBignum` object to create, maintain and destroy its own `ZafBignumData` object automatically. *left*, *top* and *width* specify the position and size of the object on its parent. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired.

value is the value you wish to initially appear in the new `ZafBignum` object and can be either an integral or a floating point value.

```
ZafBignum(int left, int top, int width, ZafBignumData  
          *bignumData = ZAF_NULLP(ZafBignumData));
```

This constructor is useful in straight-code situations where a `ZafBignumData` object has already been created. This constructor could be used when manually maintaining a *bignumData* object, rather than having the `ZafBignum` class create and maintain the data object automatically. For more information on using `ZafBignumData` objects, see the chapter on `ZafBignumData`. See the previous constructor for a description of *left*, *top* and *width* parameters.

```
ZafBignum(const ZafBignum &copy);
```

The copy constructor calls the overloaded `Duplicate()` to create a new `ZafBignum` object and initialize its data from *copy*. If the original data objects are `StaticData()` then the new `ZafBignum` object simply points to the original data, otherwise `StaticData()` copies are made.

```
ZafBignum(const ZafIChar *name, ZafObjectPersistence  
          &persist);
```

The final constructor is used for persistence. Refer to `ZafWindow` for more information, since most persistence is done at the `ZafWindow` level.

Sample `ZafBignum` creation techniques follow:

```
// Create a sample window with bignum objects.  
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);  
// Create bignum objects and pass in the values directly.  
window1->Add(new ZafBignum(0, 1, 25, 3.1415927));  
window1->Add(new ZafBignum(0, 2, 25, 100));  
...  
// Create a sample window with bignum objects.  
ZafWindow *window2 = new ZafWindow(10, 10, 40, 10);  
// Create bignum data objects.  
ZafBignumData *bigData1 = new ZafBignumData(3.1415927);  
ZafBignumData *bigData2 = new ZafBignumData(100);
```

```
// Create bignums that use the data previously created.
window2->Add(new ZafBignum(0, 1, 25, bigData1));
window2->Add(new ZafBignum(0, 2, 25, bigData2));
```

Destructor

```
virtual ~ZafBignum(void);
```

The destructor is used to free the memory associated with a ZafBignum object, including all data objects that are Destroyable(). It chains to the ZafString, ZafWindowObject, and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafBignum object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~~ZafWindow().

Members

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function receives all events that get sent to the ZafBignum object and either handles them or passes them to ZafString, its immediate base class. See ZafWindowObject for more information.

ZafBignum specifically handles the following events:

Event	Description
S_COPY_DATA	causes the object to copy event.windowObject's BignumData() if event.windowObject is a ZafBignum object
S_SET_DATA	causes the object to create a new BignumData() object, then copy into it event.windowObject's BignumData() if event.windowObject is non-null and is a ZafBignum object

```
ZafBignumData *BignumData(void) const;
virtual ZafError SetBignumData(ZafBignumData
    *bignumData);
```

*BignumData() contains the actual information used by ZafBignum. The BignumData() object may be used by one or more ZafBignum objects, or other objects. If shared, all associated ZafBignum objects will be notified when the BignumData() changes. For more information on data sharing in ZAF, see ZafDataManager.

BignumData() returns a pointer to the BignumData() object associated with the ZafBignum object. The return value for SetBignumData() is normally ZAF_ERROR_NONE. See the constructor code snippet for an example using ZafBignumData objects with ZafBignum.

ZafBignumData

Clear	double	FormattedText
IValue	long	RValue
SetBignum	operator -	operator --
operator +	operator ++	operator *
operator /	operator %	operator =
operator +=	operator -=	operator *=
operator /=	operator %=	operator ==
operator !=	operator <	operator <=
operator >	operator >=	

ZafBignumData objects can be used to store and manipulate large numeric values to an arbitrary level of precision. Internally, ZafBignumData uses BCD (Binary Coded Decimal) representation to store and manipulate large numbers.

ZafBignumData combines number encapsulation with data and object notification from ZafData. It is most often used in conjunction with the ZafBignum user interface object but may be used as a stand-alone object if desired.

Numeric precision of ZafBignumData defaults to thirty digits to the left of the decimal and eight to the right. If desired, this precision may be changed by rebuilding the library after changing two constants (ZAF_NUMBER_WHOLE, ZAF_NUMBER_DECIMAL) in z_bnum.hpp.

All ZafData objects may make use of printf-style formatting and parsing arguments during string operations. In addition to printf arguments normally used by real and integer data types, ZafBignumData adds additional custom arguments (conversion characters) to those normally available to the printf family of functions:

Format conversion char	Description
"B"	Formats the printf() argument as a bignum. The stream is right-padded with zeros to the full precision of a bignum. Leading zeros are suppressed.
Parse conversion char	
"B"	Parses a scanf() stream component as a bignum and stores the resulting value in the supplied bignum argument.

Refer to standard library documentation for detailed information on printf functions and conversion characters.

Declaration	<code>#include <z_bnum.hpp></code>
Inheritance	<pre>ZafBignumData : ZafFormatData : ZafData : (ZafNotification,ZafElement)</pre>
Constructors	<p>ZafBignumData constructors initialize the member variables associated with a new ZafBignumData object and allocate space to hold the bignum data.</p> <p>The default values set by ZafBignumData follow, if they are overridden from those set by base class constructors:</p>

Member Initializations

ZafBignumData

<code>IValue()</code>	<code>(varies by constructor)</code>
<code>RValue()</code>	<code>(varies by constructor)</code>

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_BIGNUM_DATA</code>
<code>ClassName()</code>	<code>"ZafBignumData"</code>

ZafBignumData(void);

The basic constructor allocates a ZafBignumData instance and initializes its value to 0.

ZafBignumData(long value);

ZafBignumData(double value);

These constructors allocate a ZafBignumData instance and initialize its contents to *value*. The data is automatically converted to its equivalent BCD representation and stored in an internal buffer.

ZafBignumData(const ZafIChar *string, const ZafIChar
*format = ZAF_NULLP(ZafIChar));

This constructor allocates a ZafBignumData instance and initializes its value to the numeric equivalent of *string*. The conversion uses the printf-style specifier *format* to interpret the string. If *format* is null ZafBignumData uses its locale-specific default format.

ZafBignumData(const ZafBignumData ©);

This constructor is the copy constructor. It allocates a new `ZafBignumData` instance and copies all member data from *copy*.

```
ZafBignumData(const ZafIChar *name, ZafDataPersistence  
                  &persist);
```

This constructor is the persistent constructor. It allocates a new `ZafBignumData` instance and reads most member data from the *name* directory of the persistent data file referred to by *persist*. The `StringID()` of the new data is *name*.

```
// Sample ZafBignumData creation techniques  
double value = 123.45;  
ZafBignumData bignum1(value);  
ZafBignumData copyBignum = bignum1;  
ZafBignumData zeroBignum;
```

Destructor

```
virtual ~ZafBignumData(void);
```

The destructor is used to free the memory associated with an instantiated `ZafBignumData` object. Unless `StaticData()` is true a `ZafBignumData` object is usually destroyed automatically when all `ZafBignum` objects that refer to it are destroyed.

Members

```
virtual void Clear(void);
```

`Clear()` sets the value of a `ZafBignumData` object to zero.

```
operator double();
```

See `IValue()`.

```
virtual int FormattedText(ZafIChar *buffer, int  
                          maxLength, const ZafIChar *format = 0) const;
```

`FormattedText()` fills *buffer* with a string representation of the `ZafBignumData` using the `printf`-style specifier *format* to build the string. A locale-specific default format is used if *format* is not included. Buffer contents will be truncated if they exceed *maxLength*. `FormattedText()` returns the integer value it receives from its call to `Sprintf()`.

```
// Show various results of FormattedText().  
ZafIChar buffer[256];
```

```
ZafBignumData bignum(12345.6789);
```

```

bignum.FormattedText(buffer, sizeof(buffer));
printf("bignum - %s\n", buffer);

bignum.FormattedText(buffer, sizeof(buffer), "%ld");
printf("integer - %s\n", buffer);

bignum.FormattedText(buffer, sizeof(buffer), "%5.2lf");
printf("real - %s\n", buffer);

=====
bignum - 12345.67890000
integer - 12346
real - 12345.68

```

```

long IValue(void) const;
operator long();

```

`IValue()` returns the integer equivalent of a `ZafBignumData` object as a long. The convenience operator `long()`, which returns `IValue()`, is more commonly used.

In the examples below, note that the compiler must be able to resolve static values to longs or doubles to properly interpret the overloaded operator.

```

// Perform numerical operations on a bignum.

ZafBignumData bignum(12345.6789);
bignum = bignum.IValue() + 1.0;
printf("bignum - %b\n", bignum);

bignum = bignum - 0.1;
printf("bignum - %b\n", bignum);

=====
bignum - 12346.00000000
bignum - 12345.90000000

```

```

double RValue(void) const;
operator double();

```

`RValue()` returns the real value of a `ZafBignumData` object as a double. The convenience operator `double()`, which returns `RValue()`, is more commonly used.

In the examples below, note that the compiler must be able to resolve static values to longs or doubles to properly interpret the overloaded operator. Compare the following code to the `IValue()` code above.

```
// Perform numerical operations on a bignum.
```

```
ZafBignumData bignum(12345.6789);  
bignum = bignum.RValue() + 1.0;  
printf("bignum - %B\n", bignum);
```

```
bignum = bignum - 0.1;  
printf("bignum - %B\n", bignum);
```

```
=====  
bignum - 12346.67890000  
bignum - 12346.57890000
```

```
virtual ZafError SetBignum(long value);  
virtual ZafError SetBignum(double value);  
virtual ZafError SetBignum(const ZafIChar *buffer, const  
    ZafIChar *format);  
virtual ZafError SetBignum(const ZafBignumData &number);
```

SetBignum() functions set the value of the **ZafBignumData** object from various numeric input types, *scanf*-style specifiers *buffer* and *format* (if not included, a locale-specific default format is used), or another bignum. Refer to **FormattedText()** for more information on bignum/string conversions. The overloaded operator **=** offers similar functionality to **SetBignum()**.

```
ZafBignumData operator-(const ZafBignumData &number1,  
    const ZafBignumData &number2);  
ZafBignumData operator-(const ZafBignumData &number, long  
    value);  
ZafBignumData operator-(long value, const ZafBignumData  
    &number);  
ZafBignumData operator-(const ZafBignumData &number,  
    double value);  
ZafBignumData operator-(double value, const ZafBignumData  
    &number);
```

These operators allow simple inline subtraction operations involving **ZafBignumData** objects, longs and doubles. The following code shows the proper use of these operators:

```
// Create some ZafBignumData objects.  
ZafBignumData big1(1000), big2(500000), big3(2000000);
```

```
// Create another ZafBignumData object and modify the others.
ZafBignumData big4 = big2 - big1;
big3 -= big4;
big1--;
// Compare two of the objects.
if (big4 > big3)
    return (-1);
```

```
ZafBignumData operator--(void);
ZafBignumData operator--(int);
```

These pre- and post-operators decrement the ZafBignumData object's value by 1. See operator- for sample code.

```
ZafBignumData operator+(const ZafBignumData &number1,
    const ZafBignumData &number2);
ZafBignumData operator+(const ZafBignumData &number, long
    value);
ZafBignumData operator+(long value, const ZafBignumData
    &number);
ZafBignumData operator+(const ZafBignumData &number,
    double value);
ZafBignumData operator+(double value, const ZafBignumData
    &number);
```

These operators allow simple inline addition operations involving ZafBignumData objects, longs and doubles. See operator+ for sample code.

```
ZafBignumData operator++(void);
ZafBignumData operator++(int);
```

These pre- and post-operators increment the ZafBignumData object's value by 1. See operator+ for sample code.

```
ZafBignumData operator*(const ZafBignumData &number1,
    const ZafBignumData &number2);
ZafBignumData operator*(const ZafBignumData &number, long
    value);
ZafBignumData operator*(long value, const ZafBignumData
    &number);
```

```
ZafBignumData operator*(const ZafBignumData &number,  
    double value);  
ZafBignumData operator*(double value, const ZafBignumData  
    &number);
```

These operators allow simple inline multiplication operations involving ZafBignumData objects, longs and doubles. See operator- for sample code.

```
ZafBignumData operator/(const ZafBignumData &number1,  
    const ZafBignumData &number2);  
ZafBignumData operator/(const ZafBignumData &number, long  
    value);  
ZafBignumData operator/(long value, const ZafBignumData  
    &number);  
ZafBignumData operator/(const ZafBignumData &number,  
    double value);  
ZafBignumData operator/(double value, const ZafBignumData  
    &number);
```

These operators allow simple inline division operations involving ZafBignumData objects, longs and doubles. See operator- for sample code.

```
ZafBignumData operator%(const ZafBignumData &number1,  
    const ZafBignumData &number2);  
ZafBignumData operator%(const ZafBignumData &number, long  
    value);  
ZafBignumData operator%(long value, const ZafBignumData  
    &number);  
ZafBignumData operator%(const ZafBignumData &number,  
    double value);  
ZafBignumData operator%(double value, const ZafBignumData  
    &number);
```

These operators allow simple inline modulus operations involving ZafBignumData objects, longs and doubles. See operator- for sample code.

```
ZafBignumData &operator=(const ZafBignumData &number);  
ZafBignumData &operator=(long value);  
ZafBignumData &operator=(double value);
```

These operators assign the ZafBignumData object's value to the input *value* which may be a long, double, or another ZafBignumData. See operator- for sample code.

```
ZafBignumData &operator+=(const ZafBignumData &number);  
ZafBignumData &operator+=(long value);  
ZafBignumData &operator+=(double value);
```

These operators increment the ZafBignumData object's value by the input *value*. See operator- for sample code.

```
ZafBignumData &operator-=(const ZafBignumData &number);  
ZafBignumData &operator-=(long value);  
ZafBignumData &operator-=(double value);
```

These operators decrement the ZafBignumData object's value by the input *value*. See operator- for sample code.

```
ZafBignumData &operator*=(const ZafBignumData &number);  
ZafBignumData &operator*=(long value);  
ZafBignumData &operator*=(double value);
```

These operators multiply the ZafBignumData object's value by the input value and use the resulting product to set the ZafBignumData object's *value*. See operator- for sample code.

```
ZafBignumData &operator/=(const ZafBignumData &number);  
ZafBignumData &operator/=(long value);  
ZafBignumData &operator/=(double value);
```

These operators divide the ZafBignumData object's value by the input value and use the resulting quotient to set the ZafBignumData object's value. *value* is used as the divisor. See operator- for sample code.

```
ZafBignumData &operator%=(const ZafBignumData &number);  
ZafBignumData &operator%=(long value);  
ZafBignumData &operator%=(double value);
```

These operators work exactly the same as the /= operators above, but use the operation's remainder to set the ZafBignumData's value. See operator- for sample code.

```
bool operator==(const ZafBignumData &number1, const
                ZafBignumData &number2)
bool operator==(const ZafBignumData &number, long value)
bool operator==(long value, const ZafBignumData &number)
bool operator==(const ZafBignumData &number, double
                value)
bool operator==(double value, const ZafBignumData
                &number)
bool operator!=(const ZafBignumData &number1, const
                ZafBignumData &number2)
bool operator!=(const ZafBignumData &number, long value)
bool operator!=(long value, const ZafBignumData &number)
bool operator!=(const ZafBignumData &number, double
                value)
bool operator!=(double value, const ZafBignumData
                &number)
```

These operators allow simple inline equivalence comparisons between ZafBignumData objects, longs and doubles. See operator- for sample code.

```
bool operator<(const ZafBignumData &number1, const
               ZafBignumData &number2)
bool operator<(const ZafBignumData &number, long value)
bool operator<(long value, const ZafBignumData &number)
bool operator<(const ZafBignumData &number, double value)
bool operator<(double value, const ZafBignumData &number)
bool operator<=(const ZafBignumData &number1, const
                ZafBignumData &number2)
bool operator<=(const ZafBignumData &number, long value)
bool operator<=(long value, const ZafBignumData &number)
bool operator<=(const ZafBignumData &number, double
                value)
bool operator<=(double value, const ZafBignumData
                &number)
bool operator>(const ZafBignumData &number1, const
                ZafBignumData &number2)
bool operator>(const ZafBignumData &number, long value)
bool operator>(long value, const ZafBignumData &number)
bool operator>(const ZafBignumData &number, double value)
```

```
bool operator>(double value, const ZafBignumData &number)
bool operator>=(const ZafBignumData &number1, const
    ZafBignumData &number2)
bool operator>=(const ZafBignumData &number, long value)
bool operator>=(long value, const ZafBignumData &number)
bool operator>=(const ZafBignumData &number, double
    value)
bool operator>=(double value, const ZafBignumData
    &number)
```

These operators allow simple inline magnitude comparisons between ZafBignumData objects, longs and doubles. See `operator-` for sample code.

ZafBorder

Width

The ZafBorder object may only be added to a ZafWindow. The ZafBorder is the border decoration on a ZafWindow, and is generally drawn by the environment. The width of the ZafBorder is therefore determined by the environment, and may not be modified by the programmer in most cases.

Note that ZafPrompt utilizes the Bordered() attribute differently than other window objects. With ZafPromp, the Bordered() attribute means the object is to be aligned with a Bordered() field.

Declaration

#include <z_border.hpp>

Inheritance

ZafBorder : ZafWindowObject : ZafElement

Constructors

All ZafBorder constructors initialize the member variables associated with an instantiated ZafBorder object. The default values set by the ZafBorder and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafBorder. “†”Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafBorder

Width()

Environment specific

ZafWindowObject

AcceptDrop()

false†

Bordered()

false†

CopyDraggable()

false†

Disabled()

false†

Focus()

false†

HelpContext()

null†

HelpObjectTip()

null†

LinkDraggable()

false†

MoveDraggable()

false†

Noncurrent()

true†

ParentDrawBorder()

false†

ParentDrawFocus()

false†

ParentPalette()

false†

QuickTip()

null†

Member Initializations

RegionType()	ZAF_OUTSIDE_REGION [†]
Selected()	false [†]
SupportObject()	true [†]
UserFunction()	null [†]

ZafElement

ClassID()	ID_ZAF_BORDER
ClassName()	"ZafBorder"

ZafBorder(void);

The first constructor is useful in straight-code situations, and is used to instantiate a ZafBorder object to be added to a ZafWindow object.

ZafBorder(const ZafBorder ©);

The copy constructor is used in conjunction with the overloaded Duplicate() function. It accepts another ZafBorder object and copies the object's information.

ZafBorder(const ZafIChar *name, ZafObjectPersistence &persist);

The final constructor is used for persistence. The parameters and values of this constructor are deferred to the ZafWindow section of this manual, since most persistence is done at the ZafWindow level.

Below is a simple example:

```
// Create a sample window.
ZafWindow *window = new ZafWindow(0, 0, 40, 10);
ZafBorder *border = new ZafBorder;
window->Add(border);
```

Destructor

```
virtual ~ZafBorder(void);
```

The destructor is used to free the memory associated with a ZafBorder object. It chains to the ZafWindowObject and ZafElement destructors. Generally, the programmer will not directly destroy a ZafBorder object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~~ZafWindow().

Members

```
virtual int Width(void);  
virtual ZafError SetWidth(int width);
```

Since a ZafBorder object is normally a decoration added to the window by the host environment, the width of a ZafBorder is usually dictated by the environment, and the programmer may not change it. The Width() function returns the width of the ZafBorder object.

SetWidth() allows the programmer to change the width of the ZafBorder object on environments that allow it. Normally the width of a border is set by the operating system.

ZafButton

AllowDefault	AllowToggling	AutoRepeatSelection
AutoSize	BitmapData	ButtonType
ClearImage	ClearText	ConvertCoordinates
CoordinateType	Depressed	Depth
Event	HotKeyChar	HotKeyIndex
HzJustify	OSUpdatePalettes	Region
Selected	SelectOnDoubleClick	SelectOnDownClick
SendMessage	SendMessageWhen-Selected	StringData
Text	Value	Visible
VtJustify		

The ZafButton class provides support for native buttons, 3-d buttons, flat buttons, toolbar buttons, radio buttons and check boxes. By default, ZafButton objects support the native environment look-and-feel, but by deriving from ZafButton, the programmer may provide custom button objects.

The ZafButton class is used as a base class for other selectable action classes such as ZafIcon and ZafPopUpItem. These classes inherit much of the base functionality provided by ZafButton.

Declaration `#include <z_button.hpp>`

Inheritance `ZafButton : ZafWindowObject : ZafElement`

Constructors All ZafButton constructors initialize the member variables associated with an instantiated ZafButton object. Default values set by the ZafButton constructor are listed below, as well as values overridden from those set by base class constructors.

Member Initializations

ZafButton

AllowDefault()	false
AllowToggling()	false
AutoRepeatSelection()	false
AutoSize()	true
BitmapData()	ZAF_NULLP(ZafBitmapData)
ButtonType()	ZAF_NATIVE_BUTTON
Depressed()	false
Depth()	2
HotKeyChar()	0
HotKeyIndex()	-1

Member Initializations

HzJustify()	ZAF_HZ_CENTER
SelectOnDoubleClick()	false
SelectOnDownClick()	false
SendMessageWhen- Selected()	false
StringData()	ZAF_NULLP(ZafStringData)
Value()	0
VtJustify()	ZAF_VT_CENTER

ZafElement

ClassID()	ID_ZAF_BUTTON
ClassName()	"ZafButton"

```
ZafButton(int left, int top, int width, int height, const  
    ZafIChar *text, ZafBitmapData *bitmapData =  
    ZAF_NULLP(ZafBitmapData), ZafButtonType buttonType =  
    ZAF_NATIVE_BUTTON);
```

This constructor is useful in straight-code situations, particularly if you wish the ZafButton object to create, maintain and destroy its own ZafStringData object automatically. Simply pass the string you wish to appear in the new ZafButton object into the *text* parameter. *left* and *top* specify the position where the left and top of the button will be placed on its parent. *width* and *height* specify the width and height of the button. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired.

bitmapData specifies the bitmap to be displayed on the button object. The *buttonType* parameter specifies the type of button to be created. See ButtonType() for more information on button types.

```
ZafButton(int left, int top, int width, int height,  
    ZafStringData *stringData, ZafBitmapData *bitmapData =  
    ZAF_NULLP(ZafBitmapData), ZafButtonType buttonType =  
    ZAF_NATIVE_BUTTON);
```

This constructor is also useful in straight-code situations, particularly if you have already created a ZafStringData object to be associated with the button. This constructor could be used to maintain string data pieces yourself, rather than having the ZafButton class create and maintain the string data pieces automatically. For example, to maintain a database of ZafStringData objects and tie them into ZafButton objects, maintain your own ZafStringData objects and

create ZafButton objects using your ZafStringData objects. For more information on using ZafStringData objects, see ZafStringData. The *stringData* parameter specifies the string data object to be associated with the button. Either the *bitmapData* or the *stringData* parameter may be null, to provide a string-only or bitmap-only button. Otherwise, this constructor (and parameters) is the same as the first.

```
ZafButton(const ZafButton &copy);
```

The copy constructor is used in conjunction with the overloaded Duplicate() function. It accepts another ZafButton object (*copy*) and copies the object's information. It is important to note that if the data objects (such as StringData()) are StaticData(), then the new ZafButton object simply points to the original data objects. If the data objects are not StaticData(), then a copy is made for the new ZafButton object. This behavior allows a programmer to use static data for more than one ZafButton object.

```
ZafButton(const ZafIChar *name, ZafObjectPersistence
           &persist);
```

The final constructor is used for persistence. The parameters and values of this constructor are deferred to the ZafWindow section of this manual, since most persistence is done at the ZafWindow level.

Here are some code snippets that show various ZafButton object creation techniques.

```
// Create a sample window with some button objects.
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);

// Add the buttons to the window.
extern ZafBitmapData *bitmapData;
window1->Add(new ZafButton(1, 4, 15, 1, bitmapData,
                          ZAF_NULLP(ZafStringData)));
window1->Add(new ZafButton(20, 4, 15, 1,
                          ZAF_NULLP(ZafBitmapData), "Button2"));
...
// Create a sample window with a group of radio buttons.
ZafWindow *window2 = new ZafWindow(10, 10, 40, 10);
ZafGroup *group = new ZafGroup(1, 1, 30, 5, "Group");

// Add the radio buttons to the group.
group->SetAutoSelect(true);
group->Add(new ZafButton(1, 0, 25, 1, ZAF_NULLP(ZafBitmapData),
                      "Choice 1", ZAF_RADIO_BUTTON));
```

```
group->Add(new ZafButton(1, 1, 25, 1, ZAF_NULLP(ZafBitmapData),
    "Choice 2", ZAF_RADIO_BUTTON));
group->Add(new ZafButton(1, 2, 25, 1, ZAF_NULLP(ZafBitmapData),
    "Choice 3", ZAF_RADIO_BUTTON));
group->Add(new ZafButton(1, 3, 25, 1, ZAF_NULLP(ZafBitmapData),
    "Choice 4", ZAF_RADIO_BUTTON));
// Finally, add the group of radio buttons to the window.
window2->Add(group);
```

Destructor

```
virtual ~ZafButton(void);
```

The destructor is used to free the memory associated with a ZafButton object, including all the data object pieces (such as StringData()) that are Destroyable(). It chains to the ZafWindowObject and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafButton object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
bool AllowDefault(void) const;
virtual bool SetAllowDefault(bool allowDefault);
```

A default button is the button that will be selected when the user types <Enter> or <Return>. The default button commonly has a bold border around it to indicate to the user that it is the default button. If a ZafButton object is to display as the default button on a window, AllowDefault() must be true; otherwise, the default border may not be drawn, even when the button is the default button. The default value of this attribute is false, but the user may call SetAllowDefault() to change it.

In order to specify a button as the default button, ZafWindow::SetDefaultButton() must also be called. ZafWindow::SetDefaultButton() tells the parent window which button is to function as the default button, but does not set the AllowDefault() attribute on the button. SetAllowDefault(true) must be called for all the buttons that are aligned together, if one of them is to be the default button. The AllowDefault() attribute affects the visual aspect of a button, and ZafWindow::SetDefaultButton() affects which button functions as the default button on the window. Refer to ZafWindow::DefaultButton() for more information.

In edit mode, such as when modifying a ZafButton object in Zinc Designer, the ZafButton object displays a large border, indicating the largest area the object may occupy in any environment. Currently, the largest area a default button will occupy is under Motif.

The following code provides a brief example:

```
// Get the attribute.
if (object->AllowDefault())
    break;
...
// Create the OK button as the default button.
ZafButton *button1 = new ZafButton(1, 4, 12, 1,
    ZAF_NULLP(ZafBitmapData), "OK");
ZafButton *button2 = new ZafButton(15, 4, 12, 1,
    ZAF_NULLP(ZafBitmapData), "Cancel");
button1->SetAllowDefault(true);
button2->SetAllowDefault(true);
window->Add(button1);
window->Add(button2);
window->SetDefaultButton(button1);
```

```
bool AllowToggling(void) const;
virtual bool SetAllowToggling(bool allowToggling);
```

A toggle button may stay in the selected or de-selected state. A check box and a radio button may be considered toggle buttons, since they keep their selection state. When the end-user clicks a toggle button, it toggles its selection state. For example, if the toggle button is in its normal state, and the user selects it, it will display itself in its selected state. For a check box, that means a mark inside its box. For a normal toggle button, the selected button stays depressed, so that the user knows that it is currently selected. AllowToggling() is false by default, but the user may call SetAllowToggling() to change it.

The following code shows how to create a toggle button:

```
// Create a toggle button.
ZafButton *button = new ZafButton(1, 4, 12, 1,
    ZAF_NULLP(ZafBitmapData), "Toggle Button");
button->SetAllowToggling(true);
```

```
bool AutoRepeatSelection(void) const;
virtual bool SetAutoRepeatSelection(bool
    autoRepeatSelection);
```


If `AutoRepeatSelection()` is true, as long as the button is depressed by the user, the button performs its action repeatedly, at a delay rate specified by the environment. See `ZafWindowObject::initialDelay` and `ZafWindowObject::repeatDelay` for more information on the delay rate. For example, if the button is also `SendMessageWhenSelected()`, then the button will repeatedly post events on the event manager's queue. If a user function is specified for the button, the user function will be repeatedly called. A scroll bar's arrow button is an example of an object with this behavior. While the user selects the arrow button, the scroll bar's thumb button moves, scrolling the associated object, such as a list. The default value of this attribute is false, but the user may call `SetAutoRepeatSelection()` to change it.

The following code shows how to create an auto-repeat button:

```
// Create a button that automatically repeats during selection.
ZafButton *button = new ZafButton(1, 4, 12, 1,
    ZAF_NULLP(ZafBitmapData), "Repeat Button");
button->SetSendMessageWhenSelected(true);
button->SetValue(USER_ACTION_EVENT);
button->SetAutoRepeatSelection(true);
```

```
bool AutoSize(void) const;
virtual bool SetAutoSize(bool autoSize);
```

If `AutoSize()` is true, the button will automatically grow (up) vertically to the size it needs to be to display all of its data. If the button does not contain bitmap data, then the button is sized vertically to the default size used by the native environment, depending on the system font. For example, an “OK” button that is `AutoSize()` will be the same size as a normal “OK” button that you would see in the native environment's system utility applications. The default value of this attribute is true, but the user may call `SetAutoSize()` to change it.

```
ZafBitmapData *BitmapData(void) const;
virtual ZafError SetBitmapData(ZafBitmapData
    *bitmapData);
```

The `BitmapData()` object is where the actual bitmap data is stored. The `BitmapData()` may be shared among several `ZafButton` objects (perhaps to save memory by allowing many button objects to use the same bitmap data), or it may belong to a single `ZafButton` object. If shared among several `ZafButton` objects, all the associated `ZafButton` objects will be updated when the `BitmapData()` changes. `SetBitmapData()` may be used to associate a `BitmapData()` object with a `ZafButton` object. For more information on data sharing in ZAF, see `ZafDataManager`.

The return value for `BitmapData()` is a pointer to the `BitmapData()` object associated with the `ZafButton` object. The return value for `SetBitmapData()` is normally `ZAF_ERROR_NONE`.

The following code shows the proper use of these functions:

```
// Get the data.
const ZafBitmapData *data = button->BitmapData();
...
// Add the bitmap data.
extern ZafBitmapData *bitmapData;
button->SetBitmapData(bitmapData);
```

```
ZafButtonType ButtonType(void) const;
virtual ZafButtonType SetButtonType(ZafButtonType
    buttonType);
```

`ButtonType()` specifies the type of button a `ZafButton` object is. The default value of this attribute is `ZAF_NATIVE_BUTTON`, but the user may call `SetButtonType()` to change it. Here are the possible button types:

<code>ButtonType()</code>	Description
<code>ZAF_NATIVE_BUTTON</code>	Creates a native push button object (may look different from environment to environment)
<code>ZAF_RADIO_BUTTON</code>	Creates a radio button object
<code>ZAF_CHECK_BOX</code>	Creates a check box object
<code>ZAF_3D_BUTTON</code>	Creates a 3-d push button object
<code>ZAF_FLAT_BUTTON</code>	Creates a flat push button object
<code>ZAF_TOOLBAR_BUTTON</code>	Creates a push button object that displays specially to look appropriate when placed on a toolbar object

```
virtual void ClearImage(void);
```

`ClearImage()` deletes the bitmap data portion of a `ZafButton` object, if the bitmap data is `Destroyable()`. `ClearImage()` is an advanced routine, and should normally not be called by the programmer. This routine is called internally by the Zinc libraries.

```
virtual void ClearText(void);
```

ClearText() deletes the string data portion of a ZafButton object, if the string data is Destroyable(). ClearText() is an advanced routine, and should normally not be called by the programmer. This routine is called internally by the Zinc libraries.

```
virtual void ConvertCoordinates(ZafCoordinateType  
    coordinateType);
```

ConvertCoordinates() overloads ZafWindowObject::ConvertCoordinates(), and provides specific functionality for the ZafButton class before calling the base class ZafWindowObject::ConvertCoordinates(). ConvertCoordinates() is an advanced routine, and should normally not be called by the programmer. See ZafWindowObject::ConvertCoordinates() for more information.

```
virtual ZafCoordinateType  
    SetCoordinateType(ZafCoordinateType coordinateType);
```

SetCoordinateType() overloads ZafWindowObject::SetCoordinateType(), and provides specific functionality for the ZafButton class after calling the base class ZafWindowObject::SetCoordinateType(). SetCoordinateType() is an advanced routine, and should normally not be called by the programmer. See ZafWindowObject::SetCoordinateType() for more information.

```
bool Depressed(void) const;  
virtual bool SetDepressed(bool depressed);
```

A button in the process of being selected appears depressed until the selection operation is completed (for example, by releasing the mouse button). Depressed() and SetDepressed() are advanced routines, and should normally not be called by the programmer. These routines are called internally by the Zinc libraries.

```
int Depth(void) const;  
virtual int SetDepth(int depth);
```

The depth of a button is the number of pixels the button appears to be above the plane of its parent. Depth() and SetDepth() are advanced routines, and should normally not be called by the programmer. These routines are called internally by the Zinc libraries.

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function handles all events that are sent to the ZafButton object, either by processing the events itself, or by passing the event down for base class processing. Refer to ZafWindowObject::Event() for complete details. Following are events that are handled by ZafButton in addition to those handled by its base classes:

Event type	Description
S_HOT_KEY	Causes the object to perform its action, if event.key.value is the object's hot key
S_REDISPLAY_DEFAULT	Causes the object to display or erase its default border, as appropriate

```
ZafIChar HotKeyChar(void) const;
int HotKeyIndex(void) const;
virtual ZafIChar SetHotKey(ZafIChar hotKeyChar, int index
    = -1);
```

An object's hot key character is the character that when typed with a modifier key (such as <ALT> or <Command>) causes the object to be selected. The object's action (such as an event being posted to the event manager's queue if the button is SendMessageWhenSelected()) is performed as a result of the selection.

The hot key index is the zero-based index into the object's text that specifies the character to be visually displayed as the hot key character, usually with an underline.

It is important to note that the hot key character does not cause any display modification, and the hot key index does not cause any action to be performed when that character is typed with the modifier key. The default value of HotKeyChar() is 0, indicating that there is no hot key character associated with the object, and the default value of HotKeyIndex() is -1, indicating that no character is to be displayed as the hot key character on the object. The user may call SetHotKey() to change the HotKeyChar() and the HotKeyIndex() attributes. The *hotKeyChar* parameter specifies the hot key character, and the *index* parameter specifies the hot key index.

The following code shows how to create a button with a hot key:

```
// Create a button with a hot key.
ZafButton *button = new ZafButton(1, 4, 12, 1,
    ZAF_NULLP(ZafBitmapData), "Quit");
button->SetHotKey('Q', 0);
```

```
ZafHzJustify HzJustify(void) const;  
ZafVtJustify VtJustify(void) const;  
virtual ZafHzJustify SetHzJustify(ZafHzJustify  
    hzJustify);  
virtual ZafVtJustify SetVtJustify(ZafVtJustify  
    vtJustify);
```

HzJustify() and VtJustify() control the button's horizontal and vertical justification, respectively. HzJustify() is ZAF_HZ_CENTER by default, and VtJustify() is ZAF_VT_CENTER by default. The user may call SetHzJustify() or SetVtJustify() to change them.

If there are both textual and bitmap pieces associated with the button, both are justified as a unit within the available space on the button. With combinations of horizontal and vertical justification other than ZAF_HZ_CENTER and ZAF_VT_CENTER together, the bitmap is justified first, and then the text. When the button is ZAF_HZ_CENTER and ZAF_VT_CENTER together, the bitmap is placed on top of the text, and both as a unit are centered horizontally and vertically on the button. Here is a picture to help in visualizing justification effects on a button object.

(*** picture here ***)

```
virtual ZafError OSUpdatePalettes(void);
```

OSUpdatePalettes() causes the object to adjust its operating system palettes, if needed. For more information on palettes and their operations, see ZafPaletteData. OSUpdatePalettes() is an advanced routine, and should normally not be called by the programmer. This routine is called internally by the Zinc libraries.

```
virtual ZafRegionStruct Region(void) const;  
virtual void SetRegion(const ZafRegionStruct &region);
```

Region() and SetRegion() overload the ZafWindowObject methods, and provide specific functionality for the ZafButton class. A button object that is either AutoSize() or AllowDefault() may appear on the screen larger than its corresponding object region. For that reason, these overloaded functions are necessary to allow for this discrepancy between the object's region and the operating environment's representation of the object. See ZafWindowObject for more detailed descriptions of these functions.

```
virtual bool SetSelected(bool selected);
```

This overloaded function adds to the functionality of `ZafWindowObject::SetSelected()` by calling platform-specific functions to cause the button object to display appropriately according to its selected state. For example, a native check box on Microsoft Windows must receive a `BM_SETCHECK` message to toggle its state. This function is also one place where the button's `Depressed()` attribute is maintained internally during the operation of selecting a button.

```
bool SelectOnDoubleClick(void) const;
virtual bool SetSelectOnDoubleClick(bool
    selectOnDoubleClick);
```

If `SelectOnDoubleClick()` is true, the button performs its action on a double-click, as opposed to a single click. The maximum time between the first up-click and the second down-click is determined by the operating environment. For MS-DOS, `ZafWindowObject::doubleClickRate` is used. See `ZafWindowObject::doubleClickRate` for more information on the double-click rate. An example of an object with this behavior is an icon that causes an application to be launched by the operating environment. A single click simply causes the icon to receive focus, and a double-click causes the operating environment to launch the application associated with the icon. The default value of this attribute is false, but the user may call `SetSelectOnDoubleClick()` to change it.

```
bool SelectOnDownClick(void) const;
virtual bool SetSelectOnDownClick(bool
    selectOnDownClick);
```

If `SelectOnDownClick()` is true, the button performs its action on a down-click, as opposed to an up-click. An example of an object with this behavior is a combo-box button. A down-click on the combo-box button causes the list of choices to appear, allowing the user to position the mouse over an item and up-clicking to choose it. Normal buttons wait until the up-click to perform their action. The default value of this attribute is false, but the user may call `SetSelectOnDownClick()` to change it.

```
ZafEventType SendMessage(const ZafEventStruct &event,
    ZafEventType ccode);
bool SendMessageWhenSelected(void) const;
virtual bool SetSendMessageWhenSelected(bool
    sendMessageWhenSelected);
```

One way to tie a button to an action is through the `SendMessageWhenSelected()` attribute. The default value of this attribute is false, but the user may call `SetSendMessageWhenSelected()` to change it. The programmer may define a user event type, set the value of the button to the user event type through `SetValue()`, and handle the user event type in a derived object's `Event()` method. For example:

```
// Define the user event type.
const ZafEventType USER_ACTION_EVENT = 20001;
...
// Create a button that posts an event on the event manager's
// queue.
ZafButton *button = new ZafButton(1, 4, 12, 1,
    ZAF_NULLP(ZafBitmapData), "Action");
button->SetSendMessageWhenSelected(true);
button->SetValue(USER_ACTION_EVENT);
```

Alternatively, the programmer may wish to use a pre-defined ZAF event type. For example, to create a button that will cause the application to close down, use the following code snippet.

```
// Create a button that causes the application to close down.
ZafButton *button = new ZafButton(1, 4, 12, 1,
    ZAF_NULLP(ZafBitmapData), "Quit");
button->SetSendMessageWhenSelected(true);
button->SetValue(S_EXIT);
```

If `SendMessageWhenSelected()` is true, the `SendMessage()` method is automatically called when the button is selected. `SendMessage()` simply packages an event of the same type as the button's `Value()` attribute with `event.windowObject` pointing to the button, and posts the event on the event manager's queue. The programmer should never need to call `SendMessage()`, since it is called internally by the `ZafButton` class.

The return value for `SendMessage()` is an error code indicating any error that may have occurred. The return value is usually `ZAF_ERROR_NONE`, meaning that no error occurred.

```
ZafStringData *StringData(void) const;
virtual ZafError SetStringData(ZafStringData
    *stringData);
```

The `StringData()` object is the piece of the `ZafButton` object where the actual string data is stored. The `StringData()` may be shared among several `ZafButton`

objects, or it may belong to a single ZafButton object. If shared among several ZafButton objects, all the associated ZafButton objects will be updated when the StringData() piece changes. SetStringData() may be used to associate a StringData() object with a ZafButton object. For more information on data sharing in ZAF, see the chapter on ZafDataManager.

The return value for StringData() is a pointer to the StringData() object associated with the ZafButton object. The return value for SetStringData() is normally ZAF_ERROR_NONE.

```
virtual const ZafIChar *Text(void);
virtual ZafError SetText(const ZafIChar *text);
```

The textual data of a ZafButton (contained in the StringData() object) may be returned or set with Text() and SetText(). These functions provide simple accessibility to the StringData() of a ZafButton, and may be used if the programmer does not wish to interact directly with the data portion of the object.

The return value for Text() is a pointer to the textual information in the data object of a ZafButton. The return value for SetText() is normally ZAF_ERROR_NONE.

```
ZafEventType Value(void) const;
virtual ZafEventType SetValue(ZafEventType value);
```

Value() stores the event type that is posted on the event manager's queue for SendMessageWhenSelected() buttons. See SendMessageWhenSelected() for more information. Value() is 0 by default, but the user may call SetValue() to change it to any event type desired, including user-defined event types.

```
virtual bool SetVisible(bool visible);
```

SetVisible() overloads ZafWindowObject::SetVisible(), and provides platform-specific functionality (internal interface to the OS) for the ZafButton class before calling the base class ZafWindowObject::SetVisible(). See ZafWindowObject::SetVisible() for more information.

```
ZafVtJustify VtJustify(void) const;
virtual ZafVtJustify SetVtJustify(ZafVtJustify
    vtJustify);
```

See HzJustify().

ZafData

Clear	Destroyable	Duplicate
Error	GetObject	

ZafData is the base class for all low-level data types such as dates, times, strings, bitmaps, etc. This class is derived from both the ZafElement and ZafNotification base classes, giving it the inherited capabilities associated with list elements and data notification objects. These inherited features include string and number identifications, notification to window objects when a particular data value changes, and notification when the contents of an associated user interface object is modified.

ZafData is recognized as an abstract class since it has some virtual functions. In addition, it has a protected constructor, which means you cannot directly instantiate a ZafData class object. Since you cannot directly instantiate a ZafData class, you must determine and instantiate a publicly available derived class. There are three types of data objects that are supported by Zinc:

- data objects that allow formatted text
- data objects that have image information
- data objects that maintain internationalization information

These ZAF classes are listed below, presented under their data category.

Formatted	Images	Miscellaneous
ZafBignumData	ZafBitmapData	ZafCharMapData
ZafDateData	ZafIconData	ZafLanguageData
ZafIntegerData	ZafMouseData	ZafLocaleData
ZafStringData	ZafScrollData	
ZafRealData		
ZafTimeData		
ZafUTimeData		

Declaration	<code>#include <z_data.hpp></code>
Inheritance	<code>ZafData : (ZafElement, ZafNotification)</code>
Constructors	The ZafData class constructor is protected. Thus, it can only be instantiated from a derived class's constructor such as ZafDateData, ZafStringData, ZafIconData, or ZafLanguageData.

The primary purpose of this constructor is to initialize the ZafElement and ZafNotification portions of the class and to clear the error value (ZAF_ERROR_NONE) associated with the data object. The default values set by ZafData are listed below, as well as values overridden from those set by base class constructors.

Member Initializations

ZafData

Destroyable()	true
Error()	ZAF_ERROR_NONE

ZafElement

ClassID()	ID_ZAF_DATA
ClassName()	"ZafData"

ZafData(void);

All other aspects of this class constructor are deferred to the appropriate derived class definition. Please refer to the specific data object you are using for complete information about the object's construction.

ZafData(const ZafIChar *name, ZafDataPersistence
&persist);

This constructor is used for persistence. The parameters and values of this constructor are deferred to the ZafWindow section of this manual, since most persistence is done at the ZafWindow level.

Destructor

virtual ~**ZafData**(void);

This virtual destructor is used to free the memory associated with an instantiated ZafData object. The ZafData portion of the destructor performs no internal operations because no memory is allocated for ZafData members. It simply provides a chain from the derived object's destructor up to the ZafElement and ZafNotification class destructors.

A ZafData pointer may be deleted even though the class definition is abstract. This is done by allocating a derived object and by setting the returned object to a ZafData pointer.

The following code shows the correct use of the ZafData destructor under these conditions.

```
// Create a string data object.
```

```
ZafData *string = new ZafStringData("string", 100);
...
// Free the string.
delete string;
```

The pointer assignment, shown above, is permitted because ZafData is a base class to ZafStringData. When the string object’s destructor is called, the actual contents of the ZafStringData instance are freed because the base class destructor is declared virtual.

For complete information on the type of memory that is freed as a result of a call to the destructor, see the reference chapter on the particular object you created.

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
virtual void Clear(void) = 0;
```

This is a pure virtual function that abstractly defines “clearing” functionality for a derived object. The Clear() function operates differently on various derived objects. For example:

Derived class	Description of Clear()
ZafBitmapData	Destroys the original bitmap array associated with the object and also any environment specific bitmap handles that were created during the application’s run-time operation.
ZafIntegerData	Sets the integer value of the object to be 0.
ZafLocaleData	Resets the locale information based on a set of information known as the canonical locale.
ZafStringData	Sets the string value of the object to be a blank string, but does not destroy the string buffer.

Although this is a pure virtual function, you can clear a derived ZafData object by constructing the object, then by calling its Clear() function.

The following code shows how this is done.

```
// Create an integer.
ZafData *data = new ZafIntegerData(100);
```

```
...
// Clear the data object's contents.
data->Clear();
```

If data notification is active, and the integer data is associated with a window object, then a call to `Clear()` will not only clear the internal value of the data, but will also make a request to the associated window object to re-display its screen information.

```
bool Destroyable(void) const;
bool SetDestroyable(bool destroyable);
```

If `Destroyable()` is true, then the data object is considered non-static, and is maintained by the corresponding window object. In other words, when the corresponding window object is destroyed, then the data object is also destroyed, usually by the window object's destructor. On the other hand, if `Destroyable()` is false, ZAF will not destroy the data object, and the programmer assumes responsibility for the data object. The default value of this attribute is true, but it may be changed by calling `SetDestroyable()`.

```
virtual ZafData *Duplicate(void) = 0;
```

`Duplicate()` abstractly defines functionality for duplicating a derived object. A derived class must provide a `Duplicate()` function, since it is declared pure virtual. Generally, `Duplicate()` simply calls the copy constructor for the derived class. `Duplicate()` may be called correctly with a `ZafData` pointer, since it is declared virtual. For example, the following code correctly creates two string data objects:

```
// Create an integer.
ZafData *data = new ZafIntegerData(100);
...
// Create a duplicate of the integer data object.
ZafData *dataCopy = data->Duplicate();
```

```
ZafError Error(void) const;
ZafError SetError(ZafError error);
```

`Error()` stores the last error that occurred with the object. The default value of `Error()` is `ZAF_ERROR_NONE`, but it may be set internally by the library whenever an error occurs, and `SetError()` may be called to change it. Note that the programmer is responsible for setting this attribute back to `ZAF_ERROR_NONE` when appropriate. The types of errors that can be set

are defined in the header file `z_env.hpp`. Generally, however, only the following error values will be used by a `ZafData` object:

Error()	Description
<code>ZAF_ERROR_NONE</code>	No error exists.
<code>ZAF_ERROR_INVALID</code>	The contents of the data object are invalid, meaning the data object's value can be shown on the screen and used in calculations, but that the value is incorrect in the context of the application. For instance, the value 45 is a legitimate integer, but is invalid when used to describe the number of days in the month of February.
<code>ZAF_ERROR_OUT_OF_RANGE</code>	An error occurred while trying to convert data from one type to another or where the argument was too big or too small for the return value. For example, a value of 1000 does not fit into a “%c” <code>sprintf()</code> style argument.
<code>ZAF_ERROR_INVALID_TARGET</code>	The target data could not accept the source value being presented. Such an error would occur while trying to set an integer value from a large floating point number.
<code>ZAF_ERROR_INVALID_SOURCE</code>	The source data is either of an incorrect type, or the data cannot be converted by the target object. This error is similar to the invalid target error, except that the error occurred in the information being passed by the source, rather than by an error in the target.

In addition to the error types described above, error values greater than or equal to 10,000 are reserved for use on user-defined objects.

The following code fragments show how to define and use an error value with a derived data object.

```
// Define the class and constant.  
const ZafError CARTESIAN_ERROR = 10000;  
class Cartesian : public ZafData  
...
```

```
// Create a new coordinate.
Cartesian coordinate(120, 100);
...
// Check for coordinate error.
if (coordinate > point1 || coordinate < point2)
    coordinate.SetError(CARTESIAN_ERROR);
...
// Check the error status.
if (coordinate.Error())
    break;
```

```
virtual ZafData *GetObject(ZafNumberID numberID);
virtual ZafData *GetObject(const ZafIChar *stringID);
```

At the data level, the `GetObject()` functions simply check the parameter against the data object's identification. The *numberID* parameter is checked against `NumberID()`, and the *stringID* parameter is checked against `StringID()`. If they match, `GetObject()` returns a pointer to the data object, or null otherwise.

ZafDate

Event	DateData
-------	----------

ZafDate is a single-line date object that allows user input through the keyboard. ZafDate is fully internationalized to display and input using any format.

All ZafDate objects refer to data contained in a ZafDateData object (refer to this class for additional essential information).

Declaration `#include <z_date1.hpp>`

Inheritance `ZafDate : ZafString : ZafWindowObject : ZafElement`

Constructors All ZafDate constructors initialize the member variables associated with an instantiated ZafDate object. The default values set by the ZafDate and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafDate. “†”Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafDate

DateData()	null
------------	------

ZafString

LowerCase()	false [†]
Password()	false [†]
StringData()	null [†]
UpperCase()	false [†]
VariableName()	false [†]

ZafElement

ClassID()	ID_ZAF_DATE
ClassName()	"ZafDate"

ZafDate(int left, int top, int width, int year, int month, int day);

This constructor is useful in straight-code situations, particularly if you wish the ZafDate object to create, maintain and destroy its own ZafDateData object automatically. *left*, *top* and *width* specify the position and size of the object on

its parent. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired.

year, *month*, and *day* specify the date values you wish to initially appear in the new ZafDate object.

```
ZafDate(int left, int top, int width, ZafDateData
        *dateData = ZAF_NULLP(ZafDateData));
```

This constructor is useful in straight-code situations where a ZafDateData object has already been created. This constructor could be used when manually maintaining a *dateData* object, rather than having the ZafDate class create and maintain the data object automatically. For more information on using ZafDateData objects, see the chapter on ZafDateData. See the previous constructor for a description of *left*, *top* and *width* parameters.

```
ZafDate(const ZafDate &copy);
```

The copy constructor calls the overloaded Duplicate() to create a new ZafDate object and initialize its data from *copy*. If the original data objects are StaticData() then the new ZafDate object simply points to the original data, otherwise StaticData() copies are made.

```
ZafDate(const ZafIChar *name, ZafObjectPersistence
        &persist);
```

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

Sample ZafDate creation techniques follow:

```
// Create a sample window with date objects.
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);

// Create date objects and pass in the values directly.
window1->Add(new ZafDate(0, 1, 25, 1984, 1, 22));
window1->Add(new ZafDate(0, 2, 25, 2000, 1, 1));
...
// Create a sample window with date objects.
ZafWindow *window2 = new ZafWindow(10, 10, 40, 10);

// Create date data objects.
ZafDateData *dateData1 = new ZafDateData(1984, 1, 22);
ZafDateData *dateData2 = new ZafDateData(2000, 1, 1);

// Create dates that use the data previously created.
```

```

window2->Add(new ZafDate(0, 1, 25, dateData1));
window2->Add(new ZafDate(0, 2, 25, dateData2));

```

Destructor

```
virtual ~ZafDate(void);
```

The destructor is used to free the memory associated with a ZafDate object, including all the data objects that are Destroyable(). It chains to the ZafString, ZafWindowObject, and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafDate object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~~ZafWindow().

Members

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function receives all events that get sent to the ZafDate object and either handles them or passes them to ZafString, its immediate base class. See ZafWindowObject for more information.

ZafDate specifically handles the following events:

Event	Description
S_COPY_DATA	causes the object to copy event.windowObject's DateData() if event.windowObject is a ZafDate object
S_SET_DATA	causes the object to create a new DateData() object, then copy into it event.windowObject's DateData() if event.windowObject is non-null and is a ZafDate object

```
ZafDateData *DateData(void) const;
```

```
virtual ZafError SetDateData(ZafDateData *dateData);
```

*DateData() contains the actual information used by ZafDate. The DateData() object may be used by one or more ZafDate objects, or other objects. If shared, all associated ZafDate objects will be notified when the DateData() changes. For more information on data sharing in ZAF, see ZafDataManager.

DateData() returns a pointer to the DateData() object associated with the ZafDate object. The return value for SetDateData() is normally ZAF_ERROR_NONE. See the constructor code snippet for an example using ZafDateData objects with ZafDate.

ZafDevice

display	DeviceState	DeviceType
Event	Installed	Next
Previous	Poll	

ZafDevice is an abstract class that defines the basic functionality necessary to support input devices such as a mouse, a keyboard, or a cursor. All ZAF input device classes derive from this class.

ZAF device classes share certain characteristics in common. For example, all device objects are added to the ZafEventManager object. Device object input is packaged into events and placed on the event manager’s event queue. Frequently, device objects handle events dispatched to them such as activation and deactivation.

In this section, examples are used that apply to ZafDevice, but also may apply to the creation and use of all device objects. Also refer to ZafEventManager for information regarding the general operation of device objects.

Declaration

#include <z_device.hpp>

Inheritance

ZafDevice : ZafElement

Constructor

The ZafDevice constructor initializes the member variables associated with an instantiated ZafDevice object. The default values set by ZafDevice and its base class constructor follow, if they override values set by the base class constructor.

Member Initializations

ZafDevice

DeviceState()	user-supplied parameter
DeviceType()	user-supplied parameter
display	null
eventManager	null
Installed()	false

ZafElement

ClassID()	ID_ZAF_DEVICE
ClassName()	"ZafDevice"

ZafDevice(ZafDeviceType type, ZafDeviceState state);

This constructor is used to instantiate a ZafDevice object to be added to a ZafEventManager object. *type* specifies the type of the device, such as E_MOUSE. *state* specifies the initial state of the device, such as DM_VIEW. This constructor should not normally be called by the programmer, since the constructors of classes derived from ZafDevice chain to this constructor.

Sample ZafDevice object creation techniques follow:

```
// Instantiate the input devices.
ZafEventManager *eventManager = new ZafEventManager;
eventManager->Add(new ZafKeyboard);
eventManager->Add(new ZafMouse);
eventManager->Add(new ZafCursor);
```

Destructor

```
virtual ~ZafDevice(void);
```

The destructor is used to free the memory associated with a ZafDevice object. It chains to the ZafElement destructor.

Generally, the programmer will not directly destroy a ZafDevice object, since it is automatically destroyed when the event manager is destroyed. For more information on device object deletion, see ZafEventManager::~ZafEventManager().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
static ZafDisplay *display;
```

The display member of this class is a static pointer to the display object instantiated by the ZafApplication constructor. This member is useful since most device objects have direct interaction with the display.

```
ZafDeviceState DeviceState(void) const;
virtual ZafDeviceState SetDeviceState(ZafDeviceState
    deviceState);
```

DeviceState() returns a constant that indicates a device object's current state. For example, DeviceState() returns DM_VIEW for a mouse device that currently displays the default pointer image. SetDeviceState() may be called to change a device's state.

An example of device state manipulation follows:

```
// Indicate that a lengthy operation is in process.
mouse->SetDeviceState(DM_WAIT);
DoLotsOfWork();
mouse->SetDeviceState(DM_VIEW);
```

```
ZafDeviceType DeviceType(void) const;
```

`DeviceType()` returns the constant that identifies a device object's type. For example, `E_MOUSE` is returned for a mouse device, `E_KEY` is returned for a keyboard device, and `E_CURSOR` is returned for a cursor device.

```
virtual ZafEventType Event(const ZafEventStruct &event) =
    0;
```

This function is pure virtual. Any device class using `ZafDevice` as a base class must overload the `Event()` function to handle events passed to it.

Classes deriving from `ZafDevice` must handle the event `D_STATE`, which causes a device to return its state.

```
bool Installed(void) const;
```

This function returns true if the device object has been successfully installed in the event manager. If an error occurred during device object initialization, this function will return false.

```
ZafDevice *Next(void) const;
```

```
ZafDevice *Previous(void) const;
```

These overloaded functions add a type-safe cast of "`ZafDevice *`" to the object while accessing the next or previous sibling in the event manager's list of devices. This allows initialization and manipulation of a list of associated device objects with the proper base type declaration. For example:

```
// Find the mouse device.
for (ZafDevice *device = eventManager->First(); device; device =
    device->Next())
    if (device->DeviceType() == E_MOUSE)
        break;
```

```
virtual void Poll(void) = 0;
```

This function is pure virtual. Any device class using ZafDevice as a base class must overload the Poll() function to post events on the event manager queue. For more information on device object polling, see ZafEventManager::Get().

ZafDisplay

AddColor	AddFont	Background
BeginDraw	Bitmap	ClipRegion
ColorTable	columns	ConvertToOSBitmap
ConvertToOSIcon	ConvertToOSMouse	ConvertToZafBitmap
ConvertToZafIcon	ConvertToZafMouse	DestroyColor
DestroyFont	DestroyOSBitmap	DestroyOSIcon
DestroyOSMouse	DestroyZafBitmap	DestroyZafIcon
DestroyZafMouse	DisplayContext	DisplayMode
DrawContext	Ellipse	EndDraw
FillPattern	Font	fontTable
Foreground	Icon	InitializeOSBitmap
InitializeOSIcon	InitializeOSMouse	Line
lines	LineStyle	lineTable
Mode	modeTable	MonoForeground
monoTable	Mouse	Palette
patternTable	Pixel	Polygon
preSpace	postSpace	Rectangle
RectangleXORDiff	RegionCopy	ResetOSBitmap
ResetOSIcon	ResetOSMouse	RestoreDisplayContext
RestoreDrawContext	Text	TextSize

ZafDisplay defines the basic functionality necessary to interface with the screen. ZafDisplay provides many useful graphic display primitives for use in a ZAF application, such as Line(), Rectangle() and Text(). It also provides methods for interfacing with images, such as bitmaps and icons.

ZafDisplay is important to displayable ZAF classes because it provides the interface necessary to interact with the screen. A ZafDisplay object is instantiated by the ZafApplication class, and should not be instantiated by the programmer. When needed, the member ZafWindowObject::display should be used to access the ZafDisplay object instantiated automatically by the ZafApplication class.

Declaration	#include <z_dsp.hpp>
Inheritance	ZafDisplay : ZafCoordinateStruct
Constructor	The ZafDisplay constructor initializes the member variables associated with an instantiated ZafDisplay object. The default values set by the ZafDisplay con-

structor and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafDisplay

<code>columns</code>	<code>screen-dependent</code>
<code>lines</code>	<code>screen-dependent</code>
<code>preSpace</code>	<code>environment-specific</code>
<code>postSpace</code>	<code>environment-specific</code>
<code>lineTable[]</code>	<code>environment-specific</code>
<code>patternTable[]</code>	<code>environment-specific</code>
<code>colorTable[]</code>	<code>environment-specific</code>
<code>monoTable[]</code>	<code>environment-specific</code>
<code>fontTable[]</code>	<code>environment-specific</code>
<code>modeTable[]</code>	<code>environment-specific</code>

ZafCoordinateStruct

<code>cellWidth</code>	<code>environment-specific</code>
<code>cellHeight</code>	<code>environment-specific</code>
<code>miniNumeratorX</code>	<code>1</code>
<code>miniDenominatorX</code>	<code>10</code>
<code>miniNumeratorY</code>	<code>1</code>
<code>miniDenominatorY</code>	<code>10</code>
<code>pixelsPerInchX</code>	<code>environment-specific</code>
<code>pixelsPerInchY</code>	<code>environment-specific</code>

```
ZafDisplay(int argc, char **argv);
```

The constructor is useful in straight-code situations. The *argc* and *argv* parameters are simply the *argc* and *argv* parameters passed into the application, and are used to find the application's name in environments that require the application name to be registered with the system. This constructor will not normally be called by the programmer, since it is automatically called by `ZafApplication`.

Destructor

```
virtual ~ZafDisplay(void);
```

The destructor is used to free the memory associated with a `ZafDisplay` object. Generally, the programmer will not directly destroy a `ZafDisplay` object since it is automatically destroyed when the `ZafApplication` object is destroyed.

Members

```
ZafLogicalColor AddColor(ZafLogicalColor index, ZafUInt8  
    red, ZafUInt8 green, ZafUInt8 blue);  
ZafError DestroyColor(ZafLogicalColor color);
```

ZafDisplay allows the use of up to 127 colors. In order to save execution time and memory, however, ZafDisplay automatically registers just the most common 16 colors with the environment.

The 16 colors registered by ZafDisplay, and usable in any ZAF program are:

- ZAF_CLR_BLACK
- ZAF_CLR_BLUE
- ZAF_CLR_GREEN
- ZAF_CLR_CYAN
- ZAF_CLR_RED
- ZAF_CLR_MAGENTA
- ZAF_CLR_BROWN
- ZAF_CLR_LIGHTGRAY
- ZAF_CLR_DARKGRAY
- ZAF_CLR_LIGHTBLUE
- ZAF_CLR_LIGHTGREEN
- ZAF_CLR_LIGHTCYAN
- ZAF_CLR_LIGHTRED
- ZAF_CLR_LIGHTMAGENTA
- ZAF_CLR_YELLOW
- ZAF_CLR_WHITE

Run-time support for any RGB color may be added via AddColor(). The *index* parameter specifies the zero-based index (or logical color) into the ZAF internal color table where the color is to be inserted. The 16 colors preloaded by ZAF occupy indices 0-15. If ZAF_CLR_NULL is passed into *index*, the new color is loaded into the next unused ZafLogicalColor index. The *red*, *green* and *blue* parameters specify the 8-bit RGB values of the new color. DestroyColor() causes the associated internal color table entry to be cleared.

The ZafLogicalColor to be used in referencing the new color is returned. If ZAF_CLR_NULL is passed in, but there are no unused indices, then ZAF_CLR_NULL is returned. Also, if index is out of range, or some other error occurs, ZAF_CLR_NULL is returned.

Below is an example using these functions:

```
// Add Antique White to the color table.
```

```

ZafLogicalColor antiqueWhite = display->AddColor(ZAF_CLR_NULL,
    250, 235, 215);
SetForeground(antiqueWhite);
DrawSomethingWithColor(antiqueWhite);
...
// Remove Antique White from the color table.
display->DestroyColor(antiqueWhite);

```

```

ZafLogicalFont AddFont(ZafLogicalFont index, char
    *fontFamily, int pointSize, ZafFontWeight weight =
    ZAF_FNT_WEIGHT_NORMAL, ZafFontSlant slant =
    ZAF_FNT_SLANT_NORMAL);
ZafError DestroyFont(ZafLogicalFont font);

```

ZafDisplay allows the use of up to 10 fonts. However, in order to save execution time and memory, ZafDisplay automatically registers just the most common 5 fonts with the environment.

The 5 fonts registered by ZafDisplay, and usable in any ZAF program are as follows:

Logical font	Description
ZAF_FNT_SMALL	Small font, commonly used in icons
ZAF_FNT_DIALOG	Dialog font, commonly used in dialog windows
ZAF_FNT_APPLICATION	Application font, commonly used in multi-line text fields
ZAF_FNT_SYSTEM	System font, commonly used in buttons
ZAF_FNT_FIXED	Fixed-width font, commonly used in code snippets

Run-time support for any available font may be added via `AddFont()`. The *index* parameter specifies the zero-based index (or logical font) into the ZAF internal font table where the color is to be inserted. The 5 fonts preloaded by ZAF occupy indices 0-4. If `ZAF_FNT_NULL` is passed into *index*, the new font is loaded into the next available `ZafLogicalFont` index. The *fontFamily* parameter specifies the name of the font family, and the *pointSize* parameter specifies the desired point size of the font to be supported.

The *weight* parameter specifies the desired weight of the font, and may be any of the following:

- `ZAF_FNT_WEIGHT_NORMAL`
- `ZAF_FNT_WEIGHT_BOLD`

The *slant* parameter specifies the desired slant of the font, and may be any of the following:

- ZAF_FNT_SLANT_NORMAL
- ZAF_FNT_SLANT_ITALIC

The ZafLogicalFont to be used in referencing the new font is returned. If ZAF_FNT_NULL is passed in, but there are no unused indices, then ZAF_FNT_NULL is returned. Also, if index is out of range, or some other error occurs, ZAF_FNT_NULL is returned.

It is important to note that AddFont() is provided for limited use and may need to be called differently on different environments, according to the font families available on different environments. DestroyFont() causes a font to be unregistered with the environment, and the associated internal font table entry to be cleared.

The following shows how to use these functions:

```
// Add 10 point Helvetica to the font table.
ZafLogicalFont helvetica10Font = display->AddFont(ZAF_FNT_NULL,
    "helvetica", 10);
SetFont(helvetica10Font);
DrawSomethingWithFont(helvetica10Font);
...
// Remove Helvetica from the font table.
display->DestroyFont(helvetica10Font);
```

```
ZafLogicalColor Background(void) const;
ZafLogicalColor SetBackground(ZafLogicalColor color);
```

Background() specifies the color used by any drawing operation requiring a background color. When drawing shapes or textual information, the background color is used as the fill color. See Rectangle() and Text() for more information on the fill color. The default value for Background() is ZAF_CLR_WHITE, but it may be changed by calling SetBackground().

```
ZafError BeginDraw(OSDisplayContext displayContext,
    OSDrawContext drawContext, const ZafRegionStruct
    &draw, const ZafRegionStruct &clip);
```

All actual drawing operations must occur between calls to BeginDraw() and EndDraw(). BeginDraw() sets up the environment's display to accept drawing operations, and EndDraw() finalizes the drawing operation. The *displayContext* parameter specifies the display context to use for the drawing

operation (see `DisplayContext()` for an explanation of display contexts). The *drawContext* parameter specifies the draw context to use (see `DrawContext()` for an explanation of draw contexts). The *draw* parameter specifies the region where drawing is to occur, and the *clip* parameter specifies the region (usually inside draw) where clipping will occur. These functions are advanced, and should not normally be called by the programmer. They are called internally in the Zinc libraries. `ZafWindowObject::BeginDraw()` and `ZafWindowObject::EndDraw()` should be used to properly encapsulate drawing operations. See `ZafWindowObject::BeginDraw()` for a description of performing draw operations for an object.

```
virtual ZafError Bitmap(int column, int line,
    ZafBitmapStruct &bitmap);
```

`Bitmap()` is used to display the bitmap specified in the `bitmap` parameter, with its left top corner specified by the *column* and *line* parameters. This function is optimized for bitmaps that have already been converted to the environment's native format, using `ConvertToOSBitmap()`. If the bitmap has not yet been converted, `Bitmap()` calls `ConvertToOSBitmap()` automatically. See `ZafBitmapStruct` for more information on bitmaps. Normally, `Bitmap()` returns `ZAF_ERROR_NONE`.

```
ZafRegionStruct ClipRegion(void) const;
ZafRegionStruct SetClipRegion(const ZafRegionStruct
    &region);
```

`ClipRegion()` specifies the region (usually inside the current drawing region) that drawing operations will be clipped to. `ClipRegion()` defaults to the entire drawing region, but it may be changed by calling `SetClipRegion()`. `ClipRegion()` returns the current clipping region, and `SetClipRegion()` returns the clipping region before it was changed.

For example, when drawing a 3-d button object whose text is longer than will fit inside the button's region, it is not desirable for the text to be drawn over the shadow portion of the button. Calling `SetClipRegion()` with the region inside the button's shadow will cause `Text()` to clip its drawing operation inside the shadow's region, as shown below:

```
// Begin the drawing operation.
ZafRegionStruct drawRegion = BeginDraw();
// Draw a one pixel border.
DrawBorder(drawRegion, ccode);
// Draw the 3D shadow.
DrawShadow(drawRegion, depth, ccode);
```

```
// Erase the background.
DrawBackground(drawRegion, ccode);
// Set the clipping region.
display->SetClipRegion(drawRegion);
// Set the text palette.
display->SetPalette(LogicalPalette(ZAF_PM_TEXT,
    PaletteState()));
// Draw the text.
display->Text(drawRegion, "Long button name", -1);
// End the drawing operation.
EndDraw();
```

```
static OSColor ZAF_FARDATA colorTable[ZAF_MAXCOLORS];
```

The static array `colorTable` is the internal ZAF color table that stores environment-specific information for each ZAF logical color. It is used internally by the Zinc libraries, and should normally not be modified by the programmer. See `AddColor()` for a description of adding support for RGB colors.

```
int columns;
int lines;
```

The `columns` and `lines` members are initialized in the `ZafDisplay` constructor, and contain the width and height in pixels, respectively, of the main monitor or display that the application is running on.

```
virtual ZafError ConvertToOSBitmap(ZafBitmapStruct
    &bitmap);
virtual ZafError ConvertToOSIcon(ZafIconStruct &icon);
virtual ZafError ConvertToOSMouse(ZafMouseStruct &mouse);
```

`ConvertToOSBitmap()`, `ConvertToOSIcon()` and `ConvertToOSMouse()` convert ZAF platform-independent images to completely native versions of the images that may be passed to native API routines. The ZAF image is passed into the routine, and the converted image is stored within the same structure. Normally, `ZAF_ERROR_NONE` is returned, but one of the following may be returned in special cases:

Return value	Description
<code>ZAF_ERROR_INVALID_SOURCE</code>	Indicates that the ZAF source image is empty or invalid

Return value	Description
ZAF_ERROR_INVALID_TARGET	Indicates that the native image already exists and is StaticHandle(), and may not be re-converted

```
virtual ZafError ConvertToZafBitmap(ZafBitmapStruct
    &bitmap);
virtual ZafError ConvertToZafIcon(ZafIconStruct &icon);
virtual ZafError ConvertToZafMouse(ZafMouseStruct
    &mouse);
```

ConvertToZafBitmap(), ConvertToZafIcon() and ConvertToZafMouse() convert the native versions of images to ZAF platform-independent versions of the images that may be stored and retrieved independent of the particular environment. The ZAF image structure containing the native image is passed into the routine, and the converted image is stored within the same structure. Normally, ZAF_ERROR_NONE is returned, but one of the following may be returned in special cases:

Return value	Description
ZAF_ERROR_INVALID_SOURCE	Indicates that the native source image is empty or invalid
ZAF_ERROR_INVALID_TARGET	Indicates that the ZAF image already exists and is StaticArray(), and may not be re-converted

```
ZafError DestroyColor(ZafLogicalColor color);
```

See AddColor().

```
ZafError DestroyFont(ZafLogicalFont font);
```

See AddFont().

```
virtual ZafError DestroyOSBitmap(ZafBitmapStruct
    &bitmap);
virtual ZafError DestroyOSIcon(ZafIconStruct &icon);
virtual ZafError DestroyOSMouse(ZafMouseStruct &mouse);
```

DestroyOSBitmap(), DestroyOSIcon() and DestroyOSMouse() destroy the native versions of the images contained in the ZAF image structures passed into the functions. The ZAF image is untouched. If the native version of the image is StaticHandle(), it is not destroyed. Normally, ZAF_ERROR_NONE is returned.

```
virtual ZafError DestroyZafBitmap(ZafBitmapStruct  
    &bitmap);  
virtual ZafError DestroyZafIcon(ZafIconStruct &icon);  
virtual ZafError DestroyZafMouse(ZafMouseStruct &mouse);
```

DestroyZafBitmap(), DestroyZafIcon() and DestroyZafMouse() destroy the ZAF versions of the images contained in the ZAF image structures passed into the functions. The native image is untouched. If the ZAF version of the image is StaticArray(), it is not destroyed. Normally, ZAF_ERROR_NONE is returned.

```
OSDisplayContext DisplayContext(void) const;  
OSDisplayContext SetDisplayContext(OSDisplayContext  
    context);  
OSDisplayContext RestoreDisplayContext(void);
```

The display context is environment-specific, and usually specifies the device context used in display operations (for example, either a window's or printer's logical port). These functions are advanced, and should not normally be called by the programmer. They are called internally in the Zinc libraries to get, set and restore a display context. See ZafWindowObject::BeginDraw() for a description of performing draw operations for an object.

```
OSDrawContext DrawContext(void) const;  
OSDrawContext SetDrawContext(OSDrawContext context);  
OSDrawContext RestoreDrawContext(void);
```

The draw context is environment-specific, and usually specifies the logical drawing port used in display operations. These functions are advanced, and should not normally be called by the programmer. They are called internally in the Zinc libraries to get, set and restore a draw context. See ZafWindowObject::BeginDraw() for a description of performing draw operations for an object.

```
ZafDisplayMode DisplayMode(void) const;
```

DisplayMode() returns whether the run-time system supports color or black and white. On multiple-monitor systems such as is supported by the Macintosh, DisplayMode() may be misleading, since it does not indicate what mode each monitor connected to the system supports. In this case, though a system supports color, it may have only a black and white monitor connected to it; so DisplayMode() returns ZAF_DISPLAY_COLOR, even though there are no displays connected to the system that support color.

One of the following may be returned by DisplayMode():

Return value	Description
ZAF_DISPLAY_COLOR	Indicates that the run-time system supports color or grayscale
ZAF_DISPLAY_MONO	Indicates that the run-time system supports only black and white

virtual ZafError **Ellipse**(int left, int top, int right, int bottom, int startAngle, int endAngle, int width = 1, bool fill = false);

Ellipse() is used to display an elliptical shape specified by the parameters. The shape is contained by the *left*, *top*, *right* and *bottom* parameters. The *startAngle* and *endAngle* parameters are specified in degrees, beginning with the positive x-axis. If startAngle and endAngle are the same, a closed elliptical shape is drawn; otherwise a wedge is drawn beginning at startAngle and ending at endAngle. The *width* parameter specifies the width in pixels of the shape's outline, and the outline is drawn with the Foreground() color, using LineStyle(). The *fill* parameter specifies whether or not the inside of the shape should be filled with the Background() color, using FillPattern(). Normally, Ellipse() returns ZAF_ERROR_NONE.

The following shows how to use Ellipse():

```
void MyObject::Draw(void)
{
    ZafRegionStruct drawRegion = BeginDraw();
    // Draw a yellow ellipse with a blue border.
    SetBackground(ZAF_CLR_YELLOW);
    SetForeground(ZAF_CLR_BLUE);
    display->Ellipse(drawRegion.left, drawRegion.top,
        drawRegion.right, drawRegion.bottom, 0, 360, 1, true);
    EndDraw();
}
```



```
ZafError EndDraw(void);
```

See BeginDraw().

```
ZafLogicalFillPattern FillPattern(void) const;
ZafLogicalFillPattern
    SetFillPattern(ZafLogicalFillPattern pattern);
```

FillPattern() specifies the fill pattern used by any drawing operation that specifies filling. When drawing shapes or textual information, the fill pattern is used when filling the shape or text. The default value for FillPattern() is ZAF_PTN_SOLID_FILL, but it may be changed by calling SetFillPattern(). The fill patterns defined for use in ZAF are as follows:

Logical fill pattern	Description
ZAF_PTN_SOLID_FILL	Fills the entire area with the Background() color
ZAF_PTN_INTERLEAVE_FILL	Fills the area with an 50% interleave pattern of the Background() and Foreground() colors

```
ZafLogicalFont Font(void) const;
ZafLogicalFont SetFont(ZafLogicalFont font);
```

Font() specifies the logical font used by any textual drawing or sizing operation. When drawing or specifying textual information, the environment's font corresponding to the ZAF logical font is used. The ZafDisplay constructor initializes 5 fonts automatically, but support for additional fonts may be added. See AddFont() for more information on the 5 default fonts and how to add support for additional fonts. The default value for Font() is ZAF_FNT_DIALOG, but it may be changed by calling SetFont().

```
static OSFont ZAF_FARDATA fontTable[ZAF_MAXFONTS];
```

The static array fontTable is the internal ZAF font table that stores environment-specific information for each ZAF logical font. It is used internally by the Zinc libraries, and should normally not be modified by the programmer. See AddFont() for a description of adding support for available fonts.

```
ZafLogicalColor Foreground(void) const;
ZafLogicalColor SetForeground(ZafLogicalColor color);
```

Foreground() specifies the color used by any drawing operation requiring a foreground color. When drawing shapes or textual information, the foreground color is used as the outline or text color. See Rectangle() and Text() for more information on the outline or text color. The default value for Foreground() is ZAF_CLR_BLACK, but it may be changed by calling SetForeground().

```
virtual ZafError Icon(int column, int line, ZafIconStruct
    &icon);
```

Icon() is used to display the icon specified in the icon parameter, with its left top corner specified by the column and line parameters. This function is optimized for icons that have already been converted to the environment's native format, using ConvertToOSIcon(). If the icon has not yet been converted, Icon() calls ConvertToOSIcon() automatically. See ZafIconStruct for more information on icons. Normally, Icon() returns ZAF_ERROR_NONE.

```
static ZafError InitializeOSBitmap(ZafBitmapStruct
    &bitmap);
static ZafError InitializeOSIcon(ZafIconStruct &icon);
static ZafError InitializeOSMouse(ZafMouseStruct &mouse);
```

InitializeOSBitmap(), InitializeOSIcon() and InitializeOSMouse() initialize the native versions of the images contained in the ZAF image structures passed into the functions. The ZAF image is untouched. These functions are advanced, and should normally not be called by the programmer. They are used internally by the Zinc libraries to initialize image structures during construction. Normally, ZAF_ERROR_NONE is returned.

```
virtual ZafError Line(int column1, int line1, int column2,
    int line2, int width = 1);
```

Line() is used to display a line specified by the parameters. The line is drawn between the point (*column1*, *line1*) and (*column2*, *line2*). The *width* parameter specifies the line's width in pixels, and the line is drawn with the Foreground() color, using LineStyle(). Normally, Line() returns ZAF_ERROR_NONE.

```
int lines;
```

See columns.

```
ZafLogicalLineStyle LineStyle(void) const;  
ZafLogicalLineStyle SetLineStyle(ZafLogicalLineStyle  
    line);
```

`LineStyle()` specifies the line style used by any drawing operation. When drawing shapes, the line style is used when drawing the shape's outline. The line styles defined for use in ZAF are `ZAF_LINE_SOLID` and `ZAF_LINE_DOTTED`. The default value for `LineStyle()` is `ZAF_LINE_SOLID`, but it may be changed by calling `SetLineStyle()`.

```
static OSLineStyle ZAF_FARDATA lineTable[ZAF_MAXLINES];
```

The static array `lineTable` is the internal ZAF line style table that stores environment-specific information for each ZAF logical line style. It is used internally by the Zinc libraries, and should normally not be modified by the programmer.

```
ZafLogicalMode Mode(void) const;  
ZafLogicalMode SetMode(ZafLogicalMode mode);
```

`Mode()` specifies the drawing mode used by any drawing operation. The default value for `Mode()` is `ZAF_MODE_COPY`, but it may be changed by calling `SetMode()`. The drawing modes defined for use in ZAF are as follows:

Logical drawing mode	Description
<code>ZAF_MODE_COPY</code>	Draws normally, regardless of what was previously in the same position
<code>ZAF_MODE_XOR</code>	Performs a logical exclusive OR between the pixels being drawn, and the pixels that were already in the same position

```
static OSMode ZAF_FARDATA modeTable[ZAF_MAXMODES];
```

The static array `modeTable` is the internal ZAF drawing mode table that stores environment-specific information for each ZAF logical drawing mode. It is used internally by the Zinc libraries, and should normally not be modified by the programmer.

```
ZafLogicalColor MonoBackground(void) const;  
ZafLogicalColor SetMonoBackground(ZafLogicalColor color);
```

`MonoBackground()` specifies the color used by any drawing operation requiring a background color on a black and white display. When drawing shapes or textual information, the background color is used as the fill color. See `Rectangle()` and `Text()` for more information on the fill color. The default value for `MonoBackground()` is `ZAF_MONO_WHITE`, but it may be changed by calling `Set MonoBackground()`.

```
ZafLogicalColor MonoForeground(void) const;
ZafLogicalColor SetMonoForeground(ZafLogicalColor color);
```

`MonoForeground()` specifies the color used by any drawing operation requiring a foreground color on a black and white display. When drawing shapes or textual information, the foreground color is used as the outline or text color. See `Rectangle()` and `Text()` for more information on the outline or text color. The default value for `MonoForeground()` is `ZAF_MONO_BLACK`, but it may be changed by calling `Set MonoForeground()`.

```
static OSMono ZAF_FARDATA monoTable[ZAF_MAXCOLORS];
```

The static array `monoTable` is the internal ZAF black and white color table that stores environment-specific information for each ZAF logical black and white color. It is used internally by the Zinc libraries, and should normally not be modified by the programmer.

```
virtual ZafError Mouse(int column, int line,
    ZafMouseStruct &mouse);
```

`Mouse()` is used to display the mouse cursor specified in the mouse parameter, with its hot spot specified by the column and line parameters. The column and line parameters are relative to the screen. This function is optimized for mouse cursors that have already been converted to the environment's native format, using `ConvertToOSMouse()`. If the mouse cursor has not yet been converted, `Mouse()` calls `ConvertToOSMouse()` automatically. See `ZafMouseStruct` for more information on mouse cursors. Normally, `Mouse()` returns `ZAF_ERROR_NONE`.

```
ZafPaletteStruct Palette(void) const;
ZafPaletteStruct SetPalette(ZafPaletteStruct palette);
```

`Palette()` specifies the drawing palette used by any drawing operation. The palette encases all the drawing properties in one structure. See `ZafPaletteStruct`

for more information on palettes. The default values for `Palette()` are as follows, but may be changed by calling `SetPalette()`:

Palette() fields	Default values
<code>palette.lineStyle</code>	<code>ZAF_LINE_SOLID</code>
<code>palette.fillPattern</code>	<code>ZAF_PTN_SOLID_FILL</code>
<code>palette.colorForeground</code>	<code>ZAF_CLR_BLACK</code>
<code>palette.colorBackground</code>	<code>ZAF_CLR_WHITE</code>
<code>palette.monoForeground</code>	<code>ZAF_MONO_BLACK</code>
<code>palette.monoBackground</code>	<code>ZAF_MONO_WHITE</code>
<code>palette.font</code>	<code>ZAF_FNT_DIALOG</code>

```
static OSFillPattern ZAF_FARDATA
    patternTable[ ZAF_MAXPATTERNS ] ;
```

The static array `patternTable` is the internal ZAF fill pattern table that stores environment-specific information for each ZAF logical fill pattern. It is used internally by the Zinc libraries, and should normally not be modified by the programmer.

```
virtual ZafError Pixel(int column, int line,
    ZafLogicalColor color = ZAF_CLR_DEFAULT,
    ZafLogicalColor mono = ZAF_MONO_DEFAULT);
```

`Pixel()` is used to display a single pixel at the position (*column*, *line*). The *color* and *mono* parameters specify logical colors for the pixel. If the screen on which the pixel appears supports color or grayscale, then the pixel will use the logical color *color*. Otherwise, the pixel will use the logical black and white color *mono*. Normally, `Pixel()` returns `ZAF_ERROR_NONE`.

```
virtual ZafError Polygon(int numPoints, const int
    *polygonPoints, int width = 1, bool fill = false, bool
    close = false);
```

`Polygon()` is used to display a polygon specified by the parameters. The parameter *numPoints* specifies the number of points in the parameter *polygonPoints*, and *polygonPoints* is a list of points that specify the vertices of the polygon. The *width* parameter specifies the width in pixels of the shape's outline, and the outline is drawn with the `Foreground()` color, using `LineStyle()`. The *fill* parameter specifies whether or not the inside of the shape should be filled with the `Background()` color, using `FillPattern()`. The *close*

parameter specifies whether the polygon should be closed. For example, a polygon specified by three points will have only two sides if close is false; but it will have three sides if close is true, since Polygon() will draw the third side by joining the last point with the first point. Normally, Polygon() returns ZAF_ERROR_NONE.

The following shows how to use Polygon():

```
void MyObject::Draw(void)
{
    ZafRegionStruct drawRegion = BeginDraw();
    // Set up the array of vertices for the triangle.
    int points[6];
    points[0] = drawRegion.left; // (x1, y1)
    points[1] = drawRegion.bottom;
    points[2] = drawRegion.left + drawRegion.Width() / 2; // (x2,
        y2)
    points[3] = drawRegion.top;
    points[4] = drawRegion.right; // (x3, y3)
    points[5] = drawRegion.bottom;
    // Draw a magenta triangle with a red border.
    SetBackground(ZAF_CLR_MAGENTA);
    SetForeground(ZAF_CLR_RED);
    display->Polygon(3, points, 1, true, true);
    EndDraw();
}
```

```
int preSpace, postSpace;
```

The preSpace and postSpace members are initialized in the ZafDisplay constructor, and contain the space above and below a window object in pixels, respectively. They are used in positioning window objects on their parents in such a way as to avoid crowding by adding a small amount of whitespace between them. These members are advanced, and should normally not be accessed by the programmer. They are used internally by the Zinc libraries.

```
ZafError Rectangle(const ZafRegionStruct &region, int
    width = 1, bool fill = false);
virtual ZafError Rectangle(int left, int top, int right,
    int bottom, int width = 1, bool fill = false);
```

These overloaded functions display a rectangle specified by the parameters. With the first function, the region parameter specifies the region of the rectangle; with the second, the same information is passed into the left, top, right and bottom parameters. The width parameter specifies the width in pixels of the

shape's outline, and the outline is drawn with the `Foreground()` color, using `LineStyle()`. The fill parameter specifies whether or not the inside of the shape should be filled with the `Background()` color, using `FillPattern()`. Normally, `Rectangle()` returns `ZAF_ERROR_NONE`.

```
virtual ZafError RectangleXORDiff(const ZafRegionStruct  
    *oldRegion, const ZafRegionStruct *newRegion);
```

`RectangleXORDiff()` is useful in implementing a rectangle that appears to be moved around. An example of this operation can be seen in Zinc Designer, when an object is dragged around on the parent window. During this drag operation, a rectangle the size of the object being dragged follows the mouse until it is dropped.

The *oldRegion* parameter specifies the rectangle to be erased, and the *newRegion* parameter specifies the rectangle to be drawn. The first time `RectangleXORDiff()` is called, null should be passed for *oldRegion*, and the first region specifying the rectangle to be drawn should be passed for *newRegion*. Then during the rectangle's movement, the region previously passed to *newRegion* should be passed to *oldRegion*, and the region specifying where the rectangle should be drawn next should be passed to *newRegion*. The final call to `RectangleXORDiff()` should be passed null for *newRegion*. Normally, `RectangleXORDiff()` returns `ZAF_ERROR_NONE`.

The following is an example of how to use `RectangleXORDiff()`:

```
void MyWindow::DrawRubberBand(ZafPositionStruct  
    downClickPosition)  
{  
    // Prepare for drawing operations.  
    ZafRegionStruct drawRegion = BeginDraw();  
    SetBackground(ZAF_CLR_WHITE);  
    SetForeground(ZAF_CLR_BLACK);  
    // Draw a sizing rubber-band following the mouse.  
    ZafRegionStruct newRegion, oldRegion;  
    oldRegion.left = oldRegion.right = drawRegion.left +  
        downClickPosition.column;  
    oldRegion.top = oldRegion.bottom = drawRegion.top +  
        downClickPosition.line;  
    newRegion = oldRegion;  
    RectangleXORDiff(ZAF_NULLP(ZafRegionStruct), &newRegion);  
    ZafEventStruct event;  
    ZafEventType ccode = L_BEGIN_SELECT;  
    do  
    {  
        // Get the mouse movement events.  
        eventManager->Get(event, Q_NORMAL);
```

```

        // LogicalEvent() normalizes the mouse position.
        ccode = LogicalEvent(event);
        // Put the new mouse position into the region.
        newRegion.right = drawRegion.left + event.position.column;
        newRegion.bottom = drawRegion.bottom + event.position.line;
        RectangleXORDiff(&oldRegion, &newRegion);
        oldRegion = newRegion;
    } while (ccode != L_END_SELECT);
    // Erase the rubber-band and clean up.
    RectangleXORDiff(&oldRegion, ZAF_NULLP(ZafRegionStruct));
    EndDraw();
}

```

```

virtual ZafError RegionCopy(const ZafRegionStruct
    &oldRegion, int newColumn, int newLine);

```

RegionCopy() visually copies whatever appears in *oldRegion* to the new position (*newColumn*, *newLine*). Since **RegionCopy()** performs a copy operation instead of a move operation, the original pixels copied are not erased. Whatever appears at (*newColumn*, *newLine*) will be overwritten by the copy. The position (*newColumn*, *newLine*) may be inside *oldRegion*, in which case some (or all) of the pixels in *oldRegion* will be overwritten. This type of operation is commonly used in scrolling the client region associated with a scroll bar (such as in a scrollable text object). Normally, **RegionCopy()** returns **ZAF_ERROR_NONE**.

```

static ZafError ResetOSBitmap(ZafBitmapStruct &bitmap,
    const ZafBitmapStruct &copy);
static ZafError ResetOSIcon(ZafIconStruct &icon, const
    ZafIconStruct &copy);
static ZafError ResetOSMouse(ZafMouseStruct &mouse, const
    ZafMouseStruct &copy);

```

ResetOSBitmap(), **ResetOSIcon()** and **ResetOSMouse()** reset the native versions of the images contained in the ZAF image structures passed into the functions to the values stored in the copy parameters. The ZAF image is untouched. These functions are advanced, and should normally not be called by the programmer. They are used internally by the Zinc libraries to reset image structures. Normally, **ZAF_ERROR_NONE** is returned.

```

OSDisplayContext RestoreDisplayContext(void);

```

See **DisplayContext()**.


```
OSDrawContext RestoreDrawContext(void);
```

See DrawContext().

```
ZafError Text(int left, int top, const ZafIChar *text, int  
    length = -1, int hotKeyIndex = -1, bool fill = false);  
ZafError Text(const ZafRegionStruct &region, const  
    ZafIChar *text, int length = -1, ZafHzJustify  
    hzJustify = ZAF_HZ_LEFT, ZafVtJustify vtJustify =  
    ZAF_VT_CENTER, int hotKeyIndex = -1, bool fill =  
    false);  
virtual ZafError Text(int left, int top, int right, int  
    bottom, const ZafIChar *text, int length = -1,  
    ZafHzJustify hzJustify = ZAF_HZ_LEFT, ZafVtJustify  
    vtJustify = ZAF_VT_CENTER, int hotKeyIndex = -1, bool  
    fill = false);
```

These overloaded functions display the text specified by the text parameter. With the first function, the left top corner of the text is drawn at the position (*left*, *top*). If *length* is -1, the entire string is drawn, and is assumed to be null-terminated; otherwise, only length characters (as opposed to bytes) are drawn. If *hotKeyIndex* is -1, none of the characters in the string is drawn as a hot key character; otherwise, the character at the position *hotKeyIndex* (zero-based) is drawn as a hot key character. A hot key character is usually presented with an underline. The *fill* parameter specifies whether or not the background of the text should be filled with the Background() color, using FillPattern(). Normally, Text() returns ZAF_ERROR_NONE.

The second and third functions are like the first, but provide additional functionality for justifying the text within a region. With the second function, the *region* parameter specifies the region of the rectangle; with the third, the same information is passed into the *left*, *top*, *right* and *bottom* parameters. The *hzJustify* parameter specifies the horizontal justification of the text within the specified region, and the *vtJustify* parameter specifies its vertical justification. Possible values for *hzJustify* are ZAF_HZ_LEFT, ZAF_HZ_CENTER and ZAF_HZ_RIGHT. Possible values for *vtJustify* are ZAF_VT_TOP, ZAF_VT_CENTER and ZAF_VT_BOTTOM. If true is passed as the *fill* parameter into these two functions, the background of the entire region specified will be filled, whether or not the text occupies the entire region.

```
virtual ZafRegionStruct TextSize(const ZafIChar *text,  
    int length = -1);
```

TextSize() calculates the width and height of the text being passed in, using the current Font(). The *text* parameter specifies the string to be measured. If *length* is -1, the entire string is calculated, and is assumed to be null-terminated; otherwise, only *length* characters (as opposed to bytes) are calculated. The region returned contains the string's width in its right field, and the string's height in its bottom field. The region returned also returns its Width() and Height() attributes correctly.

ZafElement

ClassID	ClassName	EvaluatelsA
IsA	ListIndex	Next
Previous	NumberID	StringID

ZafElement, the ultimate base class for most ZAF classes, provides member variables and functions for list participation, class identification, and object identification. Almost all ZAF classes act as list heads and/or list elements. For example, windows, menus, vertical lists, horizontal lists, notebooks, tables, the event manager, the window manager, the data manager, and many other classes act as list heads, maintaining lists of ZAF objects. Other classes like strings, buttons, devices, and data act as list elements. All list elements derive from ZafElement, which provides previous and next member variables.

In addition to providing list element functionality, ZafElement also provides member variables and functions for class and object identification.

ZafElement is important as a base class, providing list and identification characteristics to a substantial number of derived classes. You will probably never instantiate this class directly, but you may want to derive your own classes from ZafElement. Many ZAF classes derive directly from ZafElement. Following are important examples:

- ZafData: low-level data such as dates, numbers, and bitmaps
- ZafConstraint: places size and location constraints on window objects
- ZafWindowObject: graphical user interface objects
- ZafRegionElement: region that reserves portions of the screen
- ZafQueueElement: stores run-time event information
- ZafDevice: input devices such as keyboard and mouse

Declaration `#include <z_list.hpp>`

Inheritance Root class

Constructor ZafElement initializes its members to the following default values:

Member Initializations

ZafElement

ClassID()	ID_ZAF_ELEMENT
ClassName()	"ZafElement"
Next()	null
NumberID()	0

Member Initializations

Previous()	null
StringID()	null

ZafElement(void);

The ZafElement class constructor is protected, thus it can only be instantiated from a derived class's constructor such as ZafButton, ZafStringData, or ZafKeyboard.

Destructor

virtual ~**ZafElement**(void);

This virtual destructor is used to free the memory associated with an instantiated ZafElement object. The ZafElement portion of the destructor destroys the StringID() associated with the object, if one has been specified.

Generally, the programmer will not directly destroy a ZafElement object, but rather the derived object. The following code shows how this destruction is accomplished:

```
// Create a string data object.
ZafElement *string = new ZafStringData("Hello World!");
...
// Free the string.
delete string;
```

The pointer assignment shown above is permitted because ZafElement is a base class to ZafStringData. When the object's destructor is called, the actual contents of the ZafStringData instance are freed because the base class destructor is declared virtual.

For complete information on the type of memory that is freed as a result of a call to the destructor, see the reference chapter on the particular object you instantiated.

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
virtual ZafClassID ClassID(void) const;
virtual ZafClassNameChar ClassName(void) const;
```

These two functions return the class identification and name for the object. The const value `ID_ZAF_ELEMENT` and string “ZafElement” are the identifiers associated with the ZafElement class, but each Zinc Application Framework class has a unique class identification. For example, here is a partial list of classes and their associated identifications:

Class	ClassID()	ClassName()
ZafWindowObject	ID_ZAF_WINDOW_OBJECT	“ZafWindowObject”
ZafButton	ID_ZAF_BUTTON	“ZafButton”
ZafDateData	ID_ZAF_DATE_DATA	“ZafDateData”
ZafKeyboard	ID_ZAF_KEYBOARD	“ZafKeyboard”

The identification associated with the class is always a const declaration, found in the source file `gbl_def.cpp`, that is the class name with an “ID_” prefix (e.g., ZafButton -> `ID_ZAF_BUTTON`). The associated name is a string equivalent to the actual class declaration (e.g., ZafKeyboard -> “ZafKeyboard”).

Class identifications are normally used in conjunction with the `IsA()` member function to indicate if an object is derived from a particular class. For instance, the following code shows how the `ClassName()` and `ClassID()` functions can be used with the `IsA()` function to determine the matching identification of a given window object.

```
// Check for an exact match.
if (object1->ClassID() == object2->ClassID())
    printf("Objects are of the same class!\n");

// Look for inheritance relationships.
if (object1->IsA(object2->ClassID())
    printf(Object1 (%s) is derived from object2 (%s).\n",
           object1->ClassName(), object2->ClassName());
else if (object2->IsA(object1->ClassID())
    printf(Object2 (%s) is derived from object1 (%s).\n",
           object2->ClassName(), object1->ClassName());
```

Since `ClassID()` and `ClassName()` are virtual functions, you will only receive the most derived name or identification of the object (most derived meaning the lowest derivation in the Zinc Application Framework hierarchy, or the exact type of class that was instantiated). For example, a `ClassName()` call to an instantiated ZafButton will result in the return value of “ZafButton”, not “ZafWindowObject” or “ZafElement”, even though you may have stored the contents of the new operation into a ZafElement pointer.

```
// Print the type of object we just created.
```

```

ZafElement *element = new ZafButton(0, 0, 10, 1, myBitmap,
    ZAF_NULLP(ZafIChar));
printf("The element is: %s\n", element->ClassName());
=====
The element is: ZafButton

```

The only way to override these return values is to scope the `ClassID()` call:

```

// Print the type of object we just created.
ZafElement *element = new ZafButton(0, 0, 10, 1, myBitmap,
    ZAF_NULLP(ZafIChar));
printf("The element is: %s\n", element->
    ZafWindowObject::ClassName());
=====
The element is: ZafWindowObject

```

```

static ZafElement *EvaluateIsA(ZafElement *element,
    ZafClassID compareID);

```

If a compiler doesn't support RTTI via `dynamic_cast()`, ZAF provides similar functionality for classes derived from `ZafElement` with a `DynamicPtrCast()` macro. `DynamicPtrCast()` calls `EvaluateIsA()`, which in turn calls `IsA()`. The intermediate step of `EvaluateIsA()` is necessary in order to avoid calling `IsA()` twice in the macro `DynamicPtrCast()`. It is important to note that for compilers that support RTTI, `DynamicPtrCast()` simply calls `dynamic_cast()`.

`EvaluateIsA()` simply passes its parameters to `IsA()`. The *element* parameter specifies a pointer to the object to be checked, and the *compareID* parameter specifies the class identification to be checked against. See `IsA()` for more information.

```

virtual bool IsA(ZafClassID compareID) const;
virtual bool IsA(ZafClassName compareName) const;

```

These overloaded functions provide the base definition for the hierarchical chain of inheritance relationships used in Zinc Application Framework. A particular instantiation of an element will not only match `IsA()` queries for `ZafElement`, but also for the derived class. These functions return true if one of the following two conditions is met:

- the object is an instantiation of the specified class
- the object is derived from the specified class

Otherwise, the function returns false.

Pre-defined Zinc Application Framework values and strings can be passed to the IsA function. Following are examples of these values:

Class	ClassID()	ClassName()
ZafElement	ID_ZAF_ELEMENT	"ZafElement"
ZafWindow	ID_ZAF_WINDOW	"ZafWindow"
ZafKeyboard	ID_ZAF_KEYBOARD	"ZafKeyboard"
ZafStringData	ID_ZAF_STRING_DATA	"ZafStringData"
ZafButton	ID_ZAF_BUTTON	"ZafButton"

In addition to ZAF pre-defined values and strings, new values and strings can be defined for objects derived from a Zinc Application Framework class. Consider the following code that defines a new class called MyClass:

```
const ZafClassID ID_MY_CLASS = 10000;

ZafClassID MyClass::classID = ID_MY_CLASS;
ZafClassNameChar MyClass::className[] = "MyClass";
```

Zinc reserves the values 0 through 9,999 for their class identifications and the “Zaf” and “Zdc” prefix for their class names (“Zdc” is reserved for the Zinc DataConnect product). All other values and prefixes can be used by developers in their applications.

For up-to-date information on these constant declarations, refer to the source file gbl_def.cpp and the full Zinc Application Framework class hierarchy.

The following code shows the proper use of the IsA function with various argument methods:

```
// Check for a string object using the identifier.
if (object->IsA(ID_ZAF_STRING))
    break;

// Check for a main window using the class name.
if (object->IsA("ZafWindow"))
    eventManager->Put(S_EXIT);

// Check for a mouse using the mouse classID.
if (device->IsA(ZafMouse::classID))
    device->SetDeviceState(DM_VIEW);

// Check for a date data class.
ZafDateData *date;
if (date->IsA(ZafDateData::className))
```

```
date = DynamicPtrCast(data, ZafDateData);
```

```
int ListIndex(void);
```

This function is an element level method (as opposed to the list method `ZafList::Index()`) used to determine an object's position in a list. It returns a zero-based value representing the position. Consider the following code:

```
ZafElement element1, element2;

ZafList list;
list.Add(&element1);
list.Add(&element2);

printf("List position of element2 = %d\n",
       element2.ListIndex());
printf("List position of element1 = %d\n",
       list.Index(element1));
=====
List position of element2 = 1
List position of element1 = 0
```

As a special warning, in addition to a 0 value representing the first position in a list, if the `ZafElement` is not a member of a list, this function also returns zero. If you are not sure that the element is indeed attached to a list, you should use the `ZafList::Index()` function instead of this function, since the list function returns -1 when the element is not found in the list.

```
ZafElement *Next(void) const;
ZafElement *Previous(void) const;
```

These functions return a pointer to the next/previous element in the list. If the `ZafElement` is not a member of a list, or if the element is the first member of a list and the `Previous()` function is called, or the last member in a list and the `Next()` function is called, the return value is null. Consider the following code:

```
// Initialize all my children.
for (ZafWindowObject *object = window->First(); object; object =
     object->Next())
    object->Event(S_INITIALIZE);

// Find the previous object in the list.
if (element->Previous())
```



```
printf("previous sibling %s\n", element->Previous()->
StringID());
```

In addition to the `ZafElement` class, the following classes overload the `Next()` and `Previous()` functions:

Class	Return value	Used by
<code>ZafData</code>	<code>ZafData *</code>	<code>ZafDataManager</code>
<code>ZafDevice</code>	<code>ZafDevice *</code>	<code>ZafEventManager</code>
<code>ZafWindowObject</code>	<code>ZafWindowObject *</code>	derived <code>ZafWindows</code>
<code>ZafConstraint</code>	<code>ZafConstraint *</code>	<code>ZafGeometryManager</code>
<code>ZafRegionElement</code>	<code>ZafRegionElement *</code>	<code>ZafDisplay</code>
<code>ZafQueueElement</code>	<code>ZafQueueElement *</code>	<code>ZafEventManager</code>
<code>ZafTableRecord</code>	<code>ZafTableRecord *</code>	<code>ZafTable</code>

These functions are overloaded to enable you to obtain the proper type of class pointer for a given parent object. The code shown above showed a common use for the overloaded `ZafWindowObject::Next()` function, as used with the `ZafWindow::First()` function. For more information on these overloads, see the appropriate class object's definition.

```
ZafNumberID NumberID(void) const;
ZafNumberID SetNumberID(ZafNumberID numberID);
```

These functions are used to associate a number identification with an instantiated Zinc Application Framework object. The base element constructor sets the number identification to 0, and in general, you are responsible for changing this value if you want a unique identifier for your object. There are several occasions, however, where the number identification is automatically set by Zinc Application Framework:

- If you instantiate a window object and attach it to a root window without a number identification, the `ZafWindowObject::Event()` function will automatically create an ascending identifier for each child object when the `S_INITIALIZE` message is processed (such as when the object's root window is added to the window manager). For example, the following code would create a window whose children had number identifications of 0 and 1:

```
ZafWindow *window = new ZafWindow(0, 0, 40, 10);
window->Add(new ZafButton(0, 0, 20, 1,
    ZAF_NULLP(ZafBitmapData), ZAF_NULLP(ZafIChar)));
window->Add(new ZafString(0, 1, 20,
    ZAF_NULLP(ZafStringData)));
```

```
windowManager->Add(window);
```

- If you instantiate any of a number of special support window objects (there will only be one of these objects per window hierarchy). The following support objects have default number identifications (the const declarations are given in `z_numid.hpp`):

Class	NumberID()
ZafBorder	ZAF_NUMID_BORDER
ZafGeometryManager	ZAF_NUMID_GEOMETRY_MANAGER
ZafMaximizeButton	ZAF_NUMID_MAXIMIZE
ZafMinimizeButton	ZAF_NUMID_MINIMIZE
ZafIcon (minimize)	ZAF_NUMID_MIN_ICON
ZafPullDownMenu	ZAF_NUMID_PULL_DOWN_MENU
ZafSystemButton	ZAF_NUMID_SYSTEM
ZafSystemButton::menu	ZAF_NUMID_SYSTEM_BUTTON_MENU
ZafScrollBar (corner)	ZAF_NUMID_C_SCROLL
ZafScrollBar (horizontal)	ZAF_NUMID_HZ_SCROLL
ZafScrollBar (vertical)	ZAF_NUMID_VT_SCROLL
ZafTitle	ZAF_NUMID_TITLE

- If you instantiate special pop-up items that use the special `ZafPopUpItemType` enumeration value defined in `z_popup.hpp` (the const declarations are given in `z_numid.hpp`):

Enumeration	NumberID()
ZAF_CLOSE_OPTION	ZAF_NUMID_OPT_CLOSE
ZAF_MAXIMIZE_OPTION	ZAF_NUMID_OPT_MAXIMIZE
ZAF_MINIMIZE_OPTION	ZAF_NUMID_OPT_MINIMIZE
ZAF_MOVE_OPTION	ZAF_NUMID_OPT_MOVE
ZAF_RESTORE_OPTION	ZAF_NUMID_OPT_RESTORE
ZAF_SIZE_OPTION	ZAF_NUMID_OPT_SIZE
ZAF_SWITCH_OPTION	ZAF_NUMID_OPT_SWITCH

```
const ZafStringID StringID(void) const;
ZafStringID SetStringID(const ZafStringID stringID);
```

These functions are used to associate a string identification with an instantiated Zinc Application Framework object. The base element constructor sets the

string identification to null, and in general, you are responsible for changing this value if you want a unique identifier for your object. There are several occasions, however, where the string identification is automatically set by Zinc Application Framework:

- If you instantiate a window object and attach it to a root window, without a string identification, the `ZafWindowObject::Event()` function will automatically create an incremented field identifier when the `S_INITIALIZE` message is processed (such as when the object's root window is added to the window manager). For example, the following code would create a window whose children had string identifications of "FIELD_0" and "FIELD_1":

```
ZafWindow *window = new ZafWindow(0, 0, 40, 10);
window->Add(new ZafButton(0, 0, 20, 1,
    ZAF_NULLP(ZafBitmapData), ZAF_NULLP(ZafIChar)));
window->Add(new ZafString(0, 1, 20,
    ZAF_NULLP(ZafStringData)));
windowManager->Add(window);
```

- If you instantiate any of a number of special support window objects (there will only be one of these objects per window hierarchy). The following support objects have default string identifications:

Class	StringID()
ZafBorder	"ZAF_NUMID_BORDER"
ZafGeometryManager	"ZAF_NUMID_GEOMETRY"
ZafMaximumButton	"ZAF_NUMID_MAXIMIZE"
ZafMinimizeButton	"ZAF_NUMID_MINIMIZE"
ZafIcon (minimize)	"ZAF_NUMID_MIN_ICON"
ZafPullDownMenu	"ZAF_NUMID_PULL_DOWN_MENU"
ZafSystemButton	"ZAF_NUMID_SYSTEM"
ZafSystemButton::menu	"ZAF_NUMID_SYSTEM_BUTTON_MENU"

ZafEventManager

Blocked	DestroyEvent	DeviceImage
DevicePosition	DeviceState	Event
Get	PollDevices	Put
ReadFromBeginning	ReadFromEnd	

ZafEventManager is the top-level class used to manage all the events and supported devices on the system. Some systems have a native event manager that ZafEventManager monitors. In these cases, the ZafEventManager packages the native events into ZAF events and posts them on its event queue for processing by the ZAF system. Generally, native devices such as mice and keyboards are automatically polled by the native environment, so ZAF receives device events directly from the native event manager. Some environments such as DOS must poll each input device using its Poll() function (see ZafDevice for more information). The Poll() function posts any device events on the event manager's queue.

To cause the event manager to manage a device, simply add the device to the event manager's list with the Add() function.

Declaration `#include <z_evtmgr.hpp>`

Inheritance `ZafEventManager : ZafList`

Constructor `ZafEventManager(int noOfQueueEvents = 100);`

This constructor should normally not be called by the programmer, since it is called by the ZafApplication constructor. The *noOfQueueEvents* parameter specifies the maximum number of events the internal event queue should hold. Static members of ZafDevice are initialized in this constructor. These include display and eventManager.

Destructor `virtual ~ZafEventManager(void);`

This destructor is used to free the memory associated with a ZafEventManager object. It chains to the ZafList destructor.

Generally, the programmer will not directly destroy a ZafEventManager object, since it is automatically destroyed when the ZafApplication object is destroyed.

Members `bool Blocked(ZafQFlags qFlags) const;`

Blocked() returns false if *qFlags* contains the Q_NO_BLOCK flag. Otherwise, it returns true, reflecting the default of Q_BLOCK. See Get() for more information.

```
bool DestroyEvent(ZafQFlags qFlags) const;
```

DestroyEvent() returns false if *qFlags* contains the Q_NO_DESTROY flag. Otherwise, it returns true, reflecting the default of Q_DESTROY. See Get() for more information.

```
ZafEventType DeviceImage(ZafDeviceType deviceType,  
                           ZafDeviceImage deviceImage);
```

The DeviceImage() function finds the first device of type *deviceType* attached to the event manager and changes its device image to *deviceImage*. The return value is usually *deviceImage*.

An example of calling this function follows:

```
// Change the mouse image to DM_WAIT.  
eventManager->DeviceImage(E_MOUSE, DM_WAIT);
```

```
ZafEventType DevicePosition(ZafDeviceType deviceType,  
                              int column, int line);
```

The DevicePosition() function finds the first device of type *deviceType* attached to the event manager and sends it a D_POSITION event, with event.position (*column*, *line*). The return value is D_POSITION for environments that allow the device to be programmatically positioned, or S_UNKNOWN for environments that disallow it.

An example of calling this function follows:

```
// Change the mouse image to the top left of the screen.  
eventManager->DevicePosition(E_MOUSE, 0, 0);
```

```
ZafEventType DeviceState(ZafDeviceType deviceType,  
                           ZafDeviceState deviceState);
```

The DeviceState() function finds the first device of type *deviceType* attached to the event manager and changes its device state to *deviceState*. If *deviceState* is D_STATE, the current state of the device is not changed, but is returned. The return value is usually *deviceState*.

An example of calling this function follows:

```
// Find the mouse state.
ZafEventType mouseState = eventManager->DeviceState(E_MOUSE,
    D_STATE);
```

```
virtual ZafEventType Event(const ZafEventStruct &event,
    ZafDeviceType deviceType = E_DEVICE);
```

The `Event()` function passes an event to devices attached to the event manager. If *deviceType* is `E_DEVICE`, the event is passed to all the devices for processing. Otherwise, the event is passed only to the devices of type *deviceType*. `Event()` returns the return code from the last device that processed the event. See `ZafDevice::Event()` for more information.

```
virtual int Get(ZafEventStruct &event, ZafQFlags flags =
    Q_NORMAL);
```

`Get()` returns the next available event in the `ZafEventManager`'s queue, according to the *flags* passed into the flags parameter. The default behavior is specified with `Q_NORMAL`. The following are the flags that may be OR'd together and passed to `Get()`:

Flag	Description
<code>Q_NO_BLOCK</code>	Causes <code>Get()</code> to always return, even if no event is available (default is <code>Q_BLOCK</code>)
<code>Q_NO_DESTROY</code>	Causes <code>Get()</code> to leave the event on the queue for retrieval later also (default is <code>Q_DESTROY</code>)
<code>Q_END</code>	Causes <code>Get()</code> to retrieve the event at the end of the queue (default is <code>Q_BEGIN</code>)
<code>Q_NO_POLL</code>	Causes <code>Get()</code> to not poll the devices (default is <code>Q_POLL</code>)

In environments that have native event managers, `Get()` looks at the native event manager's queue and puts the next available event packaged up as a ZAF event on the ZAF event queue. Native events packaged up as ZAF events have an event.type of `E_OSEVENT`, and must be processed by `ZafWindowObject::LogicalEvent()` before the ZAF event structure is fully initialized. See `ZafWindowObject::Event()` and `ZafWindowObject::LogicalEvent()` for more information.

During `Get()`, the `ZafEventManager` calls `Poll()` once for all the devices attached to its queue if the `Q_NO_POLL` flag is not set. If `Q_NO_BLOCK` was not passed into the *flags* parameter and no event is put on the queue during

device polling, `Get()` will keep polling the devices (or looking at the native event manager's queue) until an event is posted.

`Get()` returns the next available event on the ZAF event queue, usually from the beginning of the queue, but if `Q_END` is passed into the *flags* parameter, it will return the event on the end of the queue. If a valid event was returned in the *event* parameter, `Get()` returns 0. Otherwise, some non-zero value is returned.

```
bool PollDevices(ZafQFlags qFlags) const;
```

`PollDevices()` returns false if *qFlags* contains the `Q_NO_POLL` flag. Otherwise, it returns true, reflecting the default of `Q_POLL`. See `Get()` for more information.

```
virtual void Put(const ZafEventStruct &event, ZafQFlags  
                  flags = Q_END);
```

`Put()` posts the event passed into the *event* parameter on the `ZafEventManager`'s queue, according to the *flag* passed into the *flags* parameter. If `Q_END` is specified, the event is placed on the end of the queue. If `Q_BEGIN` is specified, the event is placed on the beginning of the queue.

```
bool ReadFromBeginning(ZafQFlags qFlags) const;
```

`ReadFromBeginning()` returns false if *qFlags* contains the `Q_END` flag. Otherwise, it returns true, reflecting the default of `Q_BEGIN`. See `Get()` and `Put()` for more information.

```
bool ReadFromEnd(ZafQFlags qFlags) const;
```

`ReadFromEnd()` returns false if *qFlags* does not contain the `Q_END` flag. Otherwise, it returns true, reflecting the default of `Q_BEGIN`. See `Get()` and `Put()` for more information.

ZafFormatData

FormattedText

ZafFormatData serves as the base class to all formattable ZAF data classes, including ZafBignumData, ZafDateData, ZafIntegerData, ZafStringData, ZafRealData, ZafTimeData, and ZafUTimeData. A common aspect of these derived classes is their use of string manipulation functions, the most apparent being the public FormattedText() member. In addition to this function, however, ZafFormatData also provides protected Sprintf() and Sscanf() members that allow derived objects to format and manipulate string information that can be used for the screen presentation of their data.

ZafFormatData is an abstract class. Its abstract nature is implied by the base ZafData pure virtual functions and by the newly declared pure virtual function called FormattedText(). Thus, derived format classes must resolve the abstractions inherited from the base ZafData and ZafFormatData classes.

Declaration `#include <z_fdata.hpp>`

Inheritance `ZafFormatData : ZafData : (ZafElement, ZafNotification)`

Constructors All ZafFormatData constructors initialize the member variables associated with an instantiated ZafFormatData object. The default values set by the ZafFormatData constructor or overridden from those set by base class constructors follow:

Member Initializations

ZafElement

ClassID()	ID_ZAF_FORMAT_DATA
ClassName()	"ZafFormatData"

ZafFormatData(void);

The ZafFormatData class constructor, like the ZafData constructor, is protected. The primary purpose of this constructor is to chain the ZafData portion of the class with its base class constructors. Please refer to the specific data object you are using for complete information about the object's construction.

ZafFormatData(const ZafIChar *name, ZafDataPersistence &persist);

This constructor is used for persistence. The parameters and values of this constructor are deferred to the ZafWindow section of this manual, since most persistence is done at the ZafWindow level.

Destructor

```
virtual ~ZafFormatData(void);
```

This virtual destructor is used to free the memory associated with an instantiated ZafFormatData object. Since ZafFormatData does not define any new members, its destructor simply chains to the ZafData class destructor.

A ZafFormatData pointer may be deleted even though the class definition is abstract. This is done by allocating a derived data object and then by setting the returned object to a format data pointer. The following code shows the correct use of the ZafFormatData destructor under these conditions:

```
// Create a string data object.
ZafFormatData *string = new ZafStringData("Hello World!");
...
// Free the string.
delete string;
```

The pointer assignment shown above is permitted because ZafFormatData is a base class to ZafStringData. When the string object's destructor is called, the actual contents of the ZafStringData instance are freed because the base class destructor is declared virtual.

For complete information on the type of memory that is freed as a result of a call to the destructor, see the reference chapter on the particular object you created.

Members

```
virtual int FormattedText(ZafIChar *buffer, int
    maxLength, const ZafIChar *format = 0) const = 0;
```

This is a pure virtual function that abstractly defines "formatted text retrieval" functionality for a derived object. The following example illustrates what the FormattedText() function returns for several derived data objects:

```
// Show various results of FormattedText().
ZafIChar buffer[256];

ZafDateData date(1, 1, 59);
date->FormattedText(buffer, sizeof(buffer));
printf("date - %s\n", buffer);

ZafIntegerData integer(100);
integer->FormattedText(buffer, sizeof(buffer), "%d");
printf("integer - %s\n", buffer);
```

```
ZafStringData string("hello");  
string->FormattedText(buffer, sizeof(buffer));  
printf("string - %s\n", buffer);
```

```
=====  
date - 01/01/1959  
integer - 100  
string - hello
```

The specific set of formatting capabilities is defined by the derived class. (See the `FormattedText()` description of each class for additional information.)

Zinc Application Framework uses this function in conjunction with its window objects to get the formatted presentation of a data object that can be presented to the screen. This allows the user interface component to pass and manipulate string information, while allowing the data portion to remain in a compact and precise internal representation.

ZafHelpTips

Event	HelpObject	HelpTipsType
InitialDelay	NewHelpTipDelay	Poll
SetUserPalette		

ZafHelpTips is a device class that can display limited help information when a user moves the mouse over a viewable object. The information displayed may consist of a “quick tip,” a “help tip,” or both.

“Quick tips” are small, temporary windows that contain a short message. ZAF’s implementation is similar to Microsoft Windows style “tool tips” except that quick tips may be invoked by almost any user interface object—not just toolbars. Quick tips may include single or multiple lines of text.

To invoke a quick tip the user must move the mouse over an object, then pause briefly. The quick tip is removed if the user moves the mouse outside the region of the object, clicks the mouse, or presses any key. SetQuickTip() must be called by each ZafWindowObject to set the text to be displayed. (See ZafWindowObject for more information.)

“Help tips” are help messages that are displayed in a traditional user interface object instead of a temporary window. For example, a status bar may display a string that changes as the mouse moves over different objects.

To invoke a help tip the user must move the mouse over an object. The help tip is removed when the user moves the mouse outside the object. SetHelpObjectTip() must be called by each ZafWindowObject to set the text to be displayed. (See ZafWindowObject for more information.) SetHelpObject() must also be called once to initialize ZafHelpTips to point at the object that will display the help tip. SetHelpObject() must be called by the programmer since this pointer is often global to an application.

Note: Quick tips and help tips require that the ZafHelpTips device first be added to the ZafEventManager.

Declaration `#include <z_https.hpp>`

Inheritance `ZafHelpTips : ZafDevice : ZafElement`

Constructor

The ZafHelpTips constructor initializes the member variables associated with a new ZafHelpTips object. The default values set by ZafHelpTips follow, if they are overridden from those set by base class constructors:

Member Initializations**ZafHelpTips**

HelpObject()	null
HelpTipsType()	ZAF_HELPTIPS_QUICKTIP
InitialDelay()	50 (milliseconds)
NewHelpTipDelay()	50 (milliseconds)

ZafDevice

DeviceType()	E_HELPTIPS
--------------	------------

ZafElement

ClassID()	ID_ZAF_HELPTIPS
ClassName()	"ZafHelpTips"

```
ZafHelpTips(ZafDeviceState state = D_OFF, ZafHelpTipsType
             tHelpTipsType = ZAF_HELPTIPS_QUICKTIP);
```

This constructor is used to instantiate a ZafHelpTips object to be added to a ZafEventManager object. *tHelpTipsType* specifies the type of help tips to be handled by the ZafHelpTips object. For a discussion on device *state*, see the ZafDevice section of this manual.

To enable help tips in an application the following code could be used:

```
int ZafApplication::Main(void)
{
    ...

    // Enable help tips in my application.
    ZafHelpTips *helpTips = new ZafHelpTips(D_ON,
        ZAF_HELPTIPS_BOTH);
    zafEventManager->Add(helpTips);

    ...
}
```

Destructor

```
virtual ~ZafHelpTips(void);
```

The virtual destructor is used to free the memory associated with an instantiated ZafHelpTips object. Normally ZafHelpTips will be destroyed when the ZafEventManager is destroyed during normal application termination.

Members

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

ZafHelpTips processes several events in addition to those handled by all ZafDevice objects.

Event (Device Request)	Action
DH_HELP_TIPS_TIMER	Immediately expires the ZafHelpTips timer causing the device to display event.windowObject-> QuickTip().
DH_SET_HELP_OBJECT	Causes ZafHelpTips to reassign its help tips object to event.windowObject. See SetHelpObject().
DH_UPDATE_HELP_OBJECT	Causes ZafHelpTips to display event.windowObject->HelpObjectTip() in the current help object.
N_MOUSE_LEAVE	Causes ZafHelpTips to remove its quick tip and to update help object with its previous text contents.
N_MOUSE_ENTER	Causes ZafHelpTips to activate the timer that must elapse before a quick tip is displayed. The help object is immediately updated with event.windowObject-> HelpObjectTip().

```
ZafWindowObject *HelpObject(void) const;
```

```
virtual void SetHelpObject(ZafWindowObject *helpObject);
```

SetHelpObject() sets the pointer to the help object of a ZafHelpTips instance. *helpObject* should point at the text-capable ZafWindowObject that will display future help tips. HelpObject() returns a pointer to the current help object.

Users may find it helpful to have different help objects in different windows. For example, as a user moves from one window to another, the help object may also move to the new window. The following code shows this technique.

```
// Handle the N_CURRENT message inside a derived window's
// Event() function. Change the help object to point at
// helpBarObject (a string object displayed on the status bar
// of the current window).
```

```
case N_CURRENT:
{
    ZafEventStruct httpEvent(DH_SET_HELP_OBJECT);
```

```

        httpEvent.windowObject = helpBarObject;
        eventManager->Event(httpEvent, E_HELPSTIPS);
    }
    break;

```

```

ZafHelpTipsType HelpTipsType(void) const;
virtual ZafHelpTipsType SetHelpTipsType(ZafHelpTipsType
    helpTipsType);

```

HelpTipsType() returns the type of help tips which are controlled by a ZafHelpTips object. The default value of this attribute is **ZAF_HELPSTIPS_QUICKTIP** but the user may call **SetHelpTipsType()** to change it. Here are the possible types of help tips:

ZafHelpTipsType	Description
ZAF_HELPSTIPS_QUICKTIP	Causes ZafHelpTips display only quick tips.
ZAF_HELPSTIPS_HELPOBJECT	Causes ZafHelpTips to display only help tips.
ZAF_HELPSTIPS_BOTH	Causes ZafHelpTips to manage both quick tips and help tips.

```

int InitialDelay(void);
static int SetInitialDelay(int initialDelay);

```

SetInitialDelay() sets the initial delay of a ZafHelpTips device. *initialDelay* is specified in milliseconds and indicates the timer interval that must elapse before the quick tip of a user interface object is displayed on the desktop.

```

int NewHelpTipDelay(void);
static int SetNewHelpTipDelay(int newHelpTipDelay);

```

SetNewHelpTipDelay() indicates the newHelpTipDelay of a ZafHelpTips device in milliseconds. This interval timer begins after exiting an object for which a quick tip has been displayed, and continues until another object is entered for which a quick tip exists, or until the timer expires. If this timer interval does not expire before the sibling is reached the sibling's quick tip is immediately displayed instead of waiting the normal **InitialDelay()** period.

```

virtual void Poll(void);

```

This virtual function is called by `ZafEventManager::Get()`. `Get()` checks the timer of the active `ZafHelpTips` device. `Get()` may cause a quick tip to be displayed, removed, or the timer to be reseeded. Users will not typically call this function.

```
void SetUserPalette(ZafPaletteStruct *userPalette);
```

`SetUserPalette()` allows quick tips to be displayed using custom colors or fonts. *userPalette* is a pointer to a valid `ZafPaletteStruct` that should already contain complete color and font information. If a `userPalette` is not assigned to the `ZafHelpTips` device, the quick tip will be displayed in its default colors. (Default colors are platform-specific and are assigned when the `ZafHelpTips` device is initially constructed.)

ZafHzList

AutoSortData	SetBackgroundColor	CellHeight
Cell Width	SelectionType	SetFont
SetTextColor		

ZafHzList object is a custom list object that arranges items horizontally in as many rows as will fit in the client region. A ZafHzList object may have a horizontal scroll bar associated with it. Like ZafVtList it supports single, multiple and extended selection methods.

Declaration	#include <z_hlist.hpp>
Inheritance	ZafHzList : ZafWindow : ((ZafWindowObject : ZafElement), ZafList)
Constructors	All ZafHzList constructors initialize the member variables associated with an instantiated ZafHzList object. The default values set by the ZafHzList and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafHzList. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafHzList	
AutoSortData()	false
CellHeight()	user-supplied parameter
CellWidth()	user-supplied parameter
ZafWindow	
Destroyable()	false†
Locked()	false†
Maximized()	false†
Minimized()	false†
Moveable()	false†
Sizeable()	false†
ZafWindowObject	
Bordered()	true
ZafElement	
ClassID()	ID_ZAF_HZ_LIST
ClassName()	"ZafHzList"


```
ZafHzList(int left, int top, int width, int height, int  
          cellWidth, int cellHeight);
```

This constructor is useful in straight-code situations. *left*, *top*, *width* and *height* specify the list's position and size on its parent. *cellWidth* and *cellHeight* specify the fixed width and height of each list item for positioning purposes. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired (see `ZafWindowObject::SetCoordinateType`).

```
ZafHzList(const ZafHzList &copy);
```

The copy constructor calls the overloaded `Duplicate()` to create a new `ZafHzList` object and initialize its data from *copy*.

```
ZafHzList(const ZafIChar *name, ZafObjectPersistence  
          &persist);
```

The final constructor is used for persistence. Refer to `ZafWindow` for more information, since most persistence is done at the `ZafWindow` level.

Sample `ZafHzList` creation techniques follow:

```
// Create a sample window with a horizontal list of strings.  
ZafWindow *window1 = new ZafWindow(0, 0, 50, 10);  
  
// Create the horizontal list object.  
ZafHzList *hList1 = new ZafHzList(1, 1, 20, 5, 10, 1);  
  
// Add a scroll bar and the strings to the horizontal list.  
hList1->Add(new ZafScrollBar(0, 0, 0, 0));  
hList1->Add(new ZafString(0, 0, 20, "String 1", -1));  
hList1->Add(new ZafString(0, 0, 20, "String 2", -1));  
hList1->Add(new ZafString(0, 0, 20, "String 3", -1));  
hList1->Add(new ZafString(0, 0, 20, "String 4", -1));  
hList1->Add(new ZafString(0, 0, 20, "String 5", -1));  
hList1->Add(new ZafString(0, 0, 20, "String 6", -1));  
  
// Add the list to the window.  
window1->Add(hList1);  
...  
// Create a sample window with a horizontal list of buttons.  
ZafWindow *window2 = new ZafWindow(10, 10, 50, 10);  
  
// Create the horizontal list object and its children.
```

```

ZafHzList *hList2 = new ZafHzList(1, 1, 20, 5, 10, 1);

// Allow the list children to draw bitmap information.
hList2->SetOSDraw(false);
extern ZafBitmapData *bitmap1, *bitmap2, *bitmap3, *bitmap4;
hList2->Add(new ZafButton(0, 0, 20, 1, bitmap1,
    ZAF_NULLP(ZafIChar)));
hList2->Add(new ZafButton(0, 0, 20, 1, bitmap2,
    ZAF_NULLP(ZafIChar)));
hList2->Add(new ZafButton(0, 0, 20, 1, bitmap3,
    ZAF_NULLP(ZafIChar)));
hList2->Add(new ZafButton(0, 0, 20, 1, bitmap4,
    ZAF_NULLP(ZafIChar)));

// Add the list to the window.
window2->Add(hList2);

```

Destructor

```
virtual ~ZafHzList(void);
```

The destructor is used to free the memory associated with a ZafHzList object. It chains to the ZafWindow, ZafList, ZafWindowObject and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafHzList object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. If the Set*() function does not successfully change the state as requested, however, it will instead return the current state.

```

bool AutoSortData(void) const;
virtual bool SetAutoSortData(bool autoSortData);

```

If AutoSortData() is true, the list will automatically sort its children as they are added to the list. The function returned by CompareFunction() is used to sort the children. By default, sorting is done in alphabetical order, but SetCompareFunction() may be called to provide a custom sorting function. See ZafList::CompareFunction() for more information about sorting list children. The default value of this attribute is false, but the user may call SetAutoSortData() to change it.

```
virtual ZafLogicalColor  
    SetBackgroundColor(ZafLogicalColor color,  
        ZafLogicalColor mono = ZAF_MONO_NULL);
```

To provide consistency in the appearance of list children, the ZafHzList object sets the ParentPalette() attribute on each of its children—therefore all children use the background color of the ZafHzList (see ZafWindowObject::ParentPalette()). This overloaded function allows this color to be changed.

```
int CellHeight(void) const;  
int CellWidth(void) const;  
virtual int SetCellHeight(int cellHeight);  
virtual int SetCellWidth(int cellWidth);
```

CellHeight() and CellWidth() specify the height and width of each item in the list. Each list item is allocated this size for spacing purposes. CellHeight() and CellWidth() are specified by CoordinateType() (see ZafWindowObject). The constructor assumes cell coordinates unless SetCoordinateType is used.

```
ZafSelectionType SelectionType(void) const;  
virtual ZafSelectionType  
    SetSelectionType(ZafSelectionType selectionType);
```

ZafHzLists may allow different types of selection behavior. SetSelectionType() allows this behavior to be changed from the single-selection default. Valid values are listed.

SelectionType()	Description
ZAF_SINGLE_SELECTION	Allows only one item to be selected. If another item is selected any previously selected item is deselected.
ZAF_MULTIPLE_SELECTION	Allows multiple items to be selected. "Selection actions," including mouse clicks, cause the selection state of an item to be toggled. The state of other list items is unchanged.
ZAF_EXTENDED_SELECTION	Allows multiple items to be selected, and does this using native multiple- and extended-selection techniques. For example, a single click might act as single-select, shift-click might select a range of items, and ctrl-click might act as multiple-select.

```
virtual ZafLogicalFont SetFont(ZafLogicalFont font);
```

To provide consistency in the appearance of list children, the ZafHzList object sets the ParentPalette() attribute on each of its children—therefore all children use the font of the ZafHzList (see ZafWindowObject::ParentPalette()). This overloaded function allows the font to be changed.

```
virtual ZafLogicalColor SetTextColor(ZafLogicalColor  
    color, ZafLogicalColor mono = ZAF_MONO_NULL);
```

To provide consistency in the appearance of list children, the ZafHzList object sets the ParentPalette() attribute on each of its children—therefore all children use the text color of the ZafHzList (see ZafWindowObject::ParentPalette()). This overloaded function allows the text color to be changed.

ZaflImage

AutoSize	PathID	PathName
Scaled	Tiled	Wallpaper

The ZaflImage object supports the display of native image types such as bitmaps and pictures. In Microsoft Windows and OS/2, bitmaps are supported. In Motif, both xbm and xpm bitmaps are supported. In DOS, PCX images are supported. On the Macintosh, PICT resources are supported.

Declaration `#include <z_image.hpp>`

Inheritance `ZaflImage : ZafWindowObject : ZafElement`

Constructors All ZaflImage constructors initialize the member variables associated with an instantiated ZaflImage object. The default values set by the ZaflImage and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZaflImage. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZaflImage

AutoSize()	false
PathID()	user-supplied parameter
PathName()	user-supplied parameter
Scaled()	false
Tiled()	false
Wallpaper()	false

ZafWindowObject

AcceptDrop()	false [†]
CopyDraggable()	false [†]
Focus()	false [†]
Font()	ZAF_FNT_NULL [†]
HelpContext()	null [†]
HelpObjectTip()	null [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
Noncurrent()	true [†]
OSDraw()	false
TextColor()	ZAF_CLR_NULL [†]

Member Initializations

ZafElement

ClassID()	ID_ZAF_IMAGE
ClassName()	"ZafImage"

```
ZafImage(int left, int top, int width, int height, const
          ZafIChar *pathName, int pathID = -1);
```

This constructor is useful in straight-code situations. The *left* and *top* parameters specify the position where the left and top of the object will be placed on its parent. The *width* and *height* parameters specify the width and height of the object. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. The *pathName* parameter may be either (1) the path and filename specifying the file on disk that contains the image, or (2) the resource name of the image in the application file or an open system file. The *pathID* parameter is the resource ID number of the image in the application file or an open system file. Usually, either *pathName* or *pathID* will be used, but not both. If *pathName* is null, it is ignored. If *pathID* is -1, it is ignored.

```
ZafImage(const ZafImage &copy);
```

The copy constructor is used in conjunction with the overloaded Duplicate() function. It accepts another ZafImage object and copies the object's information.

```
ZafImage(const ZafIChar *name, ZafObjectPersistence
          &persist);
```

The final constructor is used for persistence. Refer to ZafWindow for more information, as most persistence is done at the ZafWindow level.

Sample ZafImage creation techniques follow:

```
// Create a sample window with an image.
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);
// Load the image found in the file MYIMAGE.
window1->Add(new ZafImage(0, 1, 10, 6, "MYIMAGE"));
...
// Create a sample window with an image.
ZafWindow *window2 = new ZafWindow(10, 10, 40, 10);
// Load the image found in the application file with resource ID
3000.
```

```
const int myImageID = 3000;
window1->Add(new ZafImage(0, 1, 10, 6, ZAF_NULLP(ZafIChar),
    myImageID));
```

Destructor

```
virtual ~ZafImage(void);
```

This destructor is used to free the memory associated with a **ZafImage** object. It chains to the **ZafWindowObject** and **ZafElement** destructors.

Generally, the programmer will not directly destroy a **ZafImage** object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see **ZafWindow::~ZafWindow()**.

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the **Set*()** function. If the **Set*()** function does not successfully change the state as requested, however, it will instead return the current state.

```
bool AutoSize(void) const;
virtual bool SetAutoSize(bool autoSize);
```

If **AutoSize()** is true, the image will automatically adjust the size of its background area to match the size of its displayed image. This is particularly useful if the image has a border, and the border is to exactly surround the image.

Note that the original region (passed into the constructor or read from the persistent object file) is preserved internally by ZAF and used if region computations are required by later programmer interaction with the image.

If **AutoSize()** is false, the image will display with its original region. The default value of this attribute is false, but the user may call **SetAutoSize()** to change it.

```
int PathID(void);
virtual ZafError SetPathID(int pathID);
```

The **PathID()** is the resource ID number of the native image in the application file or an open system file. Finding a resource by ID number is generally faster than finding a resource by name, if the resource is in the application file or an open system file. If **PathID()** is -1, it is ignored. The default value of this attribute is -1, but the user may call **SetPathID()** to change it.

```
const ZafIChar *PathName(void);
virtual ZafError SetPathName(const ZafIChar *pathName);
```

The `PathName()` may be either the resource name of the native image in the application file or an open system file, or a path and filename of the file containing the resource. Finding a resource by ID number is generally faster than finding a resource by name, if the resource is in the application file or an open system file. If `PathName()` is null it is ignored. The user may pass this attribute's initial value into the constructor, and the user may call `SetPathID()` to change it.

```
bool Scaled(void) const;
virtual bool SetScaled(bool scaled);
```

If `Scaled()` is true, the image will automatically adjust its size (either stretching or shrinking) to exactly match the size of its containing region. `Scaled()` and `Tiled()` should never be true at the same time, as the resulting behavior is undefined.

If `Scaled()` is false, the image will display with its original size. The default value of this attribute is false, but the user may call `SetScaled()` to change it.

```
bool Tiled(void) const;
virtual bool SetTiled(bool tiled);
```

If `Tiled()` is true, the image will display enough copies of itself within its containing region so as to completely fill the region. Some copies of itself on the right and bottom of the containing region may be clipped to the region. `Scaled()` and `Tiled()` should never be true at the same time, as the resulting behavior is undefined.

If `Tiled()` is false, the image will display just one copy of itself. The default value of this attribute is false, but the user may call `SetTiled()` to change it.

```
bool Wallpaper(void) const;
virtual bool SetWallpaper(bool wallpaper);
```

If `Wallpaper()` is true, the image will automatically become a `SupportObject()`, adjust its background to occupy its parent's entire client region, and display behind all other objects in its parent's client region. If a `Wallpaper()` image is neither `Scaled()` nor `Tiled()`, the image will be centered within its parent's client region.

If `Wallpaper()` is false, the image will display as a normal child object. The default value of this attribute is false, but the user may call `SetWallpaper()` to change it.

ZafInteger

Event	IntegerData
-------	-------------

The ZafInteger object is a single-line integer object that allows user input through the keyboard. ZafInteger inherits base class functionality from ZafString, allowing support for operations such as copy/cut/paste.

Declaration `#include <z_intl.hpp>`

Inheritance `ZafInteger : ZafString : ZafWindowObject : ZafElement`

Constructors All ZafInteger constructors initialize the member variables associated with an instantiated ZafInteger object. The default values set by the ZafInteger and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafInteger. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafInteger

<code>IntegerData()</code>	<code>null</code>
<code>ZafString</code>	
<code>LowerCase()</code>	<code>false</code> [†]
<code>Password()</code>	<code>false</code> [†]
<code>StringData()</code>	<code>null</code> [†]
<code>UpperCase()</code>	<code>false</code> [†]
<code>VariableName()</code>	<code>false</code> [†]

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_INTEGER</code>
<code>ClassName()</code>	<code>"ZafInteger"</code>

ZafInteger(int left, int top, int width, long value);

This constructor is useful in straight-code situations, particularly if you wish the ZafInteger object to create, maintain and destroy its own ZafIntegerData object automatically. The *left* and *top* parameters specify the position where the left and top of the object will be placed on its parent. The *width* parameter specifies the width of the object. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. The

value parameter is the value you wish to initially appear in the new ZafInteger object.

```
ZafInteger(int left, int top, int width, ZafIntegerData
            *integerData = ZAF_NULLP(ZafIntegerData));
```

This constructor is useful in straight-code situations where a ZafIntegerData object has already been created. This constructor could be used to maintain data pieces yourself, rather than having the ZafInteger class create and maintain the data pieces automatically. For example, to maintain a database of ZafIntegerData objects and tie them into ZafInteger objects, maintain your own ZafIntegerData objects and create ZafInteger objects using your ZafIntegerData objects by passing them into the integerData parameter of this constructor. For more information on using ZafIntegerData objects, see the chapter on ZafIntegerData. The *left*, *top* and *width* parameters are the same as the previous constructor.

```
ZafInteger(const ZafInteger &copy);
```

The copy constructor is used in conjunction with the overloaded Duplicate() function. It accepts another ZafInteger object and copies the object's information. If the data objects are StaticData(), then the new ZafInteger object simply points to the original data objects. If the data objects are not StaticData(), then a copy is made for the new ZafInteger object. This behavior allows a programmer to use static data for more than one ZafInteger object.

```
ZafInteger(const ZafIChar *name, ZafObjectPersistence
            &persist);
```

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

Sample ZafInteger creation techniques follow:

```
// Create a sample window with integer objects.
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);
// Create integers and pass in the values directly.
window1->Add(new ZafInteger(0, 1, 25, 100));
window1->Add(new ZafInteger(0, 2, 25, 200));
...
// Create a sample window with integer objects.
ZafWindow *window2 = new ZafWindow(10, 10, 40, 10);
// Create integer data objects.
ZafIntegerData *integerData1 = new ZafIntegerData(100);
```

```
ZafIntegerData *integerData2 = new ZafIntegerData(200);  
// Create integers that use the data previously created.  
window2->Add(new ZafInteger(0, 1, 25, integerData1));  
window2->Add(new ZafInteger(0, 2, 25, integerData2));
```

Destructor

```
virtual ~ZafInteger(void);
```

The destructor is used to free the memory associated with a ZafInteger object, including all the data object pieces that are Destroyable(). It chains to the ZafString, ZafWindowObject, and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafInteger object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function handles all events that get sent to the ZafInteger object, whether by processing the events itself, or by passing them to ZafString::Event() for base class processing. See ZafWindowObject for more information.

ZafInteger handles the following events differently than its base classes:

Event()	Description
S_COPY_DATA	causes the object to copy event.windowObject's IntegerData() if event.windowObject is a ZafInteger object
S_SET_DATA	causes the object to create a new IntegerData() object, then copy into it event.windowObject's IntegerData() if event.windowObject is non-null and is a ZafInteger object

```
ZafIntegerData *IntegerData(void) const;  
virtual ZafError SetIntegerData(ZafIntegerData  
    *integerData);
```

The `IntegerData()` object is where the actual data is stored. The `IntegerData()` piece may be shared among several `ZafInteger` objects, or it may belong to a single `ZafInteger` object. If shared among several `ZafInteger` objects, all the associated `ZafInteger` objects will be updated when the `IntegerData()` piece changes. `SetIntegerData()` may be used to associate an `IntegerData()` object with a `ZafInteger` object. For more information on data sharing in ZAF, see `ZafDataManager`.

The return value for `IntegerData()` is a pointer to the `IntegerData()` object associated with the `ZafInteger` object. The return value for `SetIntegerData()` is normally `ZAF_ERROR_NONE`. The following code shows the proper use of these functions:

```
// Get the data.
const ZafIntegerData *data = integer1->IntegerData();
...
// Add the integer data.
ZafIntegerData *newData = new ZafIntegerData(100);
integer1->SetIntegerData(newData);
```


ZafIntegerData

Clear	FormattedText	SetInteger
long	Value	operator --
operator ++	operator =	operator +=
operator -=	operator *=	operator /=
operator %=		

ZafIntegerData objects can be used to store and manipulate 32-bit integers.

ZafIntegerData combines number encapsulation with data and object notification from ZafData. It is most often used in conjunction with the ZafInteger user interface object but may be used as a stand-alone object if desired.

ZafIntegerData supports the use of printf-style formatting and parsing arguments during string operations. Refer to standard library documentation for detailed information on printf functions and conversion characters.

Declaration `#include <z_int.hpp>`

Inheritance `ZafIntegerData : ZafFormatData : ZafData :
(ZafNotification, ZafElement)`

Constructors ZafIntegerData constructors allocate space for the integer data and initialize the member variables associated with a ZafIntegerData object.

The default values set by ZafIntegerData follow, if they are overridden from those set by base class constructors:

Member Initializations

ZafIntegerData

Value() (varies by constructor)

ZafElement

ClassID() ID_ZAF_INTEGER_DATA
ClassName() "ZafIntegerData"

ZafIntegerData(void);

The basic constructor allocates a ZafIntegerData instance and initializes its value to 0.

```
ZafIntegerData(long value);
```

This constructor allocates a `ZafIntegerData` instance and initialize its contents to *value*.

```
ZafIntegerData(const ZafIChar *string, const ZafIChar  
    *format = ZAF_NULLP(ZafIChar));
```

This constructor allocates a `ZafIntegerData` instance and initializes its value to the numeric equivalent of *string*. The conversion uses the printf-style specifier *format* to interpret the string. If *format* is null `ZafIntegerData` uses its locale-specific default format.

```
ZafIntegerData(const ZafIntegerData &copy);
```

This constructor is the copy constructor. It allocates a new `ZafIntegerData` instance and copies all member data from *copy*.

```
ZafIntegerData(const ZafIChar *name, ZafDataPersistence  
    &persist);
```

This constructor is the persistent constructor. It allocates a new `ZafIntegerData` instance and reads most member data from directory *name* in the persistent data file referred to by *persist*. The `StringID()` of the new data is *name*.

```
// Sample ZafIntegerData creation techniques  
long value = 100;  
ZafIntegerData integer1(value);  
ZafIntegerData copyInteger = integer1;  
ZafIntegerData zeroInteger;
```

Destructor

```
virtual ~ZafIntegerData(void);
```

This virtual destructor is used to free the memory associated with an instantiated `ZafIntegerData` object. Unless `StaticData()` is set, a `ZafIntegerData` will be destroyed automatically when all `ZafInteger` objects that refers to it are destroyed.

Members

```
virtual void Clear(void);
```

`Clear()` is a virtual function that sets the value of a `ZafIntegerData` object to zero.

```
virtual int FormattedText(ZafIChar *buffer, int
    maxLength, const ZafIChar *format = 0) const;
```

`FormattedText()` fills *buffer* with a string representation of the `ZafIntegerData` using the printf-style specifier *format* to build the string. A locale-specific default format is used if *format* is not included. Buffer contents will be truncated if they exceed *maxLength*. `FormattedText()` returns the integer value it receives from its call to `sprintf()`.

```
// Show results of FormattedText().
ZafIChar buffer[256];
```

```
ZafIntegerData myint(123);
myint.FormattedText(buffer, sizeof(buffer));
printf("decimal int - %s\n", buffer);
myint.FormattedText(buffer, sizeof(buffer), "%X");
printf("hexadecimal - %s\n", buffer);
```

```
=====
decimal int - 123
hexadecimal - 7B
```

```
virtual ZafError SetInteger(long value);
virtual ZafError SetInteger(const ZafIChar *buffer, const
    ZafIChar *format);
virtual ZafError SetInteger(const ZafIntegerData
    &integer);
```

`SetInteger()` functions set the value of the `ZafIntegerData` object from various numeric input types, another `ZafIntegerData`, or an interpreted string. Refer to `FormattedText` for more information on `ZafIntegerData`/string conversions. Overloaded operator `=` offers similar functionality to `SetInteger()` and is more commonly used.

```
long Value(void) const;
operator long();
```

`Value()` returns the value of a `ZafIntegerData` as a long. The convenience operator `long()`, which returns `Value()`, is more commonly used.

```
// Perform numerical operations on a ZafIntegerData
ZafIntegerData myint(123);
myint = myint.Value() + 15;
printf("integer - %u\n", myint);
```



```
myint = myint - 21;
printf("integer - %u\n", myint);
```

```
=====
integer - 138
integer - 117
```

```
ZafIntegerData operator--(void);
ZafIntegerData operator--(int);
```

These pre- and post-operators decrement the ZafIntegerData object's value by 1.

```
ZafIntegerData operator++(void);
ZafIntegerData operator++(int);
```

These pre- and post-operators increment the ZafIntegerData object's value by 1.

```
ZafIntegerData &operator=(long value);
```

This operator assigns the ZafIntegerData object's value to the input *value*.

```
ZafIntegerData &operator+=(long value);
```

This operator increments the ZafIntegerData object's value by the input *value*.

```
ZafIntegerData &operator-=(long value);
```

This operator decrements the ZafIntegerData object's value by the input *value*.

```
ZafIntegerData &operator*=(long value);
```

This operator multiplies the ZafIntegerData object's value by the input *value* and uses the resulting product to set the ZafIntegerData object's value.

```
ZafIntegerData &operator/=(long value);
```

This operator divides the ZafIntegerData object's value by the input *value* and uses the resulting quotient to set the ZafIntegerData object's value.

```
ZafIntegerData &operator%=(long value);
```

This operator divides the ZafIntegerData object's value by the input *value* and uses the resulting remainder to set the ZafIntegerData object's value.

ZafMouse

Event	ImageType	Poll
-------	-----------	------

ZafMouse is the class that defines mouse input device support. Some similar input devices that behave like mice are automatically supported by ZafMouse (such as trackballs and trackpads). Other similar input devices (such as touch screens) may derive from ZafMouse to get the same basic functionality, and add specific functionality in the superclass.

Declaration

#include <z_mouse2.hpp>

Inheritance

ZafMouse : ZafDevice : ZafElement

Constructors

All ZafMouse constructors initialize the member variables associated with an instantiated ZafMouse object. Default values set by the ZafMouse follow, as well as base class values when overridden by ZafMouse.

Member Initializations

ZafMouse

ImageType()

user-supplied parameter

ZafDevice

DeviceType()

E_MOUSE

ZafElement

ClassID()

ID_ZAF_MOUSE

ClassName()

"ZafMouse"

ZafMouse(ZafDeviceState state = D_ON, ZafDeviceImage
imageType = DM_WAIT);

This constructor is used to instantiate a ZafMouse object to be added to a ZafEventManager object. The *state* parameter specifies the initial state of the device, and the *imageType* parameter specifies the initial image type displayed by the device (see ImageType() for more information).

ZafMouse(const ZafMouse ©);

The copy constructor is used in conjunction with the overloaded Duplicate() function. It accepts another ZafMouse object and copies the object's information. An example of how to create a ZafMouse object follows:

```
// Instantiate the input devices.
ZafEventManager *eventManager = new ZafEventManager;
eventManager->Add(new ZafKeyboard);
eventManager->Add(new ZafMouse);
eventManager->Add(new ZafCursor);
```

Destructor

```
virtual ~ZafMouse(void);
```

The destructor is used to free the memory associated with a ZafMouse object. It chains to the ZafDevice and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafMouse object, since it is automatically destroyed when the event manager is destroyed. For more information on device object deletion, see ZafEventManager::~ZafEventManager().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function handles all events sent to the ZafMouse object. ZafMouse handles all the DM_* messages discussed in ImageType(), as well as D_STATE, which causes a device to return its state.

```
ZafDeviceImage ImageType(void) const;
virtual ZafDeviceImage SetImageType(ZafDeviceImage
    imageType);
```

The ImageType() function returns a constant that indicates the mouse's current image type. For example, ImageType() returns DM_VIEW for a mouse device that currently displays the default pointer image. SetImageType() may be

called to change a mouse's image type. The different image types supported by ZafMouse are as follows:

ImageType()	Description
DM_BOTTOM_LEFT_CORNER	causes the mouse to show the bottom-left corner image
DM_BOTTOM_RIGHT_CORNER	causes the mouse to show the bottom-right corner image
DM_BOTTOM_SIDE	causes the mouse to show the bottom side image
DM_CANCEL	causes the mouse to show the cancel image
DM_CROSS_HAIRS	causes the mouse to show the cross-hairs image
DM_EDIT	causes the mouse to show the I-bar image
DM_LEFT_SIDE	causes the mouse to show the left side image
DM_MOVE	causes the mouse to show the move image
DM_RIGHT_SIDE	causes the mouse to show the right side image
DM_SELECT	causes the mouse to show the selection image
DM_TOP_LEFT_CORNER	causes the mouse to show the top-left corner image
DM_TOP_RIGHT_CORNER	causes the mouse to show the top-right corner image
DM_TOP_SIDE	causes the mouse to show the top side image
DM_VIEW	causes the mouse to show the default pointer image
DM_WAIT	causes the mouse to show the wait image

```
virtual void Poll(void);
```

In some environments, the Poll() function checks the mouse device for any input events and posts them on the event manager's queue, if the ZafMouse's state is not D_OFF. In other environments where mouse events are handled automatically by the native environment's event queue, the Poll() event simply blocks mouse events from coming through ZAF's event manager's queue if the ZafMouse's state is D_OFF.

ZafNotification

AddNotification	ClearNotifications	NotifyCount
SubtractNotification	Update	UpdateData
UpdateObjects		

ZafNotification serves as the base class for all objects that add notification to their modes of operation. ZAF defines a whole series of data objects that may automatically notify their window object counter-part when their data has changed. These objects include dates, times, bitmaps, languages, as well as many other data types. For example:

- ZafDateData may notify ZafDate
- ZafTimeData may notify ZafTime
- ZafBitmapData may notify ZafButton
- ZafIconData may notify ZafIcon
- ZafStringData may notify ZafString or ZafText
- ZafScrollData may notify ZafScrollBar

ZafNotification is considered an advanced ZAF class. Therefore, much of the remaining information presented in this chapter is considered advanced material. Readers who don’t need a detailed understanding of the underpinnings of the ZafNotification class may elect to skip through those sections that seem too advanced.

In particular, there are only four functions that most users will need to use. These are Update(), SetUpdate(), UpdateData() and UpdateObjects(). These functions are used to directly manipulate the data associated with a set of window objects. Each of these sections is self-contained, and may be referred to without fully understanding the class’s advanced features.

Declaration	#include <z_notify.hpp>
Inheritance	Root class
Constructor	The ZafNotification class constructor is protected. Thus, it can only be called from a derived class’s constructor such as ZafData. ZafNotification sets the following default values:

Member Initializations

ZafNotification

Member Initializations

Update()	ZAF_UPDATE_ALL
NotifyCount()	0

ZafNotification(void);

The constructor initializes a dynamic array of notification objects that will be called whenever the UpdateData() or UpdateObjects() functions is called. By default, no objects are inserted into the notification array, signified by a notification count of 0. The notification array is modified by calls to AddNotification(), ClearNotification(), or SubtractNotification().

The default type of notification performed by the notify class, ZAF_UPDATE_ALL, causes both changes to the data and to the associated notification object to be simultaneously updated. See Update() for more information.

Destructor

virtual ~**ZafNotification**(void);

This virtual destructor is used to free the memory associated with an instantiated ZafNotification object. The ZafNotification portion of the destructor deletes the internal notification array allocated by calls to AddNotification().

A ZafNotification pointer should never be directly deleted! As stated earlier in this section, the ZafNotification class is generally associated with a derived object through multiple inheritance. Consider the class declaration of ZafData:

```
class ZafExportClass ZafData : public ZafElement, public
    ZafNotification
```

An explicit call to the ZafNotification destructor would not have the desired result of destroying the ZafData object, but would result in only the partial deletion of the object. To determine the type of destructor that you should call, refer to the reference chapter on the particular object you created.

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
ZafNotifyObject *AddNotification(ZafNotifyObject
    *object, ZafUpdateFunction function =
```

```

    ZAF_NULLF(ZafUpdateFunction), ZafUpdateType type =
    ZAF_UPDATE_ALL);
void ClearNotifications(void);
ZafNotifyObject *SubtractNotification(ZafNotifyObject
    *object, ZafUpdateFunction function =
    ZAF_NULLF(ZafUpdateFunction));

```

These functions clear or modify the notification objects associated with the derived ZafNotification class. ClearNotifications() deletes the entire internal notification array. AddNotification() takes the following arguments:

- the *object* to be associated with the data instance
- the *function* to be called whenever the data component changes
- the *type* of notifications the object should receive

ZafUpdateType's include:

ZafUpdateType()	Definition
ZAF_UPDATE_NONE	Do not update either the data or window object portions of the notification. Using this value causes all notification to be disabled.
ZAF_UPDATE_DATA	Only update the data component of the notification. If this value is used exclusive of ZAF_UPDATE_OBJECT, then changes to the data component will not be reflected by the associated window objects.
ZAF_UPDATE_OBJECT	Only update the window object portion of the notification. If this value is used exclusive of ZAF_UPDATE_DATA, then changes to window objects will not automatically cause the data component to be updated.
ZAF_UPDATE_ALL	Update both the data and window object portions of the notification. Using this value causes all notification to be enabled. This is the default value.

Note that ZAF_UPDATE_DATA and ZAF_UPDATE_OBJECT can be used simultaneously by using the “or” operator: (ZAF_UPDATE_DATA | ZAF_UPDATE_OBJECT). The ZafUpdateFunction argument must be in the following form:

```
ZafError (*ZafUpdateFunction)(ZafNotifyObject *, ZafUpdateType);
```

- *ZafNotifyObject **: The window object that will be called whenever the data component changes.

- *ZafUpdateType*: The type of operation being performed on the object, as defined above.

The `SubtractNotification()` arguments *object* and *function* have the same meaning as with `AddNotification()`.

These functions return a `ZafError` value. Normally, this value will be `ZAF_ERROR_NONE` (0), but may be any of the error values defined in the header file `z_env.hpp`. For a description of these values see `ZafData::Error()` and `ZafWindowObject::Error()`.

The following examples include data notification.

```
static ZafError Update(ZafString *string, ZafUpdateType type)
{ if (type == ZAF_UPDATE_OBJECT) return string->OSSetText();
  else if (type == ZAF_UPDATE_DATA)
    return string->OSGetText();
  else return (ZAF_ERROR_INVALID); }
```

```
static ZafError Update(ZafInteger *integer, ZafUpdateType type)
{ if (type == ZAF_UPDATE_OBJECT)
    return integer->OSSetInteger();
  else if (type == ZAF_UPDATE_DATA)
    return integer->OSGetInteger();
  else return (ZAF_ERROR_INVALID); }
```

Data or object notification is very useful in run-time applications. Typically, application frameworks offer window object technology and the ability to manipulate the screen presentation of an object, but few packages have automatic notification of visual presentations connected with the data objects. Whenever a ZAF window object is created with a `ZafData` derived object, the object registers itself with the data component. The following example, from `ZafInteger`, shows how this connection is established.

```
ZafError ZafInteger::SetIntegerData(ZafIntegerData
    *newIntegerData)
{
    // Remove old data notification.
    if (integerData)
        integerData->SubtractNotification(this);

    // Reset the integer data.
    ...

    // Re-add new data notification.
    if (integerData)
        integerData->AddNotification(this);
```

```

    return (OSSetInteger());
}

```

Once the data/object connection is made, the window object receives notification anytime the data object's value changes. In the code sample above, `ZafInteger::integerData` is the data component. Whenever the integer data changes, update calls are automatically sent to the `ZafInteger` object, allowing the object to refresh the information it presents to the screen.

If the data object is attached as a static data element to several window objects, each object registers itself for data notification using the `AddNotification()` function. This allows each window object to receive notification whenever the data component of the object changes.

When the window object is destroyed, it calls `SubtractNotification()` to remove itself from the data object's notification list. An example is the `ZafInteger` destructor:

```

ZafInteger::~ZafInteger(void)
{
    // Remove the data notification.
    if (integerData)
        integerData->SubtractNotification(this);
    ...
}

```

The return value for `AddNotification()` is the notification argument if the operation is successful. Otherwise null is returned. The return value for `SubtractNotification()` is the next notification object in the data object's notification list. If no notification objects exist after the specified object, null is returned.

```
int NotifyCount(void) const;
```

`NotifyCount()` returns the number of elements that have been added to the internal notification array.

```

ZafUpdateType Update(void) const;
ZafUpdateType Update(ZafNotifyObject *object) const;
ZafUpdateType SetUpdate(ZafUpdateType updateType);
ZafUpdateType SetUpdate(ZafNotifyObject *object,
    ZafUpdateType updateType);

```

These functions are used to set the notification state of an object, or to reset the particular notification constraints associated with a notification object. The possible update types are:

- ZAF_UPDATE_NONE
- ZAF_UPDATE_DATA
- ZAF_UPDATE_OBJECT
- ZAF_UPDATE_ALL

As an example, if the notification is ZAF_UPDATE_ALL, then all modifications to the data object are reflected in the associated notification objects, and all modifications to the window objects are reflected in the associated data object. The following code associates a ZafIntegerData object with two ZafString classes.

```
// Associate one data class with two window objects.
ZafIntegerData *intData = new ZafIntegerData(200);
*window1 + new ZafInteger(0, 0, 20, intData, true);
*window2 + new ZafInteger(0, 0, 20, intData, true);
```

When each ZafInteger object is created, it registers itself with the ZafIntegerData instance as a notification object. Later, if the contents of the intData object are changed, the visual representation of the two window objects will automatically be updated.

```
// Change the contents of the intData object.
*intData += 100;
```

If the notification is set to ZAF_UPDATE_NONE, then the visual representation of the object will not automatically be updated on the screen.

Finally, if you set the notification to ZAF_UPDATE_NONE and then back to ZAF_UPDATE_ALL, you must call UpdateObjects() to refresh the data notification objects.

```
// Remove the notification while we do some calculations.
intData->SetUpdate(ZAF_UPDATE_NONE);

// Do some calculations on intData.
*intData += 10;
if (*intData < 100)
    *intData *= 4;
else if (moonOverParador)
    *data++;
```

```
// Reset the notification and update all associated objects.
intData->SetUpdate(ZAF_UPDATE_ALL);
intData->UpdateObjects();
```

If you do not call `UpdateObjects()`, the contents of the window object will not be updated until the object needs to be re-displayed by the application.

```
ZafError UpdateData(ZafNotifyObject *objectSource =
    ZAF_NULLP(ZafNotifyObject));
```

`UpdateData()` forces any specified notification objects to immediately update their data information. A null argument means update all the data components.

When non-null, the *objectSource* parameter causes a `ZAF_UPDATE_DATA` request to be sent to all objects in the notification array that match *objectSource* and that have set their notification either to `ZAF_UPDATE_DATA` or `ZAF_UPDATE_ALL`. Be careful using the `UpdateData()` function, since it is used to reset the data, not the window object. If a matching object is not specified, the data will update according to all the objects in the notification array—an operation that may have undesirable results.

Usually, `ZAF_ERROR_NONE` is returned.

```
ZafError UpdateObjects(ZafNotifyObject *objectMatch =
    ZAF_NULLP(ZafNotifyObject));
```

`UpdateObjects()` forces any specified notification objects to immediately update visually. A null argument means update all the window object components.

When non-null, the *objectMatch* parameter causes a `ZAF_UPDATE_OBJECT` request to be sent to all objects in the notification array that match *objectMatch* and that have their notification set to either `ZAF_UPDATE_OBJECT` or `ZAF_UPDATE_ALL`. Remember, the notification array contains a set of objects that are to be notified whenever data has changed. Specifying an argument will only cause the specified object to be notified of the data change.

Generally, you will only want to use this function when notification has been disabled, then later re-enabled, or when you want the contents of any associated window objects to immediately update visually. The following example shows how two string objects can be updated when a `ZafStringData` object is modified.

```
// Associate one data object with two string objects.
```

```
ZafStringData *strData = new ZafStringData("Zinc Software",
    200);
window1->Add(new ZafString(0, 0, 20, strData, true));
window2->Add(new ZafString(0, 0, 20, strData, true));
...
// Remove the notification while we do some data work.
strData->SetUpdate(ZAF_UPDATE_NONE);
// Do some work with strData.
if (userWantsFax)
    *strData += ", Fax: 785-8996";
else if (userWantsPhone)
    *strData += ", Phone: 785-8900";
*strData += ", Pleasant Grove, UT";
// Reset the notification and update all associated objects.
strData->SetUpdate(ZAF_UPDATE_OBJECT);
strData->UpdateObjects();
```

Usually, ZAF_ERROR_NONE is returned.

ZafProgressBar

Current	Decrement	Delta
Increment	Maximum	Minimum
ProgressData	ProgressStyle	ProgressType
Step	TextStyle	

The ZafProgressBar object is a static informational object generally used to indicate the progress of some operation. The progress may be indicated in percentages or some other integral unit of measurement. The ZafProgressBar object allows no user interaction, and is intended for use only to display information.

Declaration `#include <z_prgrss.hpp>`

Inheritance `ZafProgressBar : ZafWindowObject : ZafElement`

Constructors All ZafProgressBar constructors initialize the member variables associated with an instantiated ZafProgressBar object. The default values set by the ZafProgressBar and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafProgressBar. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafProgressBar

ProgressData()	null
ProgressStyle()	ZAF_PROGRESS_NATIVE
ProgressType()	ZAF_PROGRESS_HORIZONTAL
TextStyle()	ZAF_PROGRESS_TEXT_VALUE

ZafWindowObject

AcceptDrop()	false [†]
CopyDraggable()	false [†]
Disabled()	true [†]
Focus()	false [†]
HelpContext()	null [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
Noncurrent()	true [†]
OSDraw()	false
ParentDrawBorder()	false [†]

Member Initializations

ParentDrawFocus()	false [†]
ParentPalette()	false [†]
Selected()	false [†]
UserFunction()	null [†]

ZafElement

ClassID()	ID_ZAF_PROGRESS_BAR
ClassName()	"ZafProgressBar"

```
ZafProgressBar(int left, int top, int width, int height,  
                ZafScrollData *scrollData);
```

This constructor is useful in straight-code situations. The *left* and *top* parameters specify the position where the left and top of the object will be placed on its parent. The *width* and *height* parameters specify the width and height of the object. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. The *scrollData* parameter specifies the `ProgressData()` object. If the *scrollData* parameter is null, this constructor will create a `ProgressData()` object automatically, with the `ZafScrollData` class default information.

```
ZafProgressBar(const ZafProgressBar &copy);
```

The copy constructor is used in conjunction with the overloaded `Duplicate()` function. It accepts another `ZafProgressBar` object and copies the object's information. If *copy*'s data objects are `StaticData()` then the new `ZafProgressBar` object simply points to the original data objects, otherwise copies are made for the new `ZafProgressBar` object. This behavior allows a programmer to use static data for more than one `ZafProgressBar` object.

```
ZafProgressBar(const ZafIChar *name,  
                ZafObjectPersistence &persist);
```

The final constructor is used for persistence. Refer to `ZafWindow` for more information, since most persistence is done at the `ZafWindow` level.

The following shows how to create a `ZafProgressBar`:

```
// Create a sample window with a progress bar.  
ZafWindow *window1 = new ZafWindow(0, 0, 40, 4);  
// Create the progress data for 0-100% with 20% intervals.
```

```

ZafScrollData *progressData = new ZafScrollData(0, 100, 0, 20);
// Create the progress bar to display percentages.
ZafProgressBar *progressBar = new ZafProgressBar(5, 1, 30, 1,
    progressData);
progressBar->SetTextStyle(ZAF_PROGRESS_TEXT_PERCENT);
// Add the progress bar to the window.
window1->Add(progressBar);

```

Destructor

```
virtual ~ZafProgressBar(void);
```

This destructor is used to free the memory associated with a ZafProgressBar object, including ProgressData(), if it is Destroyable(). It chains to the ZafWindowObject and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafProgressBar object, since it is automatically destroyed when it's parent window is destroyed. For more information on child object deletion, see ZafWindow::~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```

long Current(void) const;
long SetCurrent(long current);

```

The current value of the progress bar appears as a numerical value in the middle of the progress bar when the progress bar supports textual information. Current() returns the current value of the progress bar by calling ProgressData()->Current(). The current value of the progress bar may be set by calling SetCurrent(), which simply calls ProgressData()->SetCurrent().

```
long Decrement(long value);
```

Decrement() decrements the progress bar by the amount Delta() by calling ProgressData()->Decrement(). The current value of the progress bar is returned.

```

long Delta(void) const;
long SetDelta(long current);

```

The delta value of the progress bar is the amount that the progress bar increments or decrements each time it is updated. Delta() returns the delta value of the progress bar by calling ProgressData()->Delta(). The delta value of the

progress bar may be set by calling `SetDelta()`, which simply calls `ProgressData()->SetDelta()`.

```
long Increment(long value);
```

`Increment()` increments the progress bar by the amount `Delta()` by calling `ProgressData()->Increment()`. The current value of the progress bar is returned.

```
long Maximum(void) const;  
long SetMaximum(long maximum);
```

The maximum value of the progress bar is the value at which the progress bar quits updating, and the operation whose progress it is displaying is considered done. `Maximum()` returns the maximum value of the progress bar by calling `ProgressData()->Maximum()`. The maximum value of the progress bar may be set by calling `SetMaximum()`, which simply calls `ProgressData()->SetMaximum()`.

```
long Minimum(void) const;  
long SetMinimum(long minimum);
```

The minimum value of the progress bar is the value at which the progress bar begins updating, and the operation whose progress it is displaying is considered to be starting. `Minimum()` returns the minimum value of the progress bar by calling `ProgressData()->Minimum()`. The minimum value of the progress bar may be set by calling `SetMinimum()`, which simply calls `ProgressData()->SetMinimum()`.

```
ZafScrollData *ProgressData(void) const;  
bool SetProgressData(ZafScrollData *progress);
```

`ProgressData()` specifies the `ZafScrollData` object that contains progress parameters.

The `ProgressData()` object is the piece of the `ZafProgressBar` object where the actual data is stored. `ProgressData()` stores the minimum, maximum, current, delta and showing values for the progress bar (see `ZafScrollData` for more information). The `ProgressData()` piece may be shared among several `ZafProgressBar` objects, or it may belong to a single `ZafProgressBar` object. If shared among several `ZafProgressBar` objects, all the associated `ZafProgressBar` objects will be updated when the `ProgressData()` piece changes. `SetProgressData()` may be used to associate a `ProgressData()` object with a `ZafProgressBar` object. For more information on data sharing in ZAF, see `ZafDataManager`.

The return value for `ProgressData()` is a pointer to the `ProgressData()` object associated with the `ZafProgressBar` object. The return value for `SetProgressData()` is normally `ZAF_ERROR_NONE`.

```
ZafProgressStyle ProgressStyle(void) const;
ZafProgressStyle SetProgressStyle(ZafProgressStyle
    progressStyle);
```

A `ZafProgressBar` object may be one of four different styles, according to the `ProgressStyle()` attribute. The default value of this attribute is `ZAF_PROGRESS_NATIVE`, but the user may call `SetProgressStyle()` to change it. The possible values of this attribute are:

ProgressStyle()	Description
<code>ZAF_PROGRESS_NATIVE</code>	Appears the same as a progress bar on the native environment
<code>ZAF_PROGRESS_INDENTED</code>	Appears bevelled into the plane of its parent
<code>ZAF_PROGRESS_FLAT</code>	Appears at the same level as the plane of its parent
<code>ZAF_PROGRESS_RAISED</code>	Appears raised above the plane of its parent

```
ZafProgressType ProgressType(void) const;
ZafProgressType SetProgressType(ZafProgressType
    progressType);
```

A `ZafProgressBar` object may be oriented either horizontally or vertical, according to the `ProgressType()` attribute. The default value of this attribute is `ZAF_PROGRESS_HORIZONTAL`, but the user may call `SetProgressType()` to change it to `ZAF_PROGRESS_VERTICAL`.

```
long Step(void) const;
long SetStep(long step);
```

The step value of the progress bar is the same as the delta value. See `Delta()` for more information.

```
ZafProgressTextStyle TextStyle(void) const;
```

```
ZafProgressTextStyle SetTextStyle(ZafProgressTextStyle  
style);
```

A ZafProgressBar object may display its current value numerically at the same time as it displays a graphical bar, or it may only display its current value as a graphical bar, according to the TextStyle() attribute. The default value of this attribute is ZAF_PROGRESS_TEXT_VALUE, but the user may call SetTextStyle() to change it. The possible values of this attribute are:

TextStyle()	Description
ZAF_PROGRESS_TEXT_NONE	Does not display the current value numerically
ZAF_PROGRESS_TEXT_VALUE	Displays the current value numerically
ZAF_PROGRESS_TEXT_PERCENT	Displays the current value as a percentage

ZafPrompt

AutoSize	HotKeyChar	HotKeyIndex
SetHotKey	HzJustify	StringData
TransparentBackground	VtJustify	

The ZafPrompt object is a single-line static text object generally used as a label for an adjacent field. The ZafPrompt object allows almost no user interaction, and is intended for use only to display information.

Declaration `#include <z_prompt.hpp>`

Inheritance `ZafPrompt : ZafWindowObject : ZafElement`

Constructors All ZafPrompt constructors initialize the member variables associated with an instantiated ZafPrompt object. The default values set by the ZafPrompt and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafPrompt. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafPrompt

AutoSize()	true
HotKeyChar()	0
HotKeyIndex()	-1
HzJustify()	ZAF_HZ_LEFT
StringData()	null
TransparentBackground()	false
VtJustify()	ZAF_VT_TOP

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	true
Changed()	false [†]
CopyDraggable()	false [†]
Focus()	false [†]
HelpContext()	null [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
Noncurrent()	true [†]
RegionType()	ZAF_INSIDE_REGION [†]

Member Initializations

UserFunction()	null [†]
ZafElement	
ClassID()	ID_ZAF_PROMPT
ClassName()	"ZafPrompt"

```
ZafPrompt(int left, int top, int width, const ZafIChar  
          *text);
```

This constructor is useful in straight-code situations, particularly if you wish the ZafPrompt object to create, maintain and destroy its own ZafStringData object automatically. The *left* and *top* parameters specify the position where the left and top of the object will be placed on its parent. The *width* parameter specifies the width of the object (height defaults to 1 cell.). All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. The *text* parameter is the string you wish to initially appear in the new ZafPrompt object.

```
ZafPrompt(int left, int top, int width, ZafStringData  
          *stringData = ZAF_NULLP(ZafStringData));
```

This constructor is useful in straight-code situations where you have already created a ZafStringData object. This constructor could be used to maintain data pieces yourself, rather than having the ZafPrompt class create and maintain the data pieces automatically. For example, to maintain a database of ZafStringData objects and tie them into ZafPrompt objects, maintain your own ZafStringData objects and create ZafPrompt objects using your ZafStringData objects by passing them into the stringData parameter of this constructor. For more information on using ZafStringData objects, see the chapter on ZafStringData. The *left*, *top* and *width* parameters are the same as the previous constructor.

```
ZafPrompt(const ZafPrompt &copy);
```

The copy constructor is used in conjunction with the overloaded Duplicate() function. It accepts another ZafPrompt object and copies the object's information. If the *copy*'s data objects are StaticData() then the new ZafPrompt object simply points to the original data objects, otherwise, a copy is made for the new ZafPrompt object. This behavior allows a programmer to share data between ZafPrompt objects.

ZafPrompt(const ZafIChar *name, ZafObjectPersistence &persist);

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

Below is an example illustrating creation techniques for ZafPrompt:

```
// Create a sample window with a prompt and a string.
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);
window1->Add(new ZafPrompt(0, 1, 10, "String:"));
window1->Add(new ZafString(11, 1, 10, "String", 10));
...
// Create a sample window with a prompt and a string.
ZafWindow *window2 = new ZafWindow(10, 10, 40, 10);
ZafStringData *stringData1 = new ZafStringData("String:");
window2->Add(new ZafPrompt(0, 1, 10, stringData1));
window2->Add(new ZafString(11, 1, 10, "String", 10));
```

Destructor

virtual ~ZafPrompt(void);

This destructor is used to free the memory associated with a ZafPrompt object, including StringData() if it is Destroyable(). It chains to the ZafWindowObject and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafPrompt object, since it is automatically destroyed when it's parent window is destroyed. For more information on child object deletion, see ZafWindow::~~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
bool AutoSize(void) const;
virtual bool SetAutoSize(bool setAutoSize);
```

If AutoSize() is true, the prompt will automatically adjust the size of its background area so as to exactly match the size of its displayed text. This is particularly useful if the prompt has a helptip or if the prompt's background color is different from the window background. The original region (passed into the constructor or read from the persistent object file) is preserved internally by ZAF and is used if region computations are required by later programmer interaction with the prompt.

If `AutoSize()` is false, the prompt will display with its original region. The default value of this attribute is true, but the user may call `SetAutoSize()` to change it.

```
ZafIChar HotKeyChar(void) const;  
int HotKeyIndex(void) const;  
virtual ZafIChar SetHotKey(ZafIChar hotKeyChar, int index  
    = -1);
```

A prompt's hot key character (*hotKeyChar*) is the character that when typed with a modifier key (such as <ALT> or <Command>) causes the next object in tabbing order is given focus. The hot key *index* is the zero-based index into the prompt's text that specifies the character to be visually displayed as the hot key character, usually with an underline.

It is important to note that the hot key character does not cause any display modification, and the hot key index does not cause any action to be performed when that character is typed with the modifier key. The default value of `HotKeyChar()` is 0, indicating that there is no hot key character associated with the prompt, and the default value of `HotKeyIndex()` is -1, indicating that no character is to be displayed as the hot key character on the prompt. The user may call `SetHotKey()` to change the `HotKeyChar()` and the `HotKeyIndex()` attributes. The *hotKeyChar* parameter specifies the hot key character, and the *index* parameter specifies the hot key index.

The following code shows how to create a prompt and string pair with a hot key:

```
// Create a prompt and string pair with a hot key.  
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);  
ZafPrompt *prompt1 = new ZafPrompt(0, 1, 10, "Name:");  
prompt1->SetHotKey('N', 0);  
window1->Add(prompt1);  
// The string following the prompt gets focus when the hot  
// key is typed.  
window1->Add(new ZafString(11, 1, 10, "", 10));
```

```
ZafHzJustify HZJustify(void) const;  
virtual ZafHzJustify SetHZJustify(ZafHzJustify  
    hzJustify);
```

`HZJustify()` controls the prompt's horizontal justification, and is `ZAF_HZ_LEFT` by default. The user may call `SetHZJustify()` to change to `ZAF_HZ_RIGHT` or `ZAF_HZ_CENTER`.

```
ZafStringData *StringData(void) const;
virtual ZafError SetStringData(ZafStringData *string);
```

The StringData() object is where the actual data is stored. The StringData() piece may be shared among several ZafPrompt objects, or it may belong to a single ZafPrompt object. If shared among several ZafPrompt objects, all the associated ZafPrompt objects will be updated when the StringData() piece changes. SetStringData() may be used to associate a StringData() object with a ZafPrompt object. For more information on data sharing in ZAF, see ZafDataManager.

The return value for StringData() is a pointer to the StringData() object associated with the ZafPrompt object. The return value for SetStringData() is normally ZAF_ERROR_NONE.

```
bool TransparentBackground(void) const;
virtual bool SetTransparentBackground(bool
    transparentBackground);
```

If TransparentBackground() is true, the prompt does not erase its background area, but only draws its text. This is useful if the region the prompt overlaps should show through. If TransparentBackground() is false, the prompt will erase its background area. The default value of this attribute is false, but the user may call Set TransparentBackground() to change it.

```
ZafVtJustify VtJustify(void) const;
virtual ZafVtJustify SetVtJustify(ZafVtJustify
    vtJustify);
```

VtJustify() controls the prompt's vertical justification, and is ZAF_VT_TOP by default. The user may call SetVtJustify() to change it to ZAF_VT_CENTER or ZAF_VT_BOTTOM.

ZafPullDownItem

Add	Count	Current
Destroy	Event	First
FocusObject	Get	GetObject
Index	Last	menu
Sort	Subtract	operator +
operator -		

ZafPullDownItem objects are the items on a ZafPullDownMenu bar. ZafPullDownItem objects may only be added to a ZafPullDownMenu object, and only ZafPopUpItem objects may be added to a ZafPullDownItem object.

Declaration `#include <z_plldn.hpp>`

Inheritance `ZafPullDownItem : ZafButton : ZafWindowObject :
 ZafElement`

Constructors All ZafPullDownItem constructors initialize the member variables associated with an instantiated ZafPullDownItem object. The default values set by the ZafPullDownItem and its base class constructors follow if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafPullDownItem. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafPullDownItem

<code>menu</code>	<code>ZafPopUpMenu(0, 0)</code>
<code>menu.Destroyable()</code>	<code>false</code>
<code>menu.parent</code>	<code>this</code>
<code>menu.StringID()</code>	<code>"menu"</code>
<code>menu.Temporary()</code>	<code>true</code>

ZafButton

<code>AllowDefault()</code>	<code>false[†]</code>
<code>AllowToggling()</code>	<code>false[†]</code>
<code>AutoRepeatSelection()</code>	<code>false[†]</code>
<code>AutoSize()</code>	<code>false[†]</code>
<code>BitmapData()</code>	<code>ZAF_NULLP(ZafBitmapData)[†]</code>
<code>ButtonType()</code>	<code>ZAF_FLAT_BUTTON[†]</code>
<code>Depressed()</code>	<code>false[†]</code>
<code>Depth()</code>	<code>0[†]</code>

Member Initializations

HzJustify()	ZAF_HZ_LEFT [†]
SelectOnDoubleClick()	false [†]
SelectOnDownClick()	false [†]
SendMessageWhenSelected()	false [†]
Value()	0
VtJustify()	ZAF_VT_CENTER [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	false [†]
CopyDraggable()	false [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
ParentDrawBorder()	false [†]
ParentDrawFocus()	false [†]
ParentPalette()	false [†]
RegionType()	ZAF_INSIDE_REGION [†]
Selected()	false [†]
SupportObject()	false [†]

ZafElement

ClassID()	ID_ZAF_PULL_DOWN_ITEM
ClassName()	"ZafPullDownItem"

ZafPullDownItem(const ZafIChar *text);

This constructor is useful in straight-code situations, particularly if you wish the ZafPullDownItem object to create, maintain and destroy its own ZafStringData object automatically. You simply pass into the *text* parameter the text you wish to appear in the new ZafPullDownItem object.

ZafPullDownItem(ZafStringData *stringData);

This constructor is also useful in straight-code situations, particularly if you have already created a ZafStringData object to be associated with the ZafPullDownItem object. This constructor could be used to maintain string data pieces yourself, rather than having the ZafPullDownItem class create and maintain the string data pieces automatically. For example, to maintain a database of ZafStringData objects and tie them into ZafPullDownItem objects,

maintain your own `ZafStringData` objects and create `ZafPullDownItem` objects using your `ZafStringData` objects. For more information on using `ZafStringData` objects, see `ZafStringData`. The *stringData* parameter specifies the string data object to be associated with the `ZafPullDownItem` object.

```
ZafPullDownItem(const ZafPullDownItem &copy);
```

The copy constructor is used in conjunction with the overloaded `Duplicate()` function. It copies the information from another `ZafPullDownItem` object, *copy*. If the data objects are `StaticData()` then the new `ZafPullDownItem` object points to the original data objects, otherwise a copy is made for the new `ZafPullDownItem` object. This allows a programmer to use static data for more than one `ZafPullDownItem` object.

```
ZafPullDownItem(const ZafIChar *name,  
                 ZafObjectPersistence &persist);
```

The final constructor is used for persistence. The parameters and values of this constructor are deferred to the `ZafWindow` section of this manual, since most persistence is done at the `ZafWindow` level.

Refer to `ZafPullDownMenu` for an example of how to create a `ZafPullDownItem` object:

Destructor

```
virtual ~ZafPullDownItem(void);
```

The destructor is used to free the memory associated with a `ZafPullDownItem` object, including all the data object pieces that are `Destroyable()`. It chains to the `ZafWindowObject` and `ZafElement` destructors.

Generally, the programmer will not directly destroy a `ZafPullDownItem` object, since it is automatically destroyed when its parent menu bar is destroyed. For more information on child object deletion, see `ZafWindow::~~ZafWindow()`.

Members

```
virtual ZafWindowObject *Add(ZafWindowObject *object,  
                              ZafWindowObject *position =  
                              ZAF_NULLP(ZafWindowObject));  
ZafPullDownItem &operator+(ZafWindowObject *object);
```

This function and operator are used for adding `ZafPopUpItem` objects to the `ZafPullDownItem` object. The functionality is provided by the `ZafPopUpMenu` class through the menu member. *object* is a pointer to the `ZafPopUpItem` object to be added to the `ZafPullDownItem` object. *position* specifies the `ZafPopUpItem` object (already in the menu) that object should appear before.

If *position* is null, *object* is added to the end of the menu. See `ZafWindow::Add()` for more information.

```
int Count(void) const;
```

`Count()` returns the number of `ZafPopUpItem` objects in the menu. See `ZafList::Count()` for more information.

```
ZafWindowObject *Current(void) const;
```

`Current()` returns the current `ZafPopUpItem` object in the menu, if there is one. The `Current()` item does not necessarily have focus. Some native environment implementations of menus do not allow the menu to ever have focus. See `ZafList::Current()` for more information.

```
virtual void Destroy(void);
```

`Destroy()` causes all the `ZafPopUpItem` objects in the menu to be destroyed. This is useful if a menu is to be recreated from scratch. See `ZafList::Destroy()` for more information.

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function handles all events that get sent to the `ZafPullDownItem` object, either by processing the events itself, or by passing the event down for base class processing. Refer to `ZafWindowObject` for complete details. Following are events that are handled by `ZafPullDownItem` in addition to those handled by its base classes:

Event	Action
<code>S_ADD_OBJECT</code>	Causes <code>event.windowObject</code> to be added to the menu
<code>S_SUBTRACT_OBJECT</code>	Causes <code>event.windowObject</code> to be subtracted from the menu

```
ZafWindowObject *First(void) const;
```

`First()` returns the first `ZafPopUpItem` object in the menu, if any. See `ZafList::First()` for more information.

```
virtual ZafWindowObject *FocusObject(void) const;
```

FocusObject() returns the ZafPopupMenu object in the menu that has focus, if one does. Some native environment implementations of menus do not allow the menu to ever have focus, so this function will always return null in these cases. See ZafWindow::FocusObject() for more information.

```
ZafWindowObject *Get(int index);
```

Get() returns the ZafPopupMenu object at position *index* in the menu, if there is one. The index parameter is zero-based, meaning the first ZafPopupMenu object is at position 0. See ZafList::Get() for more information.

```
virtual ZafWindowObject *GetObject(ZafNumberID numberID);  
virtual ZafWindowObject *GetObject(const ZafIChar  
    *stringID);
```

GetObject() returns the ZafPopupMenu object with either the numberID of *numberID*, or the stringID of *stringID*, if it is found. See ZafWindowObject::GetObject() for more information.

```
int Index(ZafWindowObject const *element);
```

Index() returns the zero-based index of the ZafPopupMenu object *element* in the menu. If *element* is not found in the menu, Index() returns -1. See ZafList::Index() for more information.

```
ZafWindowObject *Last(void) const;
```

Last() returns the last ZafPopupMenu object in the menu, if there is one. See ZafList::Last() for more information.

```
ZafPopupMenu menu;
```

The menu member, used internally by the ZAF libraries to maintain the ZafPopupMenu objects added to the ZafPullDownItem object, should normally not be accessed by the programmer. See ZafPopupMenu for more information.

```
virtual void Sort(void);
```

Sort() causes the ZafPopUpItem objects in the menu to be sorted according to the function returned by CompareFunction(). See ZafList::CompareFunction() for more information.

```
virtual ZafWindowObject *Subtract(ZafWindowObject  
    *object);  
ZafPullDownItem &operator-(ZafWindowObject *object);
```

This function and operator are used for subtracting ZafPopUpItem objects from the ZafPullDownItem object. The functionality is provided by the ZafPopUpMenu class through the menu member. *object* is a pointer to the ZafPopUpItem object to be subtracted from the ZafPullDownItem object. See ZafWindow::Subtract() for more information.

ZafPullDownMenu

The ZafPullDownMenu object is a menu bar object that supports multiple-level menu hierarchies. As with all other ZAF classes, the ZafPullDownMenu class utilizes the native menu bar API if available, so the look-and-feel is exactly what the end user expects. For example, the menu bar on the Macintosh is at the top of the screen, rather than on the window itself, as with Microsoft Windows.

The menu bar object itself only reserves space for the menu items, so it is useful only when menu items are added to it. Only ZafPullDownItem objects may be added to a ZafPullDownMenu object.

Declaration

```
#include <z_plldn.hpp>
```

Inheritance

```
ZafPullDownMenu : ZafWindow : ((ZafWindowObject :  
    ZafElement), ZafList)
```

Constructors

All ZafPullDownMenu constructors initialize the member variables associated with an instantiated ZafPullDownMenu object. The default values set by the ZafPullDownMenu and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafPullDownMenu. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Moveable()	false [†]
SelectionType()	ZAF_SINGLE_SELECTION [†]
Sizeable()	false [†]
Temporary()	false [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	true [†]
Disabled()	false [†]
Noncurrent()	false [†]
ParentPalette()	false [†]

Member Initializations

RegionType()	ZAF_AVAILABLE_REGION [†]
SupportObject()	true [†]

ZafElement

ClassID()	ID_ZAF_PULL_DOWN_MENU
ClassName()	"ZafPullDownMenu"
SetNumberID()	ZAF_NUMID_PULL_DOWN_MENU
SetStringID()	"ZAF_NUMID_PULL_DOWN_MENU"

ZafPullDownMenu(void);

This constructor is useful in straight-code situations. Since the ZafPullDownMenu object is automatically placed at the appropriate position on the window or on the screen according to the native environment, there are no parameters to this constructor.

ZafPullDownMenu(const ZafPullDownMenu ©);

The copy constructor is used in conjunction with the overloaded Duplicate() function. It accepts another ZafPullDownMenu object and copies the object's information.

**ZafPullDownMenu(const ZafIChar *name,
ZafObjectPersistence &persist);**

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

The following shows how to create a ZafPullDownMenu object:

```
// Create a sample window with a menu bar.
ZafWindow *window1 = new ZafWindow(0, 0, 50, 10);
// Create the menu bar and a file menu.
ZafPullDownMenu *menuBar = new ZafPullDownMenu();
// Create and populate the file menu.
ZafPullDownItem *fileMenu = new ZafPullDownItem("File");
fileMenu->Add(new ZafPopUpItem("New"));
fileMenu->Add(new ZafPopUpItem("Open"));
fileMenu->Add(new ZafPopUpItem("", ZAF_SEPARATOR));
fileMenu->Add(new ZafPopUpItem("", ZAF_EXIT_OPTION));
// Add the file menu to the menu bar.
menuBar->Add(fileMenu);
```



```
// Add the menu bar to the window.  
window1->Add(fileMenu);
```

Destructor

```
virtual ~ZafPullDownMenu(void);
```

The destructor is used to free the memory associated with a ZafPullDownMenu object. It chains to the ZafWindow, ZafList, ZafWindowObject and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafPullDownMenu object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~~ZafWindow().

ZafPopUpItem

Add	Count	Current
Destroy	Event	First
FocusObject	Get	GetObject
Index	ItemType	Last
menu	Sort	Subtract
operator +	operator -	

ZafPopUpItem objects are the items in a menu. A hierarchical sub-menu may be created by simply adding ZafPopUpItem objects to a ZafPopUpItem object. It is important to note that ZafPopUpItem objects may only be added to a ZafPullDownItem object, a ZafPopUpMenu object, or another ZafPopUpItem object.

Declaration `#include <z_popup.hpp>`

Inheritance `ZafPopUpItem : ZafButton : ZafWindowObject : ZafElement`

Constructors All ZafPopUpItem constructors initialize the member variables associated with an instantiated ZafPopUpItem object. The default values set by the ZafPopUpItem and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafPopUpItem. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafPopUpItem

ItemType()	user-supplied parameter
menu	ZafPopUpMenu(0, 0)
menu.parent	this
menu.StringID()	“menu”

ZafButton

AllowDefault()	false [†]
AutoRepeatSelection()	false [†]
AutoSize()	false [†]
BitmapData()	ZAF_NULLP(ZafBitmapData) [†]
ButtonType()	ZAF_FLAT_BUTTON [†]
Depressed()	false [†]
Depth()	0 [†]
HzJustify()	ZAF_HZ_LEFT [†]

Member Initializations

SelectOnDoubleClick()	false [†]
SelectOnDownClick()	false [†]
VtJustify()	ZAF_VT_CENTER [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	false [†]
CopyDraggable()	false [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
ParentDrawBorder()	false [†]
ParentDrawFocus()	false [†]
ParentPalette()	false [†]
RegionType()	ZAF_INSIDE_REGION [†]
SupportObject()	false [†]

ZafElement

ClassID()	ID_ZAF_POP_UP_ITEM
ClassName()	"ZafPopUpItem"

```
ZafPopUpItem(const ZafIChar *text, ZafPopUpItemType  
          itemType = ZAF_NORMAL_ITEM);
```

This constructor is useful in straight-code situations, particularly if you wish the `ZafPopUpItem` object to create, maintain and destroy its own `ZafStringData` object automatically. *text* specifies the text to appear in the new `ZafPopUpItem` object, and *itemType* specifies the type of menu item to be created. See `ItemType()` for more information on menu item types.

```
ZafPopUpItem(ZafStringData *stringData, ZafPopUpItemType  
          itemType = ZAF_NORMAL_ITEM);
```

This constructor is also useful in straight-code situations, particularly if you have already created a `ZafStringData` object to be associated with the `ZafPopUpItem` object. This constructor could be used to maintain string data pieces yourself, rather than having the `ZafPopUpItem` class create and maintain the string data pieces automatically. For example, to maintain a database of `ZafStringData` objects and tie them into `ZafPopUpItem` objects, maintain your own `ZafStringData` objects and create `ZafPopUpItem` objects using your `ZafStringData` objects. For more information on using `ZafStringData` objects, see `ZafStringData`. The *stringData* parameter specifies the string data object to be

associated with the ZafPopUpItem object. Otherwise, this constructor (and parameters) is the same as the first.

```
ZafPopUpItem(const ZafPopUpItem &copy);
```

The copy constructor is used in conjunction with the overloaded Duplicate() function. It copies the information from another ZafPopUpItem object, *copy*. If the data objects are StaticData(), the new ZafPopUpItem object simply points to the original data objects, otherwise a copy is made for the new ZafPopUpItem object. This allows a programmer to use static data for more than one ZafPopUpItem object.

```
ZafPopUpItem(const ZafIChar *name, ZafObjectPersistence
    &persist);
```

The final constructor is used for persistence. The parameters and values of this constructor are deferred to the ZafWindow section of this manual, since most persistence is done at the ZafWindow level.

Refer to ZafPullDownMenu for an example of how to create ZafPopUpItem objects:

Destructor

```
virtual ~ZafPopUpItem(void);
```

The destructor is used to free the memory associated with a ZafPopUpItem object, including all the data object pieces (such as StringData()) that are Destroyable(). It chains to the ZafWindowObject and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafPopUpItem object, since it is automatically destroyed when its parent pop-up menu is destroyed. For more information on child object deletion, see ZafWindow::~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
virtual ZafWindowObject *Add(ZafWindowObject *object,
    ZafWindowObject *position =
    ZAF_NULLP(ZafWindowObject));
ZafPopUpItem &operator+(ZafWindowObject *object);
```

This function and operator are used to create sub-menu items by adding ZafPopUpItem objects to the ZafPopUpItem object. The functionality is provided by the ZafPopUpMenu class through the menu member. *object* is a pointer to the ZafPopUpItem object to be added to the ZafPopUpItem object's sub-menu. *position* specifies which ZafPopUpItem object already in the menu that *object* should appear before. If *position* is null, *object* is added to the end of the menu. See ZafWindow::Add() for more information.

```
int Count(void) const;
```

Count() returns the number of ZafPopUpItem objects in the sub-menu, or 0 if there is no sub-menu. See ZafList::Count() for more information.

```
ZafWindowObject *Current(void) const;
```

Current() returns the current ZafPopUpItem object in the sub-menu, if there is one. The Current() item does not necessarily have focus. Some native environment implementations of menus do not allow the menu to ever have focus. See ZafList::Current() for more information.

```
virtual void Destroy(void);
```

Destroy() causes all the ZafPopUpItem objects in the sub-menu to be destroyed, if there is one. This is useful if a sub-menu is to be recreated from scratch. See ZafList::Destroy() for more information.

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function handles all events that get sent to the ZafPopUpItem object, either by processing the events itself, or by passing the event down for base class processing. Refer to ZafWindowObject for complete details. Following are events that are handled by ZafPopUpItem in addition to those handled by its base classes:

Event	Action
S_ADD_OBJECT	Causes event.windowObject to be added to the sub-menu
S_SUBTRACT_OBJECT	Causes event.windowObject to be subtracted from the sub-menu

```
ZafWindowObject *First(void) const;
```

The First() function returns the first ZafPopUpItem object in the sub-menu, if any. See ZafList::First() for more information.

```
virtual ZafWindowObject *FocusObject(void) const;
```

FocusObject() returns the ZafPopUpItem object in the sub-menu that has focus, if there is one. Some native environment implementations of menus do not allow the menu to ever have focus, so this function will always return null in these cases. See ZafWindow::FocusObject() for more information.

```
ZafWindowObject *Get(int index);
```

Get() returns the ZafPopUpItem object at *position* index in the sub-menu, if there is one. The *index* parameter is zero-based, meaning the first ZafPopUpItem object in the sub-menu is at position 0. See ZafList::Get() for more information.

```
virtual ZafWindowObject *GetObject(ZafNumberID numberID);  
virtual ZafWindowObject *GetObject(const ZafIChar  
    *stringID);
```

GetObject() return the ZafPopUpItem object in the sub-menu with either the *numberID* of *numberID* (if there is one), or the stringID of stringID, if it is found. See ZafWindowObject::GetObject() for more information.

```
int Index(ZafWindowObject const *element);
```

If there is a sub-menu, the Index() function returns the zero-based index of the ZafPopUpItem object *element* in the sub-menu. If there is no sub-menu or *element* is not found in the sub-menu, Index() returns -1. See ZafList::Index() for more information.

```
ZafPopUpItemType ItemType(void) const;
```

```
virtual ZafPopUpItemType SetItemType(ZafPopUpItemType  
    itemType);
```

ItemType() specifies the menu item type of a ZafPopupMenu object. The default value of this attribute is ZAF_NORMAL_ITEM, but the user may call SetItemType() to make changes. Below is a list of possible menu item types:

ZafPopupMenuType	Description
ZAF_NORMAL_ITEM	Creates a normal menu item
ZAF_ABOUT_OPTION	Creates an about menu item (on the Macintosh, placed in the Apple menu)
ZAF_CLOSE_OPTION	Creates an S_CLOSE menu item with the appropriate text, hotkey and SendMessage() information
ZAF_COPY_OPTION	Creates an S_COPY menu item with the appropriate text, hotkey and SendMessage() information
ZAF_CUT_OPTION	Creates an S_CUT menu item with the appropriate text, hotkey and SendMessage() information
ZAF_EXIT_OPTION	Creates an S_EXIT menu item with the appropriate text, hotkey and SendMessage() information
ZAF_MAXIMIZE_OPTION	Creates an S_MAXIMIZE menu item in system menus
ZAF_MINIMIZE_OPTION	Creates an S_MINIMIZE menu item in system menus
ZAF_MOVE_OPTION	Creates an S_MOVE_MODE menu item in system menus
ZAF_PASTE_OPTION	Creates an S_PASTE menu item with the appropriate text, hotkey and SendMessage() information
ZAF_RESTORE_OPTION	Creates an S_RESTORE menu item in system menus
ZAF_SEPARATOR	Creates a Disabled() and Noncurrent() separator line menu item
ZAF_SIZE_OPTION	Creates an S_SIZE_MODE menu item in system menus
ZAF_SWITCH_OPTION	Creates a switch menu item in system menus

```
ZafWindowObject *Last(void) const;
```

Last() returns the last ZafPopUpItem object in the sub-menu, if any. See ZafList::Last() for more information.

ZafPopUpMenu **menu**;

The menu member, used internally by the ZAF libraries to maintain the ZafPopUpItem objects added to the ZafPopUpItem object (forming the sub-menu), should normally not be accessed by the programmer. See ZafPopUpMenu for more information.

virtual void **Sort**(void);

Sort() causes the ZafPopUpItem objects in the sub-menu (if there is one) to be sorted according to the function returned by CompareFunction(). See ZafList::CompareFunction() for more information.

virtual ZafWindowObject ***Subtract**(ZafWindowObject
*object);

ZafPopUpItem &**operator-**(ZafWindowObject *object);

This function and operator are used for subtracting (or removing) ZafPopUpItem objects from the ZafPopUpItem object's sub-menu. The functionality is provided by the ZafPopUpMenu class through the menu member. *object* is a pointer to the ZafPopUpItem object to be subtracted. See ZafWindow::Subtract() for more information.

ZafPopupMenu

The ZafPopupMenu object is a pop-up menu object that supports multiple-level menu hierarchies. As with all other ZAF classes, the ZafPopupMenu class utilizes the native pop-up menu API if available, so the look-and-feel is exactly what the end user expects. The pop-up menu object may be added to a ZafWindowManager object to provide a detached pop-up menu on the screen that disappears when the user selects an item.

If the pop-up menu is not added to a ZafWindowManager object, it is completely managed internally to the ZAF libraries, and should not be accessed by the programmer. Each ZafPullDownItem and ZafPopUpItem object has a ZafPopupMenu member associated with it to handle support for adding ZafPopUpItem objects. Only ZafPopUpItem objects may be added to a ZafPopupMenu object.

Declaration

```
#include <z_popup.hpp>
```

Inheritance

```
ZafPopupMenu : ZafWindow : ((ZafWindowObject :  
    ZafElement), ZafList)
```

Constructors

All ZafPopupMenu constructors initialize the member variables associated with an instantiated ZafPopupMenu object. The default values set by the ZafPopupMenu and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafPopupMenu. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Moveable()	false [†]
Sizeable()	false [†]
Temporary()	true [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	true [†]
Disabled()	false [†]
Noncurrent()	false [†]

Member Initializations

<code>ParentPalette()</code>	<code>false[†]</code>
<code>RegionType()</code>	<code>ZAF_INSIDE_REGION[†]</code>

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_POP_UP_MENU</code>
<code>ClassName()</code>	<code>"ZafPopupMenu"</code>

ZafPopupMenu(int left, int top);

This constructor is useful in straight-code situations. If the pop-up menu is to be added to a `ZafWindowManager` object, the *left* and *top* parameters specify the position where the left and top of the menu will be placed on the screen, respectively. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. Otherwise, the parameters are ignored.

ZafPopupMenu(const ZafPopupMenu ©);

The copy constructor is used in conjunction with the overloaded `Duplicate()` function. It accepts another `ZafPopupMenu` object and copies the object's information.

ZafPopupMenu(const ZafIChar *name, ZafObjectPersistence &persist);

The final constructor is used for persistence. Refer to `ZafWindow` for more information, since most persistence is done at the `ZafWindow` level.

The following shows how to create a `ZafPopupMenu` object:

```
// Create and populate the edit menu.
ZafPullDownItem *editMenu = new ZafPopupMenu(10, 10);
editMenu->Add(new ZafPopUpItem("Undo"));
editMenu->Add(new ZafPopUpItem("", ZAF_SEPARATOR));
editMenu->Add(new ZafPopUpItem("", ZAF_CUT_OPTION));
editMenu->Add(new ZafPopUpItem("", ZAF_COPY_OPTION));
editMenu->Add(new ZafPopUpItem("", ZAF_PASTE_OPTION));
// Add the edit menu to the screen.
windowManager->Add(editMenu);
```

Destructor

`virtual ~ZafPopupMenu(void);`

The destructor is used to free the memory associated with a `ZafPopupMenu` object. It chains to the `ZafWindow`, `ZafList`, `ZafWindowObject` and `ZafElement` destructors.

Generally, the programmer will not directly destroy a `ZafPopupMenu` object, since it is automatically destroyed when its parent menu bar is destroyed. However, if the `ZafPopupMenu` object is added to the window manager, it will have to be destroyed when it is no longer needed, since its `Destroyable()` is false in this case. For more information on child object deletion, see `ZafWindow::~~ZafWindow()`.

ZafReal

Event	RealData
-------	----------

ZafReal is a single-line real number (floating point) object that allows user input through the keyboard. ZafReal is fully internationalized to display and input using any format.

All ZafReal objects refer to data contained in a ZafRealData object (refer to this class for additional essential information).

Declaration `#include <z_real1.hpp>`

Inheritance `ZafReal : ZafString : ZafWindowObject : ZafElement`

Constructors All ZafReal constructors initialize the member variables associated with an instantiated ZafReal object. The default values set by the ZafReal and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafReal. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafReal	
RealData()	null
ZafString	
LowerCase()	false†
Password()	false†
StringData()	null†
UpperCase()	false†
VariableName()	false†
ZafElement	
ClassID()	ID_ZAF_REAL
ClassName()	"ZafReal"

ZafReal(int left, int top, int width, double value);

This constructor is useful in straight-code situations, particularly if you wish the ZafReal object to create, maintain and destroy its own ZafRealData object automatically. *left*, *top* and *width* specify the position and size of the object on

its parent. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. *value* is the value you wish to initially appear in the new ZafReal object.

```
ZafReal(int left, int top, int width, ZafRealData  
        *realData = ZAF_NULLP(ZafRealData));
```

This constructor is useful in straight-code situations where a ZafRealData object has already been created. This constructor could be used when manually maintaining a *realData* object, rather than having the ZafReal class create and maintain the data object automatically. For more information on using ZafRealData objects, see the chapter on ZafRealData. The *left*, *top* and *width* parameters are the same as the previous constructor.

```
ZafReal(const ZafReal &copy);
```

The copy constructor calls the overloaded Duplicate() to create a new ZafReal object and initialize its data from *copy*. If the original data objects are StaticData() then the new ZafReal object simply points to the original data, otherwise StaticData() copies are made.

```
ZafReal(const ZafIChar *name, ZafObjectPersistence  
        &persist);
```

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

Sample ZafReal creation techniques follow:

```
// Create a sample window with real number objects.  
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);  
  
// Create real numbers and pass in the values directly.  
window1->Add(new ZafReal(0, 1, 25, 3.1415927));  
window1->Add(new ZafReal(0, 2, 25, 360.0));  
...  
  
// Create a sample window with real number objects.  
ZafWindow *window2 = new ZafWindow(10, 10, 40, 10);  
  
// Create real number data objects.  
ZafRealData *realData1 = new ZafRealData(3.1415927);  
ZafRealData *realData2 = new ZafRealData(360.0);  
  
// Create real numbers that use the data previously created.
```

```

window2->Add(new ZafReal(0, 1, 25, realData1));
window2->Add(new ZafReal(0, 2, 25, realData2));

```

Destructor

```
virtual ~ZafReal(void);
```

The destructor is used to free the memory associated with a ZafReal object, including all data objects that are Destroyable(). It chains to the ZafString, ZafWindowObject, and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafReal object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~~ZafWindow().

Members

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function receives all events that get sent to the ZafReal object and either handles them or passes them to ZafString, its immediate base class. See ZafWindowObject for more information.

ZafReal specifically handles the following events:

Event	Description
S_COPY_DATA	causes the object to copy event.windowObject's RealData() if event.windowObject is a ZafReal object
S_SET_DATA	causes the object to create a new RealData() object, then copy into it event.windowObject's RealData() if event.windowObject is non-null and is a ZafReal object

```
ZafRealData *RealData(void) const;
```

```
virtual ZafError SetRealData(ZafRealData *number);
```

*RealData() contains the actual information used by ZafReal. The RealData() object may be used by one or more ZafReal objects, or other objects. If shared, all associated ZafReal objects will be notified when the RealData() changes. For more information on data sharing in ZAF, see ZafDataManager.

RealData() returns a pointer to the RealData() object associated with the ZafReal object. The return value for SetRealData() is normally ZAF_ERROR_NONE. See the constructor code snippet for an example using ZafRealData objects with ZafReal.

ZafRealData

Clear	double	FormattedText
SetReal	Value	operator ++
operator --	operator =	operator +=
operator -=	operator *=	opeartor /=
operator %=		

ZafRealData objects can be used to store and manipulate 64-bit doubles.

ZafRealData combines number encapsulation with data and object notification from ZafData. ZafRealData objects are normally used as the data portion of ZafReal user interface objects but they may also be used independently.

ZafRealData supports the use of printf-style formatting and parsing arguments during string operations. Refer to standard library documentation for detailed information on printf functions and conversion characters.

Declaration	<code>#include <z_real.hpp></code>
Inheritance	<code>ZafRealData : ZafFormatData : ZafData : (ZafNotification, ZafElement)</code>
Constructors	<p>ZafRealData constructors allocate space to hold the instance data, and initialize the member variables associated with a new ZafRealData object.</p> <p>The default values set by ZafRealData follow, if they are overridden from those set by base class constructors:</p>

Member Initializations

ZafRealData	
<code>Value()</code>	<code>(varies with constructor)</code>
ZafElement	
<code>ClassID()</code>	<code>ID_ZAF_REAL_DATA</code>
<code>ClassName()</code>	<code>"ZafRealData"</code>

ZafRealData(void);

The basic constructor allocates a ZafRealData instance and initializes its value to 0.

```
ZafRealData(double value);
```

This constructor allocates a ZafRealData instance and initialize its contents to *value*.

```
ZafRealData(const ZafIChar *string, const ZafIChar
    *format = ZAF_NULLP(ZafIChar));
```

This constructor allocates a ZafRealData instance and initializes its value to the numeric equivalent of *string*. The conversion uses the printf-style specifier *format* to interpret the string. If *format* is null ZafRealData uses its locale-specific default format.

```
ZafRealData(const ZafRealData &copy);
```

This constructor is the copy constructor. It allocates a new ZafRealData instance and copies all member data from *copy*.

```
ZafRealData(const ZafIChar *name, ZafDataPersistence
    &persist);
```

This constructor is the persistent constructor. It allocates a new ZafRealData instance and reads most member data from directory *name* in the persistent data file referred to by *persist*. The StringID() of the new data is *name*.

```
// Sample ZafRealData creation techniques
double value = 10.0;
ZafRealData reall(value);
ZafRealData copyReal = reall;
ZafRealData zeroReal;
```

Destructor

```
virtual ~ZafRealData(void);
```

The virtual destructor is used to free the memory associated with an instantiated ZafRealData object. Unless a ZafRealData object is marked as StaticData() it will be automatically destroyed when all ZafReal objects that refer to it are destroyed.

Members

```
virtual void Clear(void);
```

Clear() sets the value of a ZafRealData object to zero.

```
operator double();
```


See Value().

```
virtual int FormattedText(ZafIChar *buffer, int
    maxLength, const ZafIChar *format = 0) const;
```

FormattedText() fills *buffer* with a string representation of the ZafRealData using the printf-style specifier *format* to build the string. A locale-specific default format is used if format is not included. Buffer contents will be truncated if they exceed *maxLength*. FormattedText() returns the integer value it receives from its call to sprintf().

```
// Show results of FormattedText().
ZafIChar buffer[256];
```

```
ZafRealData myreal(1234.56);
myreal.FormattedText(buffer, sizeof(buffer));
printf("real - %s\n", buffer);
myreal.FormattedText(buffer, sizeof(buffer), "%+8.3g");
printf("real - %s\n", buffer);
```

```
=====
real - 1234.5600
real - +1234.560
```

```
virtual ZafError SetReal(double value);
virtual ZafError SetReal(const ZafIChar *buffer, const
    ZafIChar *format);
virtual ZafError SetReal(const ZafRealData &real);
```

SetReal() functions set the numeric value of the ZafRealData object from doubles, another ZafRealData, or an interpreted string. Refer to FormattedText() for more information on ZafRealData/string conversions. Overloaded operator = offers similar functionality to SetBignum and is more commonly used.

```
double Value(void) const;
operator double();
```

Value() returns the value of a ZafRealData object as a double. The convenience operator double(), which returns Value(), is more commonly used. Refer to ZafIntegerData for sample code showing the different usages of these two functions.

```
ZafRealData operator++(void);
```

```
ZafRealData operator++(int);
```

These pre- and post-operators increment the ZafRealData object's value by 1.

```
ZafRealData operator--(void);
```

```
ZafRealData operator--(int);
```

These pre- and post-operators decrement the ZafRealData object's value by 1.

```
ZafRealData &operator=(double value);
```

This operator assigns the ZafRealData object's value to the input *value*.

```
ZafRealData &operator+=(double value);
```

```
ZafRealData &operator-=(double value);
```

These operators increments or decrement the ZafRealData object's value by the input *value*.

```
ZafRealData &operator*=(double value);
```

This operator multiplies the ZafRealData object's value by the input *value* and uses the resulting product to set the ZafRealData object's value.

```
ZafRealData &operator/=(double value);
```

This operator divides the ZafRealData object's value by the input *value* and uses the resulting quotient to set the ZafRealData object's value.

```
ZafRealData &operator%=(double value);
```

This operator divides the ZafRealData object's value by the input *value* and uses the resulting remainder to set the ZafRealData object's value.

ZafScrollBar

AutoSize	ScrollData	ScrollType
----------	------------	------------

The ZafScrollBar object provides support for both scroll bars and sliders. Scroll bars are generally used as support objects for other classes that allow scrolling, while sliders generally allow end users to select from a value-range. ZafScrollBar utilizes native scroll bar and slider APIs, if available.

Declaration

```
#include <z_scroll2.hpp>
```

Inheritance

```
ZafScrollBar : ZafWindow : ((ZafWindowObject :
    ZafElement), ZafList)
```

Constructors

All ZafScrollBar constructors initialize the member variables associated with an instantiated ZafScrollBar object. The default values set by the ZafScrollBar and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafScrollBar. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafScrollBar

AutoSize()	true
ScrollData()	null
ScrollType()	ZAF_VERTICAL_SCROLL

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Moveable()	false [†]
SelectionType()	ZAF_SINGLE_SELECTION [†]
Sizeable()	false [†]
Temporary()	false [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	false [†]
ParentPalette()	false [†]

Member Initializations

SupportObject() true (if not a slider object)

ZafElement

ClassID() ID_ZAF_SCROLL_BAR

ClassName() "ZafScrollBar"

```
ZafScrollBar(int left, int top, int width, int height,  
              long minimum, long maximum, long current, long delta,  
              long showing, ZafScrollBarType scrollType =  
              ZAF_VERTICAL_SCROLL);
```

This constructor is useful in straight-code situations, particularly if the ZafScrollBar object is to create, maintain, and destroy its own ZafScrollData object automatically.

left and *top* specify the position where the left and top of the object will be placed on its parent, while *width* and *height* specify the width and height of the object. *left*, *top*, *width*, and *height* are specified in cell coordinates by default, but may be specified using another coordinate system if desired. See ZafWindowObject::SetCoordinateType() for more information.

minimum, *maximum*, *current*, *delta*, and *showing* specify the values to be used in the ZafScrollData object automatically created by this constructor. *scrollType* specifies the type of ZafScrollBar object to be created. See ZafScrollData and ScrollType() for more information.

```
ZafScrollBar(int left, int top, int width, int height,  
              ZafScrollData *scrollData = ZAF_NULLP(ZafScrollData),  
              ZafScrollBarType scrollType = ZAF_VERTICAL_SCROLL);
```

This constructor is also useful in straight-code situations, particularly when a ZafScrollData object, *scrollData*, has already been created to be associated with the ZafScrollBar object. For more information on using ZafScrollData objects, see ZafScrollData. Other parameters have the same meaning as in the previous constructor.

```
ZafScrollBar(const ZafScrollBar &copy);
```

The copy constructor is used in conjunction with the overloaded Duplicate() function. It allocates a new ZafScrollBar object and initializes its data from *copy*. If the original ZafScrollBar's internal data objects are StaticData() then the new ZafScrollBar object points to the originals, otherwise copies are made.

```
ZafScrollBar(const ZafIChar *name, ZafObjectPersistence
    &persist);
```

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

The following example demonstrates how to create ZafScrollBar objects:

```
// Create a vertical list with a support scroll bar.
ZafVtList *vList = new ZafVtList(1, 1, 40, 5);
vList->Add(new ZafScrollBar(0, 0, 0, 0));

// Create a slider control using a ZafScrollData object.
ZafScrollData *scrollData = new ZafScrollData(0, 100, 0, 10,
    10);
window->Add(new ZafScrollBar(1, 1, 40, 1, scrollData,
    ZAF_HORIZONTAL_SLIDER));
```

Destructor

```
virtual ~ZafScrollBar(void);
```

The destructor is used to free the memory associated with a ZafScrollBar object, including all the data object pieces that are Destroyable(). It chains to the ZafWindow, ZafList, ZafWindowObject and ZafElement destructors.

Generally, the programmer will not destroy a ZafScrollBar object directly since it is automatically destroyed when its parent object is destroyed. For more information on child object deletion, see ZafWindow::~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
bool AutoSize(void) const;
virtual bool SetAutoSize(bool autoSize);
```

If AutoSize() is true, the object will automatically adjust its thickness to match the normal size for each environment. Otherwise, the object will display with the region passed into the constructor. AutoSize() defaults to true.

```
ZafScrollData *ScrollData(void) const;
virtual ZafError SetScrollData(ZafScrollData *scroll);
```

The ScrollData() object contains the actual data used by the ZafScrollBar. Zaf-ScrollData objects may be shared among several ZafScrollBar objects (to save memory, for example) or it may belong to a single ZafScrollBar object. If shared, all the associated objects will be updated when the ScrollData() changes. SetScrollData() may be used to associate a ScrollData() object with a ZafScrollBar object. See ZafDataManager for more information on data sharing.

ScrollData() returns a pointer to the ScrollData() object associated with the ZafScrollBar. SetScrollData() normally returns ZAF_ERROR_NONE.

```
ZafScrollBarType ScrollType(void) const;  
virtual ZafScrollBarType SetScrollType(ZafScrollBarType  
    scrollType);
```

ScrollType() specifies the object's type. Each type appears and behaves differently. ScrollType() defaults to ZAF_VERTICAL_SCROLL. Possible values are listed:

ScrollType()	Description
ZAF_CORNER_SCROLL	Corner scroll bar (typically used for spacing when both horizontal and vertical scroll bars are present)
ZAF_HORIZONTAL_SCROLL	Horizontal scroll bar
ZAF_VERTICAL_SCROLL	Vertical scroll bar
ZAF_HORIZONTAL_SLIDER	Horizontal slider control
ZAF_VERTICAL_SLIDER	Vertical slider control
ZAF_SIZEGRIP_SCROLL	Corner scroll bar with size-grip appearance (typically used on ZafWindow objects for a Windows95 look)

ZafStatusBar

The ZafStatusBar object is used strictly for informational purposes. For example, a string field on a status bar may display help messages (see ZafHelpTips for more information). Many objects available in ZAF may be placed on a status bar, but any object placed on a status bar will act Disabled(), since the status bar is Disabled(). ZafStatusBar is also Noncurrent(), meaning that it does not receive focus.

Declaration `#include <z_status.hpp>`

Inheritance `ZafStatusBar : ZafWindow : ((ZafWindowObject : ZafElement), ZafList)`

Constructors All ZafStatusBar constructors initialize the member variables associated with an instantiated ZafStatusBar object. The default values set by the ZafStatusBar and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafStatusBar. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Moveable()	false [†]
SelectionType()	ZAF_SINGLE_SELECTION [†]
Sizeable()	false [†]
Temporary()	false [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	true
Disabled()	true [†]
Focus()	false [†]
Noncurrent()	true [†]
RegionType()	ZAF_AVAILABLE_REGION [†]
SupportObject()	true [†]
UserFunction()	null [†]

Member Initializations

ZafElement

ClassID()	ID_ZAF_STATUS_BAR
ClassName()	"ZafStatusBar"

```
ZafStatusBar(int left, int top, int right, int bottom);
```

This constructor is useful in straight-code situations. The *left*, *top*, *right* and *bottom* parameters specify the left, top, right and bottom of the object, respectively. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. Currently, the *left* and *right* parameters are ignored since the status bar automatically fills the bottom edge of its parent's client region, but they may be used in the future. Note that the **ZafStatusBar** object is always positioned on the bottom of its parent window's client region, so the *top* and *bottom* parameters are only used in calculating the height of the status bar.

```
ZafStatusBar(const ZafStatusBar &copy);
```

The copy constructor is used in conjunction with the overloaded **Duplicate()** function. It accepts another **ZafStatusBar** object and copies the object's information.

```
ZafStatusBar(const ZafIChar *name, ZafObjectPersistence  
              &persist);
```

The final constructor is used for persistence. Refer to **ZafWindow** for more information, since most persistence is done at the **ZafWindow** level.

An example of how to create a toolbar with buttons follows:

```
// Create a sample window with a status bar and a string.  
ZafWindow *window1 = new ZafWindow(0, 0, 50, 10);  
// Create a status bar.  
ZafStatusBar *statusBar = new ZafStatusBar(0, 0, 80, 1);  
// Add a string that occupies the entire status bar.  
ZafString *bigString = new ZafString(0, 0, 0, "Big Sample  
          String", -1);  
bigString->SetRegionType(ZAF_AVAILABLE_REGION);  
statusBar->Add(bigString);  
// Add the status bar to the window.  
window1->Add(statusBar);
```

Destructor

```
virtual ~ZafStatusBar(void);
```

The destructor is used to free the memory associated with a ZafStatusBar object. It chains to the ZafWindow, ZafList, ZafWindowObject and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafStatusBar object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~ZafWindow().

ZafString

AutoClear	CursorOffset	DefaultValidateFunction
Event	HzJustify	InputFormatData
InputFormatText	Invalid	LowerCase
OutputFormatData	OutputFormatText	Password
RangeData	RangeText	SetSelected
StringData	Text	Unanswered
UpperCase	VariableName	ViewOnly

The ZafString object is a single-line text object that allows user input through the keyboard. Other user interaction is also supported such as copy/cut/paste. Several formatting options are available in ZafString objects such as password, upper-case, lower-case, and variable-name.

The ZafString class is used as a base class for other single-line text classes such as ZafFormattedString, ZafInteger, and ZafReal. These classes inherit much of the base functionality provided by ZafString.

Declaration

#include <z_str1.hpp>

Inheritance

ZafString : ZafWindowObject : ZafElement

Constructors

All ZafString constructors initialize the member variables associated with an instantiated ZafString object. Default values set by the ZafString follow, as well as base class values when overridden by ZafString.

Member Initializations

ZafString

AutoClear()	true
HzJustify()	ZAF_HZ_LEFT
InputFormatData()	null
Invalid()	false
LowerCase()	false
OutputFormatData()	null
Password()	false
RangeData()	null
StringData()	null
Unanswered()	false
UpperCase()	false
VariableName()	false
ViewOnly()	false

Member Initializations

ZafWindowObject

Bordered()	true
memberUserFunction	ZafString:: DefaultValidateFunction
zafRegion.bottom	top

ZafElement

ClassID()	ID_ZAF_STRING
ClassName()	"ZafString"

```
ZafString(int left, int top, int width, const ZafIChar
          *text, int maxLength);
```

This constructor is useful in straight-code situations, particularly to have the ZafString object to create, maintain and destroy its own ZafStringData object automatically. The *left* and *top* parameters specify the position where the left and top of the object will be placed on its parent, respectively. The *width* parameter specifies the width of the object. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. The *text* parameter is the string you wish to initially appear in the new ZafString object, and *maxLength* is the maximum number of characters the user may type into the string. If you pass -1 in for the *maxLength* parameter, the user may type any number of characters into the string.

```
ZafString(int left, int top, int width, ZafStringData
          *stringData);
```

This constructor is useful in straight-code situations where a ZafStringData object has already been created. This constructor may be used to maintain data pieces yourself, rather than have the ZafString class create and maintain the data pieces automatically. For example, to maintain a database of ZafStringData objects and tie them into ZafString objects, maintain your own ZafStringData objects and create ZafString objects using your ZafStringData objects by passing them into the *stringData* parameter of this constructor. For more information on using ZafStringData objects, see the chapter on ZafStringData. The *left*, *top* and *width* parameters are the same as the previous constructor.

```
ZafString(const ZafString &copy);
```

The copy constructor is used in conjunction with the overloaded Duplicate() function. It accepts another ZafString object and copies the object's informa-

tion. If the data objects are StaticData() then the new ZafString object points to the original data objects, otherwise a copy is made for the new ZafString object. This allows a programmer to use static data for more than one ZafString object.

```
ZafString(const ZafIChar *name, ZafObjectPersistence
           &persist);
```

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

Sample ZafString creation techniques follow:

```
// Create a sample window with string objects.
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);
// Create strings and pass in the text directly.
window1->Add(new ZafString(0, 1, 25, "String1", 25));
window1->Add(new ZafString(0, 2, 25, "String2", 25));
...
// Create a sample window with string objects.
ZafWindow *window2 = new ZafWindow(10, 10, 40, 10);
// Create string data objects.
ZafStringData *stringData1 = new ZafStringData("String1", 25);
ZafStringData *stringData2 = new ZafStringData("String2", 25);
// Create strings that use the data previously created.
window2->Add(new ZafString(0, 1, 25, stringData1));
window2->Add(new ZafString(0, 2, 25, stringData2));
```

Destructor

```
virtual ~ZafString(void);
```

The destructor is used to free the memory associated with a ZafString object, including all the data object pieces (such as StringData()) that are Destroyable(). It chains to the ZafWindowObject and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafString object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
bool AutoClear(void) const;
```

```
virtual bool SetAutoClear(bool autoClear);
```

If `AutoClear()` is true, the string becomes entirely highlighted when the object gains focus, so that whatever the user types replaces the current text. The default value of this attribute is true, but the user may call `SetAutoClear()` to change it.

```
int CursorOffset(void) const;
virtual ZafError SetCursorOffset(int position);
```

`CursorOffset()` returns the character offset of the cursor (the insertion point) in the `ZafString` object. This offset is zero-based, so the first character offset in the `ZafString` is 0. The user may call `SetCursorOffset()` to reposition the cursor to *position* in the `ZafString`. For example:

```
// Move the cursor to the beginning of the string.
object->SetCursorOffset(0);
```

```
ZafEventType DefaultValidateFunction(const
    ZafEventStruct &event, ZafEventType ccode);
```

`DefaultValidateFunction()` is automatically called when a `ZafString` gets `L_SELECT` or `N_NON_CURRENT` events. The programmer should never call `DefaultValidateFunction()` directly. `DefaultValidateFunction()` calls the `Validate()` function, which does nothing by default in the `ZafString` class, but is overloaded by other classes that require validation, such as `ZafDate` and `ZafInteger`. *event* is the event that triggered the call to `DefaultValidateFunction()`, and *ccode* is the event type that triggered the call.

The return value for `DefaultValidateFunction()` is an error code indicating the result of the field validation. The return value is `ZAF_ERROR_NONE` if validation succeeded, and another value of the enumeration `ZafError` appropriate to the derived class otherwise. For example, `ZAF_ERROR_OUT_OF_RANGE` may be returned by a `ZafDate` object if a date out of the required range was entered.

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function handles all events sent to the `ZafString` object, whether by processing the events itself, or by passing them to `ZafWindowObject::Event()` for base class processing. See `ZafWindowObject` for more information.

In addition to those handled by its base classes, ZafString handles the following events:

Event	Description
S_COPY	causes the object to copy its selected data to the clipboard
S_CUT	causes the object to cut its selected data to the clipboard
S_PASTE	causes the object to paste the clipboard's data to its current cursor position, replacing any selected data
S_COPY_DATA	causes the object to copy event.windowObject's StringData() if event.windowObject is a ZafString object
S_SET_DATA	causes the object to create a new StringData() object, then copy into it event.windowObject's StringData() if event.windowObject is non-null and is a ZafString object

```
ZafHzJustify HzJustify(void) const;  
virtual ZafHzJustify SetHzJustify(ZafHzJustify  
    hzJustify);
```

HzJustify() controls the string's horizontal justification, and is ZAF_HZ_LEFT by default. The user may call SetHzJustify() to change it to ZAF_HZ_RIGHT or ZAF_HZ_CENTER.

```
ZafStringData *InputFormatData(void) const;  
const ZafIChar *InputFormatText(void) const;  
ZafError SetInputFormat(const ZafIChar *format);  
virtual ZafError SetInputFormatData(ZafStringData  
    *format);
```

The InputFormatData() portion of ZafStringData is used in use input validation for classes derived from ZafString, such as ZafDate and ZafFormattedString. InputFormatData() returns a pointer to the actual ZafStringData object that stores the input format data. InputFormatText() returns the string portion of the input format data.

The programmer may set the InputFormatData() with SetInputFormat() or SetInputFormatData(). SetInputFormat() is useful for passing in a simple string to be used in either modifying an existing input format data object, or automatically creating a new input format data object using the string you pass in. SetInputFormatData() is useful when you want ZafString to use an existing input format data object.

For more information on format string arguments and what they mean, see ZafStringData. Sample usage follows:

```
// Get the input format two different ways.
const ZafStringData *inputFormat = object->InputFormatData();
const ZafIChar *inputText = object->InputFormatText();
...
// Set the input format two different ways.
ZafDate *date1 = new ZafDate(0, 0, 15);
ZafDate *date2 = new ZafDate(0, 1, 15);
date1->SetInputFormat("%m/%d/%y");
ZafStringData *inputFormatData = new ZafStringData("%m/%d/%y");
date2->SetInputFormatData(inputFormatData);
```

```
bool Invalid(void) const;
virtual bool SetInvalid(bool invalid);
```

Invalid() is used in classes derived from ZafString that require validation, such as ZafBignum and ZafTime. If Invalid() is true, the user is forced to enter a valid value into the field. The default value of this attribute is false, but the user may call SetInvalid() to change it.

As an example, if a ZafInteger has a range of 1..100 and initially has the value 0, SetInvalid(true) would be called so that the user must enter a value in the range 1..100. Invalid() may be used in any scenario where a field is restricted by validation, whether the validation be automatically handled by ZAF or by another validation method that the programmer has implemented. After the user has entered a valid value into the field, Invalid() will be reset to false automatically by ZAF.

For example:

```
// Create the object with invalid data and force the user to
// enter something valid.
ZafInteger *int1 = new ZafInteger(0, 0, 15, 0);
int1->SetInvalid(true);
int1->SetRange("1..100");
```

```
bool LowerCase(void) const;
virtual bool SetLowerCase(bool lowerCase);
```

If LowerCase() is true, any upper-case characters typed into the ZafString field will be converted to lower-case. Any characters the programmer directly places into the ZafString's StringData() will be displayed as lower-case, but not converted. LowerCase() is false by default, but the user may call SetLow-

erCase() to change it. The programmer should never set both LowerCase() and UpperCase() on the same object at the same time, as the result is undefined.

```
ZafStringData *OutputFormatData(void) const;
const ZafIChar *OutputFormatText(void) const;
ZafError SetOutputFormat(const ZafIChar *format);
virtual ZafError SetOutputFormatData(ZafStringData
    *format);
```

The OutputFormatData() piece of ZafString is used to display the user's input in a uniform format, for classes derived from ZafString—such as ZafDate and ZafFormattedString. OutputFormatData() returns a pointer to the actual ZafStringData object that stores the output format data. OutputFormatText() returns the string portion of the output format data.

The programmer may set the output format data with SetOutputFormat() or SetOutputFormatData(). SetOutputFormat() is useful for passing in a simple string to be used in either modifying an existing output format data object, or automatically creating an output format data object using the string you pass in. SetOutputFormatData() is useful when you want ZafString to use an existing output format data object.

For more information on format string arguments and what they mean, see ZafStringData. Set InputFormat() for sample code.

```
bool Password(void) const;
virtual bool SetPassword(bool password);
```

If Password() is true, all characters typed into the ZafString field will be hidden by appearing as asterisk or bullet characters. Internally, however, the data will accurately represent what was typed in. Password() is false by default, but the user may call SetPassword() to change it. Note: if drag-and-drop is enabled on a Password() ZafString, the user may drop away from it and reveal the actual contents.

```
ZafStringData *RangeData(void) const;
const ZafIChar *RangeText(void) const;
ZafError SetRange(const ZafIChar *range);
virtual ZafError SetRangeData(ZafStringData *range);
```

The RangeData() portion of ZafString is used to validate that the user's input matches a specified range of values that are valid for the object, for classes derived from ZafString such as ZafDate and ZafFormattedString. RangeData()

returns a pointer to the actual ZafStringData object that stores the range data. RangeText() returns the string portion of the range data.

The programmer may set the range data with SetRange() or SetRangeData(). SetRange() is useful for passing in a simple string to be used in either modifying an existing range data object, or automatically creating a range data object using the string you pass in. SetRangeData() is useful when you want ZafString to use an existing range data object.

For more information on range string arguments and what they mean, see ZafStringData. Refer to InputFormat() for sample code.

```
virtual bool SetSelected(bool selected);
```

This overloaded function adds to the functionality of ZafWindowObject::SetSelected() by performing platform-specific display operations such as displaying the contents of the string in a “selected” state.

```
ZafStringData *StringData(void) const;  
virtual ZafError SetStringData(ZafStringData *string);
```

The StringData() object is where the actual data is stored for the ZafString object. The StringData() piece may be shared among several ZafString objects, or it may belong to a single ZafString object. If shared among several ZafString objects, all the associated ZafString objects will be updated when the StringData() piece changes. SetStringData() may be used to associate a StringData() object with a ZafString object. For more information on data sharing in ZAF, see ZafDataManager.

The return value for StringData() is a pointer to the StringData() object associated with the ZafString object. The return value for SetStringData() is normally ZAF_ERROR_NONE. The following code shows the proper use of these functions:

```
// Get the data.  
const ZafStringData *data = string->StringData();  
...  
// Add the string data.  
ZafStringData *newData = new ZafStringData("String", 25);  
string->SetStringData(newData);
```

```
virtual const ZafIChar *Text(void);  
virtual ZafError SetText(const ZafIChar *text);
```

The textual data of a ZafString (contained in the StringData() object) may be returned or set with Text() and SetText(). These functions provide simple accessibility to the StringData() of a ZafString, and may be used if the programmer does not wish to interact with the data portion of the object.

The return value for Text() is a pointer to the textual information in the data object of a ZafString. The return value for SetText() is normally ZAF_ERROR_NONE. The following code shows the proper use of these functions:

```
// Get the text.
const ZafIChar *text = string->Text();
...
// Set the new text.
string->SetText("New Text");
```

```
bool Unanswered(void) const;
virtual bool SetUnanswered(bool unanswered);
```

If Unanswered() is true, the ZafString field will be initially blank. As soon as text is entered into the ZafString object—either by calling SetText() or by the end user entering data—the Unanswered() attribute is set to false. Unanswered() is false by default, but the user may call SetUnanswered() to make changes. Calling SetUnanswered(true) will clear the contents of the string object and its StringData().

```
bool UpperCase(void) const;
virtual bool SetUpperCase(bool upperCase);
```

If UpperCase() is true, any lower-case characters typed into the ZafString field will be converted to upper-case. Any characters the programmer directly places into the ZafString's StringData() will be displayed as upper-case, but not converted. UpperCase() is false by default, but the user may call SetUpperCase() make changes. The programmer should never set both LowerCase() and UpperCase() on the same object at the same time, as the result is undefined.

```
bool VariableName(void) const;
virtual bool SetVariableName(bool variableName);
```

If VariableName() is true, all space characters typed into the ZafString field will be converted to underscore ('_') characters. Any space characters the programmer directly places into the ZafString's StringData() will be converted to

underscores. `VariableName()` is false by default, but the user may call `SetVariableName()` to change it.

```
bool ViewOnly(void) const;  
virtual bool SetViewOnly(bool viewOnly);
```

A `ViewOnly()` ZafString object may not be edited, but may be the current object of a window, may be copied into the clipboard, and arrow keys may be used to navigate it. `ViewOnly()` is false by default, but the user may call `SetViewOnly()` to change it.

ZafStringData

Append	Char	Clear
DynamicText	DynamicOSText	DynamicOSWText
Compare	Insert	Length
MaxLength	Remove	Replace
StaticData	SetOSText	SetOSWText
Text	ZafChar	operator -
operator -=	operator !=	operator +
operator +=	operator <	operator <=
operator =	operator ==	operator >
operator >=	operator []	

ZafStringData objects can be used to store and conveniently manipulate character-based data, including Unicode (double-byte) characters.

ZafStringData combines string encapsulation with data and object notification from ZafData. It is most often used in conjunction with user interface objects such as ZafString, ZafFormattedString, ZafText and ZafButton but may be used as a stand-alone object if desired.

All ZafData objects may make use of printf-style formatting and parsing arguments during string operations. Refer to standard library documentation for detailed information on printf functions and conversion characters.

Numerous overloaded operators are available to facilitate manipulation of ZafStringData objects.

Declaration `#include <z_str.hpp>`

Inheritance `ZafStringData : ZafFormatData : ZafData :`
 `(ZafNotification, ZafElement)`

Constructors ZafStringData constructors initialize the member variables associated with a new ZafStringData object and allocate space to hold the string data.

The default values set by ZafStringData follow, if they are overridden from those set by base class constructors:

Member Initializations

ZafStringData	
MaxLength()	-1 (varies by constructor)
StaticData()	false (varies by constructor)
Text()	" " (varies by constructor)

Member Initializations

ZafElement

ClassID()	ID_ZAF_STRING_DATA
ClassName()	"ZafStringData"

```
ZafStringData(bool staticData = false);
```

This constructor allocates a ZafStringData and initializes its contents to null. If *staticData* is true its internal string array is not deleted when the ZafStringData is destroyed.

```
ZafStringData(const ZafIChar *value, int maxLength = -1,
    bool staticData = false);
```

```
ZafStringData(const char *value, int maxLength = -1, bool
    staticData = false);
```

These constructors allocate a ZafStringData instance and initialize its contents to *value*. The data is automatically truncated at *maxLength* unless *maxLength* is -1 which allows the ZafStringData to dynamically allocate the minimum space necessary to store *value*. Refer to the destructor and StaticData() for more information about *staticData*.

Note: The third constructor is available for use with Unicode since ZafIChar defaults to be a char in ISO mode.

```
ZafStringData(const ZafStringData &copy);
```

This constructor is the copy constructor, which allocates a new ZafStringData instance and copies all member data from *copy*.

```
ZafStringData(const ZafIChar *name, ZafDataPersistence
    &persist);
```

This constructor is the persistent constructor. It allocates a new ZafStringData instance and reads most member data from directory *name* in the persistent data file referred to by *persist*. The StringID() of the new data is *name*.

```
// Sample ZafStringData creation techniques
ZafScrollData string1("Hello World!");
ZafStringData copyString = string1;
ZafStringData emptyString;
```

Destructor `virtual ~ZafStringData(void);`

The destructor is used to free the memory associated with a ZafStringData object. If StaticData() is true the string array associated with the ZafStringData object is not freed.

Members `virtual ZafStringData &Append(const ZafIChar *text);`

This function concatenates text onto the end of the string. If the length of the old string plus the length of the string that is being appended is greater than MaxLength() then the string is truncated.

```
// Create the string-data.
ZafStringData hello("Hello world", 20);
printf("String: %s\n", hello.Text());
...
// Append to the object's contents.
hello.Append("!!!");
printf("String: %s\n", hello.Text());
=====
String: Hello world
String: Hello world!!!
```

```
ZafIChar Char(int offset) const;
virtual ZafError SetChar(int offset, ZafIChar value, bool
    ignoreNotification = true);
```

Char() returns the character at index *offset*. Offset is zero based.

SetChar() replaces one character at position *offset* with *value*. If *ignoreNotification* is true then the ZafStringData object will not notify the items in its notification list about changes made during this operation.

```
virtual Void Clear(void);
```

Clear() sets the Text() portion of the specified object to the empty string ("").

```
ZafIChar *DynamicText(void) const;
char *DynamicOSText(void) const;
wchar_t *DynamicOSWText(void) const;
```

These functions duplicate the string contained in ZafStringData and return a pointer to the duplicate. They are useful in situations when the programmer requires a pointer to a string that may be safely deleted (by another function,

for example). `DynamicOS*Text()` functions return the result in different formats.

Note: `DynamicOSWText()` is only available in Unicode mode. It converts the data to a wide character array in the native OS codeset.

```
virtual int Compare(const ZafIChar *text, bool
    caselessCompare = false) const;
```

`Compare()` provides a safe method for performing string comparisons on `ZafStringData` objects. If *caselessCompare* is true, the strings are compared without regard for case. The following code shows the proper use of, and results from performing a `Compare()` operation.

```
// Create the string-data.
ZafStringData hello("Hello World");

if (!hello.Compare("hello world")
    printf("#1: The strings are equal.\n");

if (!hello.Compare("hello world", true)
    printf("#2: The strings are equal.\n");

=====
#2: The strings are equal.
```

```
virtual ZafStringData &Insert(int offset, const ZafIChar
    *text, int length = -1);
```

`Insert()` will insert the string *offset* into a `ZafStringData` object. *length* specifies the maximum length of the resulting `ZafStringData`. The following code shows the use of the `Insert()` operation.

```
// Create the string-data.
ZafStringData hello("Hello world");
printf("String: %s\n", hello.Text());
...
// Insert a string into the object's contents.
hello.Insert(6, "to the ");
printf("String: %s\n", hello.Text());
=====
String: Hello world
String: Hello to the world
```



```
int Length(void) const;
```

`Length()` returns the number of characters in the `ZafStringData` regardless of `MaxLength()`.

```
int MaxLength(void) const;  
virtual int SetMaxLength(int maxLength);
```

`MaxLength()` returns the maximum length of the string that may be stored in this `ZafStringData`. If the maximum length is dynamic (allowing ZAF to reallocate the buffer as the string length changes) `MaxLength()` returns -1.

`SetMaxLength()` sets the internal maximum length allowed by the `ZafStringData` object. If *maxLength* is greater than the allocated buffer and the object does not have static data then the object will allocate a larger buffer and copy the data. If *maxLength* is -1 the `ZafStringData` will dynamically allocate the minimum buffer necessary to hold its data.

```
virtual ZafString &Remove(int offset, int size);  
virtual ZafString &Remove(const ZafIChar *text);
```

These functions remove text from the `ZafStringData` object. The first form removes *size* characters beginning with character *offset*+1. The second form removes the first occurrence of *text*. The following code shows the use of the `Remove()` methods.

```
// Create the string-data.  
ZafStringData hello("Hello to the world");  
printf("String: %s\n", hello.Text());  
...  
// Remove text from the object's contents.  
hello.Remove("the ");  
printf("String: %s\n", hello.Text());  
...  
// Remove text from the object's contents.  
hello.Remove(6, 3);  
printf("String: %s\n", hello.Text());
```

```
=====  
String: Hello to the world  
String: Hello to world  
String: Hello world
```

```
virtual ZafStringData &Replace(int offset, int size,
    const ZafIChar *text, int length = -1);
```

This function combines the functionality of `Remove()` and `Insert()` to replace text in a `ZafStringData` object. See `Remove()` and `Insert()` for description of parameters. The following code demonstrates the proper use of the `Replace()` function.

```
// Create the string-data.
ZafStringData hello("Hello world");
printf("String: %s\n", hello.Text());
...
// Insert a string into the object's contents.
hello.Replace(6, 5, "Universe");
printf("String: %s\n", hello.Text());

=====
String: Hello world
String: Hello Universe
```

```
bool StaticData(void) const;
virtual bool SetStaticData(bool staticData);
```

If `StaticData()` is true, the string array associated with a `ZafStringData` object will not be deleted when the `ZafStringData` is destroyed or reallocated. If `SetStaticData(false)` while `StaticData()==true`, the object will make a copy of the data it was pointing to.

```
virtual ZafError SetOSText(const char *text);
ZafError SetOSWText(const wchar_t *text);
```

`SetOSText()` converts *text* to the internal representation used by the native OS. These functions are used internally by ZAF and will generally not be called by the programmer.

`SetOSWText()` is only available in Unicode mode.

```
const ZafIChar *Text(void) const;
virtual ZafError SetText(const ZafIChar *text);
virtual ZafError SetText(const ZafIChar *text, int
    maxLength);
virtual ZafError SetText(const ZafStringData &string);
```

```
virtual ZafError SetText(const ZafIChar *buffer, const
    ZafIChar *format);
```

Text() returns the internal pointer to the string array maintained by ZafStringData. The contents of this array should not be directly manipulated.

SetText() functions set the text of the ZafStringData object. For example:

```
// Create the string-data.
ZafStringData hello("", 20);
printf("String: %s\n", hello.Text());
...
// Append to the object's contents.
hello.SetText("Hello world");
printf("String: %s\n", hello.Text());
=====
String:
String: Hello world
```

```
operator const ZafIChar *() const { return value; }
```

ZafIChar() provides a useful alternative to Text(). Refer to Text() for more information. For example:

```
// Create a string data.
ZafStringData string ("mystring");
// The following test uses the overloaded operator
if (string)
    DoSomething();
```

```
ZafStringData operator-(const ZafStringData &string1,
    const ZafStringData &string2);
```

```
ZafStringData operator-(const ZafStringData &string,
    const ZafIChar *value);
```

```
ZafStringData operator-(const ZafIChar *value, const
    ZafStringData &string);
```

These operators create and return a new ZafStringData object that is the equivalent of the first string with the second string removed. For more information about Remove see the Remove section of the ZafStringData chapter.

```
ZafStringData &operator-=(const ZafStringData &value);
```

```
ZafStringData &operator-=(const ZafIChar *value);
```

These operators provide shortcuts to using the Remove methods. For more information about the Remove methods see the Remove section of this chapter.

```
bool operator!=(const ZafStringData &string1, const
    ZafStringData &string2);
bool operator!=(const ZafStringData &string, const
    ZafIChar *value);
bool operator!=(const ZafIChar *value, const
    ZafStringData &string);
```

These operators use the Compare method of ZafStringData to determine if the two strings are different. For more information about Compare see the Compare section of the ZafStringData chapter.

```
ZafStringData operator+(const ZafStringData &string1,
    const ZafStringData &string2);
ZafStringData operator+(const ZafStringData &string,
    const ZafIChar *value);
ZafStringData operator+(const ZafIChar *value, const
    ZafStringData &string);
```

These operators create and return a new ZafStringData object that is the equivalent of the first string with the second string appended to it. For more information about Append see the Append section of the ZafStringData chapter.

```
ZafStringData &operator+=(const ZafStringData &value);
ZafStringData &operator+=(const ZafIChar *value);
```

These operators provide shortcuts to using the Append methods. For more information about the Append methods see the Append section of this chapter.

```
bool operator<(const ZafStringData &string1, const
    ZafStringData &string2);
bool operator<(const ZafStringData &string, const
    ZafIChar *value);
bool operator<(const ZafIChar *value, const ZafStringData
    &string);
```

These operators use the Compare method of ZafStringData to determine if the first string is alphabetically before the second string. For more information about Compare see the Compare section of the ZafStringData chapter.

```
bool operator<=(const ZafStringData &string1, const
               ZafStringData &string2);
bool operator<=(const ZafStringData &string, const
               ZafIChar *value);
bool operator<=(const ZafIChar *value, const
               ZafStringData &string);
```

These operators use the Compare method of ZafStringData to determine if the first string is alphabetically before the second string, or if the strings are equivalent. For more information about Compare see the Compare section of the ZafStringData chapter.

```
ZafStringData &operator=(const ZafStringData &value);
ZafStringData &operator=(const ZafIChar *value);
```

These operators provide shortcuts to using the SetText methods. For more information about the SetText methods see the SetText section of this chapter.

```
bool operator==(const ZafStringData &string1, const
               ZafStringData &string2);
bool operator==(const ZafStringData &string, const
               ZafIChar *value);
bool operator==(const ZafIChar *value, const
               ZafStringData &string);
```

These operators use the Compare method of ZafStringData to determine if the two strings are equivalent. For more information about Compare see the Compare section of the ZafStringData chapter.

```
bool operator>(const ZafStringData &string1, const
               ZafStringData &string2);
bool operator>(const ZafStringData &string, const
               ZafIChar *value);
bool operator>(const ZafIChar *value, const ZafStringData
               &string);
```

These operators use the Compare method of ZafStringData to determine if the first string is alphabetically after the second string. For more information about Compare see the Compare section of the ZafStringData chapter.

```

bool operator>=(const ZafStringData &string1, const
    ZafStringData &string2);
bool operator>=(const ZafStringData &string, const
    ZafIChar *value);
bool operator>=(const ZafIChar *value, const
    ZafStringData &string);

```

These operators use the Compare method of ZafStringData to determine if the first string is alphabetically after the second string, or if the strings are equivalent. For more information about Compare see the Compare section of the ZafStringData chapter.

```

ZafIChar operator[](int offset) const;
ZafIChar &operator[](int offset);

```

These operators provide a method of accessing individual members of the string contained in the string data object. The following code shows a use of the operators.

```

// Create the string-data.
ZafStringData hello("Hello world.", 20);
printf("String: %s\n", hello.Text());
...
// Append to the object's contents.
hello[11] = '!';
printf("String: %s\n", hello.Text());
=====
String: Hello world.
String: Hello world!

```

ZafText

AutoClear	CursorOffset	CursorPosition
Event	HzJustify	Invalid
StringData	Text	Unanswered
ViewOnly	WordWrap	

The ZafText object is a multi-line text object that allows user input through the keyboard. Other user interaction is also supported such as copy/cut/paste. A ZafText object may have scroll bars associated with it.

Declaration `#include <z_text.hpp>`

Inheritance `ZafText : ZafWindow : ((ZafWindowObject : ZafElement),
 ZafList)`

Constructors All ZafText constructors initialize the member variables associated with an instantiated ZafText object. The default values set by the ZafText and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafText. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafText

<code>AutoClear()</code>	<code>false</code>
<code>HzJustify()</code>	<code>ZAF_HZ_LEFT</code>
<code>Invalid()</code>	<code>false</code>
<code>StringData()</code>	<code>null</code>
<code>Unanswered()</code>	<code>false</code>
<code>ViewOnly()</code>	<code>false</code>
<code>WordWrap()</code>	<code>true</code>

ZafWindow

<code>Destroyable()</code>	<code>false</code> [†]
<code>Locked()</code>	<code>false</code> [†]
<code>Maximized()</code>	<code>false</code> [†]
<code>Minimized()</code>	<code>false</code> [†]
<code>Moveable()</code>	<code>false</code> [†]
<code>SelectionType()</code>	<code>ZAF_SINGLE_SELECTION</code> [†]
<code>Sizeable()</code>	<code>false</code> [†]
<code>Temporary()</code>	<code>false</code> [†]

new ZafText object. This allows a programmer to use static data for more than one ZafText object.

```
ZafText(const ZafIChar *name, ZafObjectPersistence
        &persist);
```

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

Sample ZafText creation techniques follow:

```
// Create a sample window with text objects.
ZafWindow *window1 = new ZafWindow(0, 0, 50, 10);
// Create text objects and pass in the text directly.
window1->Add(new ZafText(0, 1, 20, 3, "Text1", 200));
window1->Add(new ZafText(25, 1, 20, 3, "Text2", 200));
...
// Create a sample window with text objects.
ZafWindow *window2 = new ZafWindow(10, 10, 50, 10);
// Create string data objects.
ZafStringData *stringData1 = new ZafStringData("Text1", 200);
ZafStringData *stringData2 = new ZafStringData("Text2", 200);
// Create text objects that use the data previously created.
window2->Add(new ZafText(0, 1, 20, 3, stringData1));
window2->Add(new ZafText(25, 1, 20, 3, stringData2));
```

Destructor

```
virtual ~ZafText(void);
```

The destructor is used to free the memory associated with a ZafText object, including all the data object pieces that are Destroyable(). It chains to the ZafWindow, ZafList, ZafWindowObject and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafText object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
bool AutoClear(void) const;
virtual bool SetAutoClear(bool autoClear);
```

If `AutoClear()` is true, the text becomes entirely highlighted when the object gains focus so whatever the user types replaces the current text. The default value of this attribute is true, but the user may call `SetAutoClear()` to change it.

```
int CursorOffset(void) const;
virtual ZafError SetCursorOffset(int position);
```

`CursorOffset()` returns the character offset of the cursor (the insertion point) in the `ZafText` object. This offset is zero-based, so the first character offset in the `ZafText` is 0. The user may call `SetCursorOffset()` to reposition the cursor to *position* in the `ZafText`. For example:

```
// Move the cursor to the beginning of the text.
object->SetCursorOffset(0);
```

```
ZafPositionStruct CursorPosition(void) const;
virtual ZafError SetCursorPosition(ZafPositionStruct
    position);
```

`CursorPosition()` returns the character position of the cursor (the insertion point) in the `ZafText` object. This position is zero-based, so the first character position in the `ZafText` is (0, 0). The user may call `SetCursorPosition()` to reposition the cursor to *position* in the `ZafText`. For example:

```
// Move the cursor down one line in the text.
ZafPositionStruct cursorPos = object->CursorPosition();
cursorPos.line++;
object->SetCursorOffset(cursorPos);
```

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function handles all events that get sent to the `ZafText` object, whether by processing the events itself, or by passing them to a base class for processing. See `ZafWindowObject` for more information.

In addition to those handled by its base classes, `ZafText` handles the following events:

Event	Description
<code>S_COPY</code>	causes the object to copy its selected data to the clipboard

Event	Description
S_COPY_DATA	causes the object to copy event.windowObject's StringData() if event.windowObject is a ZafText object
S_CUT	causes the object to cut its selected data to the clipboard
S_PASTE	causes the object to paste the clipboard's data to its current cursor position, replacing any selected data
S_SET_DATA	causes the object to create a new StringData() object, then copy into it event.windowObject's StringData() if event.windowObject is non-null and is a ZafText object

```
ZafHzJustify HzJustify(void) const;  
virtual ZafHzJustify SetHzJustify(ZafHzJustify  
    hzJustify);
```

HzJustify() controls the text's horizontal justification, and is ZAF_HZ_LEFT by default. The user may call SetHzJustify() to change it to ZAF_HZ_RIGHT or ZAF_HZ_CENTER.

```
bool Invalid(void) const;  
virtual bool SetInvalid(bool invalid);
```

This attribute provides a user hook that may be used during validation on a ZafText object. During validation, the programmer may set Invalid() to true to indicate that the end user has entered invalid data. Because it is a user hook, Invalid() is an advanced function that does not have any default behavior in ZafText. The default value of this attribute is false, but the user may call SetInvalid() to change it.

```
ZafStringData *StringData(void) const;  
virtual ZafError SetStringData(ZafStringData *string);
```

StringData() is where the actual data is stored for the ZafText object. The StringData() piece may belong to a single ZafText object, or may be shared among several ZafText objects, in which case all the associated ZafText objects will be updated when the StringData() piece changes. SetStringData() may be used to associate a StringData() object with a ZafText object. For more information on data sharing in ZAF, see ZafDataManager.

The return value for StringData() is a pointer to the StringData() object associated with the ZafText object. The return value for SetStringData() is normally

ZAF_ERROR_NONE. The following code shows the proper use of these functions:

```
// Get the data.
const ZafStringData *data = text->StringData();
...
// Add the string data.
ZafStringData *newData = new ZafStringData("Text", 25);
text->SetStringData(newData);
```

```
virtual const ZafIChar *Text(void);
virtual ZafError SetText(const ZafIChar *text);
```

The textual data of a ZafText (contained in the StringData() object) may be returned or set with Text() and SetText(). These functions provide simple accessibility to the StringData() of a ZafText, and may be used if the programmer does not wish to interact with the data portion of the object.

The return value for Text() is a pointer to the textual information in the data object of a ZafText. The return value for SetText() is normally ZAF_ERROR_NONE. The following code shows the proper use of these functions:

```
// Get the text.
const ZafIChar *text = textObject->Text();
...
// Set the new text.
textObject->SetText("New Text");
```

```
bool Unanswered(void) const;
virtual bool SetUnanswered(bool unanswered);
```

If Unanswered() is true, the ZafText field will be initially blank. When text is entered into the ZafText object, either by calling SetText(), or by the end user entering data, the Unanswered() attribute is set to false. Unanswered() is false by default, but the user may call SetUnanswered() to make changes. Calling SetUnanswered(true) will clear the contents of the text object and its StringData().

```
bool ViewOnly(void) const;
virtual bool SetViewOnly(bool viewOnly);
```

A `ViewOnly()` `ZafText` object may not be edited, but it may be the current object of a window, may be copied into the clipboard, and the arrow keys may be used to navigate it. `ViewOnly()` is false by default, but the user may call `SetViewOnly()` to change it.

```
bool WordWrap(void) const;  
virtual bool SetWordWrap(bool wrappedData);
```

A `WordWrap()` `ZafText` object automatically wraps its data at the end of each line. If `WordWrap()` is false, the end user must type a carriage return to go to the next line. `WordWrap()` is true by default, but the user may call `SetWordWrap()` to change it.

ZafTime

Event	TimeData
-------	----------

ZafTime is a single-line date object that allows user input through the keyboard. ZafTime is fully internationalized to display and input using any format.

All ZafTime objects refer to data contained in a ZafTimeData object (refer to this class for additional essential information).

Declaration

```
#include <z_time1.hpp>
```

Inheritance

```
ZafTime : ZafString : ZafWindowObject : ZafElement
```

Constructors

All ZafTime constructors initialize the member variables associated with an instantiated ZafTime object. The default values set by the ZafTime and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafTime. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafTime

TimeData()	null
------------	------

ZafString

LowerCase()	false [†]
Password()	false [†]
StringData()	null [†]
UpperCase()	false [†]
VariableName()	false [†]

ZafElement

ClassID()	ID_ZAF_TIME
ClassName()	"ZafTime"

```
ZafTime(int left, int top, int width, int hour, int
minute, int second, int milliSecond = 0);
```

This constructor is useful in straight-code situations, particularly if you wish the ZafTime object to create, maintain and destroy its own ZafTimeData object

automatically. *left*, *top* and *width* the position and size of the object on its parent. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. *hour*, *minute*, *second*, and *milliSecond* specify the time values you wish to initially appear in the new ZafTime object.

```
ZafTime(int left, int top, int width, ZafTimeData  
        *timeData = ZAF_NULLP(ZafTimeData));
```

This constructor is useful in straight-code situations where a ZafTimeData object has already been created. This constructor could be used when manually maintaining a *timeData* object, rather than having the ZafTime class create and maintain the data object automatically. For more information on using ZafTimeData objects, see the chapter on ZafTimeData. See the previous constructor for a description of *left*, *top* and *width* parameters.

```
ZafTime(const ZafTime &copy);
```

The copy constructor calls the overloaded Duplicate() to create a new ZafTime object and initialize its data from *copy*. If the original data objects are StaticData() then the new ZafTime object simply points to the original data, otherwise StaticData() copies are made.

```
ZafTime(const ZafIChar *name, ZafObjectPersistence  
        &persist);
```

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

Sample ZafTime creation techniques follow:

```
// Create a sample window with time objects.  
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);  
  
// Create time objects and pass in the values directly.  
window1->Add(new ZafTime(0, 1, 25, 8, 0, 0));  
window1->Add(new ZafTime(0, 2, 25, 17, 59, 59));  
...  
// Create a sample window with time objects.  
ZafWindow *window2 = new ZafWindow(10, 10, 40, 10);  
  
// Create time data objects.  
ZafTimeData *timeData1 = new ZafTimeData(8, 0, 0);  
ZafTimeData *timeData2 = new ZafTimeData(17, 59, 59);
```

```
// Create times that use the data previously created.
window2->Add(new ZafTime(0, 1, 25, timeData1));
window2->Add(new ZafTime(0, 2, 25, timeData2));
```

Destructor

```
virtual ~ZafTime(void);
```

The destructor is used to free the memory associated with a ZafTime object, including all the data objects that are Destroyable(). It chains to the ZafString, ZafWindowObject, and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafTime object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~~ZafWindow().

Members

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function receives all events that get sent to the ZafTime object and either handles them or passes them to ZafString, its immediate base class. See ZafWindowObject for more information.

ZafTime specifically handles the following events:

Event	Description
S_COPY_DATA	causes the object to copy event.windowObject's TimeData() if event.windowObject is a ZafTime object
S_SET_DATA	causes the object to create a new TimeData() object, then copy into it event.windowObject's TimeData() if event.windowObject is non-null and is a ZafTime object

```
ZafTimeData *TimeData(void) const;
```

```
virtual ZafError SetTimeData(ZafTimeData *timeData);
```

*TimeData() contains the actual information used by ZafTime. The TimeData() object may be used by one or more ZafTime objects, or other objects. If shared, all associated ZafTime objects will be notified when the TimeData() changes. For more information on data sharing in ZAF, see ZafDataManager.

TimeData() returns a pointer to the TimeData() object associated with the ZafTime object. The return value for SetTimeData() is normally ZAF_ERROR_NONE. See the constructor code snippet for an example using ZafTimeData objects with ZafTime.

ZafToolBar

DockType	WrapChildren
----------	--------------

The ZafToolBar object may be placed along any of the four edges of a window, and generally contains groups of buttons. However, the ZafToolBar object may contain any of ZAF’s input objects. ZafToolBar is Noncurrent() by default, meaning that it does not receive focus, but it allows the option of receiving focus if desired.

Declaration	<code>#include <z_tbar.hpp></code>
Inheritance	<code>ZafToolBar : ZafWindow : ((ZafWindowObject : ZafElement), ZafList)</code>
Constructors	All ZafToolBar constructors initialize the member variables associated with an instantiated ZafToolBar object. The default values set by the ZafToolBar and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafToolBar. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafToolBar	
<code>DockType()</code>	<code>ZAF_DOCK_TOP</code>
<code>WrapChildren()</code>	<code>true</code>
ZafWindow	
<code>Destroyable()</code>	<code>false[†]</code>
<code>Locked()</code>	<code>false[†]</code>
<code>Maximized()</code>	<code>false[†]</code>
<code>Minimized()</code>	<code>false[†]</code>
<code>Moveable()</code>	<code>false[†]</code>
<code>Sizeable()</code>	<code>false[†]</code>
<code>Temporary()</code>	<code>false[†]</code>
ZafWindowObject	
<code>AcceptDrop()</code>	<code>false[†]</code>
<code>Bordered()</code>	<code>true</code>
<code>Noncurrent()</code>	<code>true</code>
<code>RegionType()</code>	<code>ZAF_AVAILABLE_REGION[†]</code>
<code>SupportObject()</code>	<code>true</code>

Member Initializations

ZafElement

ClassID()	ID_ZAF_TOOL_BAR
ClassName()	"ZafToolBar"

ZafToolBar(int left, int top, int width, int height);

This constructor is useful in straight-code situations. *left* and *top* specify the left and top of the object. *width* and *height* specify the width and height of the object. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. Currently, the *left* and *top* parameters are ignored since the tool bar is automatically placed on the edge of its parent window's client region, but they may be used in the future. *width* and *height* parameters are ignored if the tool bar is WrapChildren().

ZafToolBar(const ZafToolBar ©);

The copy constructor is used in conjunction with the overloaded Duplicate() function. It accepts another ZafToolBar object and copies the object's information.

ZafToolBar(const ZafIChar *name, ZafObjectPersistence &persist);

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

An example of how to create a tool bar with buttons follows:

```
// Create a sample window with a tool bar and buttons.
ZafWindow *window1 = new ZafWindow(0, 0, 50, 10);
// Create a wrapping tool bar.
ZafToolBar *toolBar = new ZafToolBar(0, 0, 80, 1);
// Add two groups of buttons separated by a spacer.
// They are automatically positioned by the tool bar.
toolBar->Add(new ZafButton(0, 0, 10, 1,
    ZAF_NULLP(ZafBitmapData), "Button 1"));
toolBar->Add(new ZafButton(0, 0, 10, 1,
    ZAF_NULLP(ZafBitmapData), "Button 2"));
ZafButton *spacer = new ZafButton(0, 0, 1, 1,
    ZAF_NULLP(ZafBitmapData), ZAF_NULLP(ZafIChar));
toolBar->Add(spacer);
toolBar->Add(new ZafButton(0, 0, 10, 1,
    ZAF_NULLP(ZafBitmapData), "Button 3"));
```

```
toolBar->Add(new ZafButton(0, 0, 10, 1,  
    ZAF_NULLP(ZafBitmapData), "Button 4"));  
// Add the tool bar to the window.  
window1->Add(toolBar);
```

Destructor

```
virtual ~ZafToolBar(void);
```

The destructor is used to free the memory associated with a ZafToolBar object. It chains to the ZafWindow, ZafList, ZafWindowObject and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafToolBar object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
ZafDockType DockType(void) const;  
virtual ZafDockType SetDockType(ZafDockType dockType);
```

A ZafToolBar object may be “docked” on any of the four sides of its parent window, according to the DockType() attribute. The default value of this attribute is ZAF_DOCK_TOP, but the user may call SetDockType() to change it. The possible values of this attribute are:

- ZAF_DOCK_TOP
- ZAF_DOCK_BOTTOM
- ZAF_DOCK_LEFT
- ZAF_DOCK_RIGHT

```
bool WrapChildren(void) const;  
virtual bool SetWrapChildren(bool wrapChildren);
```

If WrapChildren() is true, the ZafToolBar object automatically positions its children based on the available space on the tool bar, and the tool bar adjusts its size on its parent window to accommodate the children. A WrapChildren() tool bar, therefore, ignores the width and height specified in the constructor. If WrapChildren() is false, the tool bar reflects the width and height specified in the constructor, and respects its children’s positioning. WrapChildren() is true by default, but the user may call SetWrapChildren() to change it.

ZafTreeItem

AutoSortData	DepthCurrent	DepthFirst
DepthLast	DepthNext	DepthPrevious
Expandable	Expanded	NormalBitmap
SelectedBitmap	SetSelectionType	StringData
Text	ToggleExpanded	TreeList
ViewCurrent	ViewFirst	ViewLast
ViewLevel	ViewNext	ViewPrevious

ZafTreeItems are contained in ZafTreeLists. Each ZafTreeItem may or may not contain other ZafTreeItems. Those that contain children may be expandable to expose or hide their children. See ZafTreeList for a more detailed description.

Declaration `#include <z_tree.hpp>`

Inheritance `ZafTreeItem : ZafWindow : ((ZafWindowObject :
 ZafElement), ZafList)`

Constructors All ZafTreeItem constructors allocate memory for an object instance and initialize member variables. The default values set by the ZafTreeItem and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafTreeItem. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafTreeItem

AutoSortData()	false
Expandable()	false
Expanded()	false
NormalBitmap()	null
SelectedBitmap()	null
StringData()	null

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Moveable()	false [†]
Sizeable()	false [†]

Member Initializations

Temporary()	false [†]
-------------	--------------------

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	false [†]
RegionType()	ZAF_INSIDE_REGION [†]
SystemObject()	false

ZafElement

ClassID()	ID_ZAF_TREE_ITEM
ClassName()	"ZafTreeItem"

```
ZafTreeItem(ZafBitmapData *normalBitmap, ZafBitmapData
             *selectedBitmap, ZafIChar *text);
```

This constructor is useful in straight-code situations, particularly if the ZafTreeItem object is to create, maintain, and destroy its own ZafStringData object automatically. *text* specifies the textual information displayed by the ZafTreeItem object. The *normalBitmap* and *selectedBitmap* parameters specify the bitmaps displayed by the ZafTreeItem object in its normal and selected states, respectively.

```
ZafTreeItem(ZafBitmapData *normalBitmap, ZafBitmapData
             *selectedBitmap, ZafStringData *stringData);
```

This constructor is also useful in straight-code situations, particularly when a ZafStringData object, *stringData*, has already been created to be associated with the ZafTreeItem object. For more information on using ZafStringData objects, see ZafStringData.

```
ZafTreeItem(const ZafTreeItem &copy);
```

The copy constructor is used in conjunction with the overloaded Duplicate() function. It allocates a new ZafTreeItem object and initializes its data from *copy*. If the original ZafTreeItem's internal data objects are StaticData() then the new ZafTreeItem object points to the originals, otherwise copies are made.

```
ZafTreeItem(const ZafIChar *name, ZafObjectPersistence
&persist);
```

The final constructor is used for persistence. Refer to `ZafWindow` for more information, since most persistence is done at the `ZafWindow` level. The following example demonstrates how to create `ZafTreeItem` objects:

```
// Create a sample window with a tree list.
ZafWindow *window1 = new ZafWindow(0, 0, 50, 10);

// Create the tree list object.
extern ZafBitmapData *nBitmap, *sBitmap;
ZafTreeList *tree = new ZafTreeList(1, 1, 20, 5);

// Create the tree items and add them to the tree.
ZafTreeItem *item1 = new ZafTreeItem(normalBitmap,
selectedBitmap, "Item 1");
item1->SetExpandable(true);
item1->Add(new ZafTreeItem(nBitmap, sBitmap, "Item 1.1"));
item1->Add(new ZafTreeItem(nBitmap, sBitmap, "Item 1.2"));
item1->Add(new ZafTreeItem(nBitmap, sBitmap, "Item 1.3"));
tree->Add(item1);
ZafTreeItem *item2 = new ZafTreeItem(nBitmap, sBitmap, "Item
2");
item2->SetExpandable(true);
item2->Add(new ZafTreeItem(nBitmap, sBitmap, "Item 2.1"));
item2->Add(new ZafTreeItem(nBitmap, sBitmap, "Item 2.2"));
item2->Add(new ZafTreeItem(nBitmap, sBitmap, "Item 2.3"));
tree->Add(item2);

// Add a vertical scroll bar to the tree list.
tree->Add(new ZafScrollBar(0, 0, 0, 0));

// Add the list to the window.
window1->Add(tree);
```

Destructor

```
virtual ~ZafTreeItem(void);
```

The destructor is used to free the memory associated with a `ZafTreeItem` object, including all the data object pieces—such as `StringData()`—that are `Destroyable()`. It chains to the `ZafWindow`, `ZafList`, `ZafWindowObject` and `ZafElement` destructors.

Generally, the programmer will not directly destroy a `ZafTreeItem` object directly, since it is automatically destroyed when its parent `ZafTreeList` is destroyed. For more information on child object deletion, see `ZafWindow::~~ZafWindow()`.

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
bool AutoSortData(void) const;
virtual bool SetAutoSortData(bool autoSortData);
```

AutoSortData() is provided at this level only for the purpose of checking the attribute, since this attribute applies for the entire hierarchy of the ZafTreeList object. SetAutoSortData() has no effect for ZafTreeItem. See ZafTreeList::AutoSortData() for more information.

```
ZafTreeItem *DepthCurrent(void);
```

DepthCurrent() returns the current leaf ZafTreeItem object in the current branch of the tree—beginning with this ZafTreeItem object and traversing down the current branch of the hierarchy. The current leaf item will not necessarily be visible.

```
ZafTreeItem *DepthFirst(void);
```

DepthFirst() returns the first ZafTreeItem object in the current sub-tree, which is always this ZafTreeItem object. This function may be used together with ZafTreeItem::DepthNext() to perform a depth traversal of the entire hierarchy of the sub-tree (beginning with this ZafTreeItem object). See ZafTreeItem::DepthFirst() for more information.

```
// Do a depth traversal of the tree.
for (ZafTreeItem *item = this; item; item = item->DepthNext())
    if (item == matchItem)
        break;
```

```
ZafTreeItem *DepthLast(void);
```

DepthLast() returns the last leaf ZafTreeItem object in the sub-tree, traversing down the last branch of the sub-tree hierarchy. The last leaf item will not necessarily be visible. This function may be used together with ZafTreeItem::DepthPrevious() to perform a reverse depth traversal of the entire hierarchy of the sub-tree. See DepthFirst() for example code.


```
ZafTreeItem *DepthNext(void);
```

DepthNext() returns the next item in a depth traversal of the sub-tree. See DepthFirst() for more information.

```
ZafTreeItem *DepthPrevious(void);
```

DepthPrevious() returns the previous item in a depth traversal of the sub-tree. See DepthFirst() and DepthLast() for more information.

```
bool Expandable(void) const;  
virtual bool SetExpandable(bool expandable);
```

An Expandable() ZafTreeItem object is considered a sub-tree of the parent ZafTreeList object and may have ZafTreeItem objects within itself. The end user will not be able to access sub-items unless the parent item is Expandable(). Expandable() is false by default.

```
bool Expanded(void) const;  
virtual bool SetExpanded(bool expanded);  
virtual bool ToggleExpanded(void);
```

The children of an Expanded() ZafTreeItem object are viewable. If Expanded() is false, the ZafTreeItem object is collapsed, and its children are not visible to the end user. This attribute is false by default, but the end user may modify it at any time, and SetExpanded() may be called to change it. ToggleExpanded() will toggle the value of this attribute, collapsing an Expanded() ZafTreeItem object, and expanding a collapsed ZafTreeItem object.

```
ZafBitmapData *NormalBitmap(void) const;  
ZafBitmapData *SelectedBitmap(void) const;  
virtual ZafError SetNormalBitmap(ZafBitmapData  
    *normalBitmap);  
virtual ZafError SetSelectedBitmap(ZafBitmapData  
    *selectedBitmap);
```

The NormalBitmap() and SelectedBitmap() objects point to the actual bitmap data. NormalBitmap() is the bitmap displayed by the ZafTreeItem object in its normal (non-selected) state, and SelectedBitmap() is the bitmap displayed by the ZafTreeItem object in its selected state.

The `NormalBitmap()` and `SelectedBitmap()` objects may be shared among several `ZafTreeItem` objects (to save memory, for example), or they may belong to a single `ZafTreeItem` object. If shared among several `ZafTreeItem` objects, all the associated `ZafTreeItem` objects will be updated when the `NormalBitmap()` or `SelectedBitmap()` objects change. For more information on data sharing in ZAF, see `ZafDataManager`.

The return value for `SetNormalBitmap()` and `SetSelectedBitmap()` is normally `ZAF_ERROR_NONE`.

```
// Set the bitmaps on a tree item if not already present.
extern ZafBitmapData *normalBitmap, *selectedBitmap;
if (!item->NormalBitmap())
    item->SetNormalBitmap(normalBitmap);
if (!item->SelectedBitmap())
    item->SetSelectedBitmap(selectedBitmap);
```

```
virtual ZafSelectionType
    SetSelectionType(ZafSelectionType selectionType);
```

This overloaded function is provided so that the entire hierarchy of `ZafTreeItem` objects will reflect the `SelectionType()` of the `ZafTreeList` object. The programmer should normally not call this function at this level, but at the `ZafTreeList` level. See `ZafTreeList::SelectionType()` for complete information.

```
ZafStringData *StringData(void) const;
virtual ZafError SetStringData(ZafStringData
    *stringData);
```

`ZafTreeItems` contain `ZafStringData` objects where their text is stored. The `StringData()` object may be shared among several `ZafTreeItem` objects (to save memory, for example), or it may belong to a single `ZafTreeItem` object. If shared, all the associated `ZafTreeItem` objects will be updated when the `StringData()` object changes. For more information on data sharing in ZAF, see `ZafDataManager`.

The return value for `SetStringData()` is normally `ZAF_ERROR_NONE`.

Refer to `Set*Bitmap()` for more information.

```
virtual const ZafIChar *Text(void);
virtual ZafError SetText(const ZafIChar *text);
```

The textual data of a `ZafTreeItem` object may be returned or set with `Text()` and `SetText()`. These functions provide simple accessibility to the `StringData()` of a `ZafTreeItem` object, and may be used if the programmer does not wish to interact with the data portion of the object.

```
virtual bool ToggleExpanded(void);
```

See `Expanded()`.

```
ZafTreeList *TreeList(void) const;
```

`TreeList()` returns a pointer to this `ZafTreeItem` object's parent `ZafTreeList` object. It returns null if this `ZafTreeItem` is not yet attached (directly or indirectly) to a `ZafTreeList` object.

```
ZafTreeItem *ViewCurrent(void);
```

The `ViewCurrent()` item in the sub-tree is the current item for the entire hierarchy of the sub-tree, beginning with this `ZafTreeItem` object.

```
ZafTreeItem *ViewFirst(void);
```

`ViewFirst()` returns a pointer to the first viewable item in the sub-tree, beginning with this `ZafTreeItem` object, which is always this `ZafTreeItem` object. In the context of `ZafTreeItem`, a viewable item may be seen by the end user by scrolling through the list without expanding any `ZafTreeItem` objects. The following code shows how to search through the viewable items in the sub-tree:

```
// Find an item in the sub-tree.  
for (ZafTreeItem *item = ViewFirst(); item; item = item->  
    ViewNext())  
    if (item->NumberID() == matchID)  
        break;
```

```
ZafTreeItem *ViewLast(void);
```

`ViewLast()` returns a pointer to the last viewable item in the sub-tree, beginning with this `ZafTreeItem` object. If this `ZafTreeItem` object is not `Expanded()`, or if there are not `ZafTreeItem` objects within this `ZafTreeItem` object, this `ZafTreeItem` object is returned. In the context of `ZafTreeItem`, a viewable item may be seen by the end user by scrolling through the list without expanding any `ZafTreeItem` objects. See `ViewFirst()` for example code.

```
int ViewLevel(void);
```

ViewLevel() returns the zero-based level at which the **ZafTreeItem** object is found. **ZafTreeItem** objects that are direct children of the parent **ZafTreeList** object are considered to be at level 0, and their children are at level 1, etc. **ViewLevel()** is used internally by the ZAF libraries for properly indenting **ZafTreeItem** objects within the parent **ZafTreeList** object.

```
ZafTreeItem *ViewNext(void);
```

ViewNext() returns the next viewable item in the sub-tree, beginning with this **ZafTreeItem** object. See **ViewFirst()** for more information.

```
ZafTreeItem *ViewPrevious(void);
```

ViewPrevious() returns the previous viewable item in the sub-tree, beginning with this **ZafTreeItem** object. See **ViewFirst** for more information.

ZafTreeList

AddDepthItem	AutoSortData	DepthCurrent
DepthFirst	DepthLast	DrawLines
SelectionType	SetBackgroundColor	SetTextColor
ViewCount	ViewCurrent	ViewFirst
ViewLast		

ZafTreeList is a scrollable hierarchical list object that intuitively presents a “tree” of list items. Each object contained in a ZafTreeList (ZafTreeItems) may or may not contain other objects. Objects that contain children may be expanded and collapsed to expose or hide contents. The ZafTreeList class utilizes the native OS tree list API if available, or is closely modeled after popular tree controls on each platform.

Declaration

#include <z_tree.hpp>

Inheritance

ZafTreeList : ZafWindow : ((ZafWindowObject : ZafElement), ZafList)

Constructors

All ZafTreeList constructors allocate memory for an object instance and initialize member variables. The default values set by the ZafTreeList and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafTreeList. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafTreeList

AutoSortData()	false
DrawLines()	true

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Moveable()	false [†]
Sizeable()	false [†]
Temporary()	false [†]

ZafWindowObject


```
item1->Add(new ZafTreeItem(nBitmap, sBitmap, "Item 1.3"));
tree->Add(item1);
ZafTreeItem *item2 = new ZafTreeItem(nBitmap, sBitmap, "Item
2");
item2->SetExpandable(true);
item2->Add(new ZafTreeItem(nBitmap, sBitmap, "Item 2.1"));
item2->Add(new ZafTreeItem(nBitmap, sBitmap, "Item 2.2"));
item2->Add(new ZafTreeItem(nBitmap, sBitmap, "Item 2.3"));
tree->Add(item2);

// Add a vertical scroll bar to the tree list.
tree->Add(new ZafScrollBar(0, 0, 0, 0));

// Add the list to the window.
window1->Add(tree);
```

Destructor

```
virtual ~ZafTreeList(void);
```

The destructor is used to free the memory associated with a ZafTreeList object. It chains to the ZafWindow, ZafList, ZafWindowObject and ZafElement destructors.

All ZafTreeItem children of a ZafTreeList are automatically destroyed when the ZafTreeList is destroyed.

Generally, the programmer will not directly destroy a ZafTreeList since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. However, if the Set*() function does not successfully change the state as requested, it will instead return the current state.

```
ZafWindowObject *AddDepthItem(const ZafIChar *pathName,
ZafWindowObject *object);
```

AddDepthItem() adds the ZafTreeItem *object* to the ZafTreeList hierarchy, using *pathName* as the parent. *pathName* is specified by stringIDs separated by the tilde ('~') character, starting with the stringID of the ZafTreeList object as the root and ending with the ZafTreeItem that will contain the new object (if any). AddDepthItem() returns a pointer to the item added, if successful, otherwise it returns null.

```
// Add Item 1.4 to the tree. "TreeList" is the StringID()
// of the tree, and "Item1" is the StringID() of the tree
```

```
// item that will contain the newly added object.
ZafTreeItem *newItem = new ZafTreeItem(normalBitmap,
    selectedBitmap, "Item 1.4");
tree->AddDepthItem("TreeList~Item1", newItem);
```

```
bool AutoSortData(void) const;
virtual bool SetAutoSortData(bool autoSortData);
```

If `AutoSortData()` is set to true, the tree list will automatically sort its children as they are added to the list. `AutoSortData()` affects the entire hierarchy of tree items, therefore each sub-tree will be independently sorted. `AutoSortData()` defaults to false.

The function returned by `CompareFunction()` is used to sort the children. By default, sorting is done in alphabetical order, but `SetCompareFunction()` may be called to provide a custom sorting function. See `ZafList::CompareFunction()` for more information about sorting list children.

```
ZafTreeItem *DepthCurrent(void);
```

`DepthCurrent()` returns the current leaf `ZafTreeItem` object in the current branch of the hierarchy. The current leaf item will not necessarily be visible.

```
ZafTreeItem *DepthFirst(void);
```

`DepthFirst()` returns the first `ZafTreeItem` object in the tree. This function may be used together with `ZafTreeItem::DepthNext()` to perform a depth traversal of the entire hierarchy of the tree list.

```
// Do a depth traversal of the entire tree.
for (ZafTreeItem *item = treeList->DepthFirst(); item; item =
    item->DepthNext())
    if (item == matchItem)
        break;
```

```
ZafTreeItem *DepthLast(void);
```

`DepthLast()` returns the last leaf `ZafTreeItem` object in the last branch of the hierarchy. The last leaf item will not necessarily be visible. This function may be used together with `ZafTreeItem::DepthPrevious()` to perform a reverse depth traversal of the entire hierarchy of the tree list. See `DepthFirst()` for sample code. See also `ZafTreeItem::DepthNext()`.


```
bool DrawLines(void) const;
virtual bool SetDrawLines(bool drawLines);
```

If DrawLines() is true, the tree list will automatically display lines between the bitmaps of its children. These lines help to visually define the hierarchical nature of the list and may differ slightly between environments as appropriate. DrawLines() is set true by default.

```
ZafSelectionType SelectionType(void) const;
virtual ZafSelectionType
    SetSelectionType(ZafSelectionType selectionType);
```

ZafTreeLists may allow different types of selection behavior. SetSelectionType() allows this behavior to be changed from the single-selection default. Valid values are listed.

SelectionType()	Description
ZAF_SINGLE_SELECTION	Allows only one ZafTreeItem to be selected. If another item is selected any previously selected item is deselected.
ZAF_MULTIPLE_SELECTION	Allows multiple ZafTreeItems to be selected. “Selection actions,” including mouse clicks, cause the selection state of an item to be toggled. The state of other tree items is unchanged.
ZAF_EXTENDED_SELECTION	Allows multiple ZafTreeItems to be selected, and does this using native multiple- and extended-selection techniques. For example, a single click might act as single-select, shift-click might select a range of items, and ctrl-click might act as multiple-select.

```
virtual ZafLogicalColor
    SetBackgroundColor(ZafLogicalColor color,
        ZafLogicalColor mono = ZAF_MONO_NULL);
```

SetBackgroundColor() specifies the background color of the ZafTreeList and the background color of all ZafTreeItems. The default color is obtained from ZafWindow.

```
virtual ZafLogicalColor SetTextColor(ZafLogicalColor
    color, ZafLogicalColor mono = ZAF_MONO_NULL);
```

To provide consistency in the appearance of tree list children, this overloaded function provides functionality for setting the text color for the entire tree list hierarchy.

```
int ViewCount(void);
```

ViewCount() returns the number of tree items that may be seen by scrolling through the list. These include top-level children of the tree list as well as viewable items in expanded branches. Internally, ZAF uses ViewCount() when calculating scroll bar values.

```
ZafTreeItem *ViewCurrent(void);
ZafTreeItem *SetViewCurrent(ZafTreeItem *item);
```

The ViewCurrent() item is the current ZafTreeItem in the current branch of the tree list. The ViewCurrent() item has focus when the tree list has focus.

```
ZafTreeItem *ViewFirst(void);
```

ViewFirst() returns a pointer to the first viewable item in the tree list, which is always the first child of the tree list. If there are no items in the tree list, null is returned. In the context of ZafTreeList, a “viewable” item may be seen by scrolling through the list without expanding any ZafTreeItem objects. The following code shows how to search through the viewable items in the ZafTreeList object:

```
// Find an item in the tree.
for (ZafTreeItem *item = ViewFirst(); item; item = item->
    ViewNext())
    if (item->NumberID() == matchID)
        break;
```

```
ZafTreeItem *ViewLast(void);
```

ViewLast() returns a pointer to the last viewable item in the tree list. If there are no items in the tree list, null is returned. See ViewFirst() for more information. See also ZafTreeItem::ViewNext().

ZafUTime

Event	UTimeData
-------	-----------

ZafUTime is a single-line date and time object that allows user input through the keyboard. ZafUTime is fully internationalized to display and input using any format.

All ZafUTime objects refer to data contained in a ZafUTimeData object (refer to this class for additional essential information).

Declaration `#include <z_utime1.hpp>`

Inheritance `ZafUTime : ZafString : ZafWindowObject : ZafElement`

Constructors All ZafUTime constructors initialize the member variables associated with an instantiated ZafUTime object. The default values set by the ZafUTime and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafUTime. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafUTime	
UTimeData()	null
ZafString	
LowerCase()	false [†]
Password()	false [†]
StringData()	null [†]
UpperCase()	false [†]
VariableName()	false [†]
ZafElement	
ClassID()	ID_ZAF_UTIME
ClassName()	"ZafUTime"

ZafUTime(int left, int top, int width, int year, int month, int day, int hour, int minute, int second, int milliSecond = 0);

This constructor is useful in straight-code situations, particularly if you wish the ZafUTime object to create, maintain and destroy its own ZafUTimeData object automatically. *left*, *top* and *width* specify the position and size of the object on its parent. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. *year*, *month*, *day*, *hour*, *minute*, *second*, and *milliSecond* specify the date and time values you wish to initially appear in the new ZafUTime object.

```
ZafUTime(int left, int top, int width, ZafUTimeData
          *utimeData = ZAF_NULLP(ZafUTimeData));
```

This constructor is useful in straight-code situations where a ZafUTimeData object has already been created. This constructor could be used when manually maintaining a *dateData* object, rather than having the ZafDate class create and maintain the data object automatically. For more information on using ZafUTimeData objects, see the chapter on ZafUTimeData. See the previous constructor for a description of *left*, *top* and *width* parameters.

```
ZafUTime(const ZafUTime &copy);
```

The copy constructor calls the overloaded Duplicate() to create a new ZafUTime object and initialize its data from *copy*. If the original data objects are StaticData() then the new ZafUTime object simply points to the original data, otherwise StaticData() copies are made.

```
ZafUTime(const ZafIChar *name, ZafObjectPersistence
          &persist);
```

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

Sample ZafUTime creation techniques follow:

```
// Create a sample window with utime objects.
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);
// Create utime objects and pass in the values directly.
window1->Add(new ZafUTime(0, 1, 25, 1984, 1, 22, 8, 0, 0));
window1->Add(new ZafUTime(0, 2, 25, 1999, 12, 31, 23, 59, 59));
...
// Create a sample window with utime objects.
ZafWindow *window2 = new ZafWindow(10, 10, 40, 10);
// Create utime data objects.
ZafUTimeData *utimeData1 = new ZafUTimeData(1984, 1, 22, 8, 0,
0);
```

```

ZafUTimeData *utimeData2 = new ZafUTimeData(1999, 12, 31, 23,
    59, 59);
// Create utimes that use the data previously created.
window2->Add(new ZafUTime(0, 1, 25, utimeData1));
window2->Add(new ZafUTime(0, 2, 25, utimeData2));

```

Destructor

```
virtual ~ZafUTime(void);
```

The destructor is used to free the memory associated with a ZafUTime object, including all data object pieces that are Destroyable(). It chains to the ZafString, ZafWindowObject, and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafUTime object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~~ZafWindow().

Members

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function receives all events that get sent to the ZafUTime object and either handles them or passes them to ZafString, its immediate base class. See ZafWindowObject for more information.

ZafUTime specifically handles the following events:

Event	Description
S_COPY_DATA	causes the object to copy event.windowObject's UTimeData() if event.windowObject is a ZafUTime object
S_SET_DATA	causes the object to create a new UTimeData() object, then copy into it event.windowObject's UTimeData() if event.windowObject is non-null and is a ZafUTime object

```

ZafUTimeData *UTimeData(void) const;
virtual ZafError SetUTimeData(ZafUTimeData *utime);

```

*UTimeData() contains the actual information used by ZafUTime. The UTimeData() object may be used by one or more ZafUTime objects, or other objects. If shared, all associated ZafUTime objects will be notified when the UTimeData() changes. For more information on data sharing in ZAF, see ZafDataManager.

UTimeData() returns a pointer to the UTimeData() object associated with the ZafUTime object. The return value for SetUTimeData() is normally ZAF_ERROR_NONE. See the constructor code snippet for an example using ZafUTimeData objects with ZafUTime.

This constructor is useful in straight-code situations. *left* and *top* specify the position where the left and top of the object will be placed on its parent. *width* and *height* specify the width and height of the object. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired.

```
ZafVtList(const ZafVtList &copy);
```

The copy constructor is used in conjunction with the overloaded Duplicate() function. It accepts another ZafVtList object and copies the object's information.

```
ZafVtList(const ZafIChar *name, ZafObjectPersistence
           &persist);
```

The final constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.

Sample ZafVtList creation techniques follow:

```
// Create a sample window with a vertical list of strings.
ZafWindow *window1 = new ZafWindow(0, 0, 50, 10);
// Create the vertical list object.
ZafVtList *vList1 = new ZafVtList(1, 1, 20, 5);
// Add a scroll bar and the strings to the vertical list.
vList1->Add(new ZafScrollBar(0, 0, 0, 0));
vList1->Add(new ZafString(0, 0, 20, "String 1", -1));
vList1->Add(new ZafString(0, 0, 20, "String 2", -1));
vList1->Add(new ZafString(0, 0, 20, "String 3", -1));
vList1->Add(new ZafString(0, 0, 20, "String 4", -1));
vList1->Add(new ZafString(0, 0, 20, "String 5", -1));
vList1->Add(new ZafString(0, 0, 20, "String 6", -1));
// Add the list to the window.
window1->Add(vList1);
...
// Create a sample window with a vertical list of buttons.
ZafWindow *window2 = new ZafWindow(10, 10, 50, 10);
// Create the vertical list object and its children.
ZafVtList *vList2 = new ZafVtList(1, 1, 20, 5);
// Allow the list children to draw bitmap information.
vList2->SetOSDraw(false);
extern ZafBitmapData *bitmap1, *bitmap2, *bitmap3, *bitmap4;
vList2->Add(new ZafButton(0, 0, 20, 1, bitmap1,
                        ZAF_NULLP(ZafIChar)));
vList2->Add(new ZafButton(0, 0, 20, 1, bitmap2,
                        ZAF_NULLP(ZafIChar)));
```



```
vList2->Add(new ZafButton(0, 0, 20, 1, bitmap3,  
    ZAF_NULLP(ZafIChar)));  
vList2->Add(new ZafButton(0, 0, 20, 1, bitmap4,  
    ZAF_NULLP(ZafIChar)));  
// Add the list to the window.  
window2->Add(vList2);
```

Destructor

```
virtual ~ZafVtList(void);
```

The destructor is used to free the memory associated with a ZafVtList object. It chains to the ZafWindow, ZafList, ZafWindowObject and ZafElement destructors.

Generally, the programmer will not directly destroy a ZafVtList object, since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion, see ZafWindow::~ZafWindow().

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*() function. If the Set*() function does not successfully change the state as requested, however, it will instead return the current state.

```
bool AutoSortData(void) const;  
virtual bool SetAutoSortData(bool autoSortData);
```

If AutoSortData() is true, the list will automatically sort its children as they are added to the list. The function returned by CompareFunction() is used to sort the children. By default, sorting is done in alphabetical order, but SetCompareFunction() may be called to provide a custom sorting function. See ZafList::CompareFunction() for more information about sorting list children. The default value of this attribute is false, but the user may call SetAutoSortData() to change it.

```
virtual ZafLogicalColor  
    SetBackgroundColor(ZafLogicalColor color,  
        ZafLogicalColor mono = ZAF_MONO_NULL);
```

To provide consistency in the appearance of list children, the ZafVtList object sets the ParentPalette() attribute on each of its children (see ZafWindowObject::ParentPalette()). In conjunction with this attribute, this overloaded function provides functionality for setting the background color for all the children in the list.

```

ZafSelectionType SelectionType(void) const;
virtual ZafSelectionType
    SetSelectionType(ZafSelectionType selectionType);

```

ZafVtLists may allow different types of selection behavior.

SetSelectionType() allows this behavior to be changed from the single-selection default. Valid values are listed.

SelectionType()	Description
ZAF_SINGLE_SELECTION	Allows only one item to be selected. If another item is selected any previously selected item is deselected.
ZAF_MULTIPLE_SELECTION	Allows multiple items to be selected. “Selection actions,” including mouse clicks, cause the selection state of an item to be toggled. The state of other list items is unchanged.
ZAF_EXTENDED_SELECTION	Allows multiple items to be selected, and does this using native multiple- and extended-selection techniques. For example, a single click might act as single-select, shift-click might select a range of items, and ctrl-click might act as multiple-select.

```

virtual ZafLogicalFont SetFont(ZafLogicalFont font);

```

To provide consistency in the appearance of list children, the ZafVtList object sets the ParentPalette() attribute on each of its children (see ZafWindowObject::ParentPalette()). In conjunction with this attribute, this overloaded function provides functionality for setting the font for all the children in the list.

```

virtual ZafLogicalColor SetTextColor(ZafLogicalColor
    color, ZafLogicalColor mono = ZAF_MONO_NULL);

```

To provide consistency in the appearance of list children, the ZafVtList object sets the ParentPalette() attribute on each of its children (see ZafWindowObject::ParentPalette()). In conjunction with this attribute, this overloaded function provides functionality for setting the text color for all the children in the list.

ZafWindow

Add	AddGenericObject	AutomaticUpdate
Border	BroadcastEvent	Changed
ClientRegion	CompareAscending	CompareDescending
CornerScrollBar	DefaultButton	Destroy
Destroyable	Event	FocusObject
GeometryManager	HorizontalScrollBar	Locked
MaximizeButton	Maximized	MinimizeButton
Minimized	MinimizeIcon	Moveable
PullDownMenu	SelectionType	Sizeable
Subtract	support	SupportCurrent
SupportDestroy	SupportFirst	SupportLast
SystemButton	SystemButtonMenu	Temporary
Text	Title	VerticalScrollBar
operator +	operator -	

ZafWindow defines the basic functionality necessary to display groups of objects on the screen. As with all other ZAF classes, the ZafWindow class utilizes the native window API if available, so the look-and-feel is exactly what the end user expects. In fact, ZAF ties into the native API so closely that system-wide modifications made by the end user are reflected in ZAF windows (such as color and font schemes), unless a user-defined palette has been specified. See `ZafWindowObject::UserPaletteData()` for more information.

Windows provided by the native environment generally have automatic support for decorations such as borders, title bars, system buttons, minimize buttons, and maximize buttons. These decorations may be added to a ZafWindow object with the following classes: `ZafBorder`, `ZafTitle`, `ZafSystemButton`, `ZafMinimizeButton`, and `ZafMaximizeButton`. However, a ZafWindow object with any of these decorations may not be a proper child window (in other words, a non-MDI child window). Only top-level windows added to the `ZafWindowManager` object may have decorations, with the exception of the `ZafMDIWindow` class. See `ZafMDIWindow` for more information.

ZafWindow is a convenient base class for other classes that maintain child objects. Therefore, many of the classes in ZAF such as `ZafGroup`, `ZafTreeList`, and `ZafVtList` derive from ZafWindow. ZAF also provides many variations of window classes based upon ZafWindow, such as `ZafDialogWindow`, `ZafMDIWindow`, and `ZafScrolledWindow`.

Refer to this section of this manual to answer questions regarding the general operation of window classes.

Declaration

```
#include <z_win.hpp>
```

Inheritance

ZafWindow : ((ZafWindowObject : ZafElement), ZafList)

Constructors

All ZafWindow constructors initialize the member variables associated with an instantiated ZafWindow object. The default values set by the ZafWindow and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafWindow. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations**ZafWindow**

DefaultButton()	null
Destroyable()	true
Locked()	false
Maximized()	false
Minimized()	false
Moveable()	true
SelectionType()	ZAF_SINGLE_SELECTION
Sizeable()	true
Temporary()	false

ZafWindowObject

CopyDraggable()	false [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
ParentDrawBorder()	false [†]
ParentDrawFocus()	false [†]
Region()	ZAF_CELL, left, top, left + width - 1, top + height - 1
Selected()	false [†]

ZafElement

ClassID()	ID_ZAF_WINDOW
ClassName()	“ZafWindow”

ZafWindow(int left, int top, int width, int height);

The first constructor is useful in straight-code situations. The four parameters *left*, *top*, *width*, and *height* define the absolute size and position of the window. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. A root window (one added to the ZafWindowManager object) positions itself on the screen such that the *left* and *top* parameters specify the left and top coordinates of its client region, respec-

tively. A child window's position is relative to the top-left corner of its parent's client region.

```
ZafWindow(const ZafWindowObject &copy);
```

This is the copy constructor which accepts another `ZafWindow`, *copy*, and copies the object's information. This constructor is used in conjunction with the overloaded `Duplicate()` function.

```
ZafWindow(const ZafIChar *name, ZafObjectPersistence  
          &persist);
```

The final constructor is used for persistence. *name* specifies the name of the window to be read from a persistent file. *persist* contains persistent information such as a pointer to the file-system and object constructors—both necessary for object creation. The following examples demonstrate how to create a window in code and how to use persistence in general window creation:

```
// Create a sample window.  
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);  
// Add the decorations to the window.  
window1->Add(new ZafBorder);  
window1->Add(new ZafMaximizeButton);  
window1->Add(new ZafMinimizeButton);  
window1->Add(new ZafSystemButton);  
window1->Add(new ZafTitle("Sample Window"));  
// Add a button to the window.  
window1->Add(new ZafButton(2, 4, 15, 1,  
          ZAF_NULLP(ZafBitmapData), "Button"));  
// Put the window on the screen.  
windowManager->Add(window1);  
  
// Open the data file.  
ZafStorage *storage = new ZafStorage("myfile.dat");  
// Create the persistence object.  
ZafObjectPersistence persist(storage);  
// Load the persistent window "MyWindow".  
windowManager->Add(new ZafWindow("MyWindow", persist));
```

Destructor

```
virtual ~ZafWindow(void);
```

The destructor is used to free the memory associated with a `ZafWindow` object, including all the attached child objects and all the data object pieces that are `Destroyable()`. It chains to the `ZafWindowObject`, `ZafElement`, and `ZafList` destructors.

Since this destructor chains to the ZafList destructor, all the attached child objects are also subtracted from the window and destroyed. This is convenient for the programmer, since child objects need not be individually destroyed. See Subtract() for more information.

Generally, the programmer will not directly destroy a ZafWindow object, since it is automatically destroyed when it is removed from the window manager, or when the parent window is destroyed. The following code provides an example:

```
// Create a sample window.
ZafWindow *window1 = new ZafWindow(2, 2, 40, 10);
// Add the decorations to the window.
window1->Add(new ZafBorder);
window1->Add(new ZafMaximizeButton);
window1->Add(new ZafMinimizeButton);
window1->Add(new ZafSystemButton);
window1->Add(new ZafTitle("Sample Window"));
// Add a button to the window.
window1->Add(new ZafButton(2, 4, 15, 1,
    ZAF_NULLP(ZafBitmapData), "Button"));
...
// Destroy the window and all its children.
delete window1;
```

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the Set*(*) function. However, if the Set*(*) function does not successfully change the state as requested, it will instead return the current state.

```
virtual ZafWindowObject *Add(ZafWindowObject *object,
    ZafWindowObject *position =
    ZAF_NULLP(ZafWindowObject));
ZafWindow &operator+(ZafWindowObject *object);
```

This function and operator overload base ZafList::Add() functionality to handle advanced addition operations typical of derived ZafWindow classes. For instance, ZafVtList has widely different implementations on the Microsoft Windows and Motif platforms. On Windows, the objects are represented internally by the OS and the only interface is accomplished through LB_* messages. On Motif, there is no widget representation at all! Thus the exact handling, insertion and updating of an object is performed uniquely by each overloaded Add() function.

For objects added to ZafWindow, these three operations are performed:

- they are added to the list of support or non-support children (either the support member or the base `ZafList` part of the class, respectively)
- their parent pointer is set to point to the `ZafWindow` object
- they are updated on the screen if the parent window is already visible to the user

This overloaded function and operator return a typesafe `ZafWindowObject` pointer, which is generally the object that was passed to the `Add()` function, but can be null if the object either currently resides within another window or is an invalid type of `Add()` operation. An invalid add operation would be adding a `ZafTable` object to a `ZafPullDownMenu`.

The following code demonstrates correct use of this function and operator:

```
// Add support children to a window.
ZafWindow *window1 = new ZafWindow(2, 2, 50, 10);
window1->Add(new ZafBorder);
window1->Add(new ZafMaximizeButton);
window1->Add(new ZafMinimizeButton);
window1->Add(new ZafSystemButton);
window1->Add(new ZafTitle("String Window"));

// Do the same thing with the + operator.
ZafWindow *window2 = new ZafWindow(3, 3, 50, 10);
*window2
+ new ZafBorder
+ new ZafMaximizeButton
+ new ZafMinimizeButton
+ new ZafSystemButton
+ new ZafTitle("String Window");

ZafWindow *AddGenericObjects(ZafStringData *title,
                             ZafWindowObject *minObject =
                             ZAF_NULLP(ZafWindowObject));
```

`AddGenericObjects()` may be called on a `ZafWindow` to add all the common decorations to it. The decorations included on a window created with `AddGenericObjects()` are `ZafBorder`, `ZafTitle`, `ZafMaximizeButton`, `ZafMinimizeButton`, `ZafSystemButton`, and optionally a minimize icon (`ZafIcon`). *title* specifies the text to be placed in the title bar. *minObject*, if non-null, specifies the `ZafIcon` object to be used when the window is minimized. A pointer to the window is returned. The following code shows the proper use of this function:

```
// Create a window with all the normal decorations on it.
ZafWindow *window = new ZafWindow(2, 2, 40, 10);
window->AddGenericObjects(new ZafStringData("Test Window"));
```

```
// Add the window to the window manager.
windowManager->Add(window);
```

```
virtual bool AutomaticUpdate(void) const;
virtual bool SetAutomaticUpdate(bool automaticUpdate);
```

This function inherits all the functionality of the base `ZafWindowObject::SetAutomaticUpdate()` function and adds functionality that optimizes the addition and subtraction of child objects. Frequently, when either adding or subtracting numerous child objects, it is desirable to have only a single visible update to the screen. For example, a `ZafVtList` that contains the names of all files in the current directory, when moving up a directory should present the new files in one clean refresh. This is done by calling `SetAutomaticUpdate(false)`, making the list changes, then resetting the automatic update to true. The following code shows how this can be done:

```
// Refresh the list with new files.
void MyFileList::RedoList(void)
{
    // Destroy the old elements in the list.
    SetAutomaticUpdate(false);
    Destroy();

    // Add new list elements
    for (FileElement *element = fileSystem->FindFirst("");
         element;
         element = fileSystem->FindNext());
        Add(new ZafString(0, 0, 20, element->Text(), -1));

    // Update the list's presentation.
    SetAutomaticUpdate(true);
}
```

```
ZafBorder *Border(void);
```

This function returns a pointer to a `ZafBorder` child object, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A `ZafBorder` object will exist only if one of the following conditions have been met:

- it has previously been added with the `ZafWindow::Add()` function
- it has been added by sending an `S_ADD_OBJECT` message with `event.windowObject` pointing to an instantiated `ZafBorder` object
- it has been added by the `AddGenericObjects()` function

- the window was constructed using the persistent constructor and a ZafBorder object was loaded as a child object

Otherwise, this function returns a null pointer.

```
virtual void BroadcastEvent(const ZafEventStruct &event);
```

This is a convenience function that dispatches an event to all the window's normal and support children. Typically, this function is used to propagate a particular event to all of the children associated with a derived window. For instance, the following code shows how an event can be passed to a ZafWindow object's children, and also how the BroadcastEvent() function works:

```
ZafEventType MyWindow::Event(const ZafEventStruct &event)
{
    ZafEventType ccode = event.type;
    if (ccode == MY_EVENT)
        BroadcastEvent(event);
    ccode = ZafWindowObject::Event(event);
    return (ccode);
}

void ZafWindow::BroadcastEvent(const ZAF_EVENT_STRUCT &event)
{
    // Broadcast the event to all children.
    ZafWindowObject *object;
    for (object = SupportFirst(); object; object = object->Next())
        object->Event(event);
    for (object = First(); object; object = object->Next())
        object->Event(event);
}
```

Notice that the return values from the childrens' Event() functions are ignored. Thus, to assess the current state of a given operation, rather than calling BroadcastEvent(), dispatch the events directly to the objects. The following example shows how this may be accomplished:

```
ZafEventType MyWindow::Broadcast(const ZAF_EVENT_STRUCT &event)
{
    // Broadcast the event to all normal children.
    ZafEventType ccode = event.type;
    for (ZafWindowObject *object = First(); object; object =
        object->Next())
    {
        ccode = object->Event(event);
    }
}
```

```

        if (ccode != event.type)
            break; // an error occurred.
    }
    return (ccode);
}

```

```

virtual bool Changed(void) const;
virtual bool SetChanged(bool changed);

```

These functions inherit all the features of `ZafWindowObject::Changed()` and `ZafWindowObject::SetChanged()`, but also add a search mechanism on the window's children. In particular, the `Changed()` function traverses all its normal children to determine if any has the “`Changed() == true`” setting. If any child's setting is true, the function returns true.

`SetChanged()` overrides the base functionality by not only resetting the window's changed status, but also by clearing the children's `Changed()` values if the argument passed is false. This in effect is a “Change All” command, but only acts as such if the argument passed is false. If the argument is true, the window resets its internal value, but does not propagate that change to all of its children. The following example demonstrates correct use of these functions:

```

// Check for a changed window.
if (window->Changed())
{
    WriteInfoToDisk(window);
    window->SetChanged(false); // clears all the children.
}

```

```

ZafRegionStruct ClientRegion(void) const;

```

The client region of a window is the area inside all the decorations and support objects. For example, after a border, a title bar, a tool bar, and a status bar are added to a window, the remaining region in the window is referred to as the window's `ClientRegion()`. Any normal child added to the window, such as a string field, is placed relative to the top-left of the window's `ClientRegion()`. In other words, the `ClientRegion()` of the window is made smaller by each support object added to the window.

Note that `ClientRegion()` returns the region defined by the native environment as the window's client region, and thus may only be used portably for determining the size of the window's client region. This is an advanced function used internally by the ZAF libraries, and should normally not be called by the programmer. Child objects are automatically placed relative to the client

region when added to the window, so the programmer should never be concerned with the position of the ClientRegion().

```
static int CompareAscending(ZafWindowObject *object1,  
                             ZafWindowObject *object2);  
static int CompareDescending(ZafWindowObject *object1,  
                               ZafWindowObject *object2);
```

These two functions provide sort methods for derived ZafWindow classes such as ZafVtList and ZafHzList to allow the insertion of list items in ascending or descending collation sequences. The functions are bound directly to ZafList::CompareFunction() whenever the SetAutoSort(true) function is called for these derived classes. In general, these functions should not be used with ZafWindow, but they are available for classes derived from ZafWindow that require sorting functionality. The following sample code shows how to set the list pointer to these functions:

```
void MyWindow::DoTheSort(bool sortAscending)  
{  
    // Reset the sort function.  
    if (sortAscending)  
        SetCompareFunction((ZafCompareFunction)CompareAscending);  
    else  
        SetCompareFunction((ZafCompareFunction)CompareDescending);  
    // Sort the children.  
    Sort();  
}
```

```
ZafScrollBar *CornerScrollBar(void);
```

This function returns a pointer to a corner ZafScrollBar child object, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A corner ZafScrollBar object will exist only if one of the following conditions have been met:

- it has previously been added with the ZafWindow::Add() function
- it has been added by sending an S_ADD_OBJECT message with event.windowObject pointing to an instantiated corner ZafScrollBar object
- the window was constructed using the persistent constructor and a corner ZafScrollBar object was loaded as a child object

Otherwise, this function returns a null pointer.

```
ZafButton *DefaultButton(void) const;
ZafButton *SetDefaultButton(ZafButton *object);
```

A default button is the button that will be selected when the user types <Enter> or <Return>. The default value of this attribute is null, but the user may call SetDefaultButton() to set a default button for a window. SetDefaultButton() tells the parent window which button is to function as the default button, but does not set the ZafButton::AllowDefault() attribute on the button, which causes the button to be visually indicated as the default button. DefaultButton() affects which button functions as the default button on the window, and the ZafButton::AllowDefault() attribute affects the visual aspect of a button. Refer to ZafButton::AllowDefault() for more information.

The following code provides a brief example:

```
// Create the OK button as the default button.
ZafButton *button1 = new ZafButton(1, 4, 12, 1,
    ZAF_NULLP(ZafBitmapData), "OK");
button1->SetAllowDefault(true);
window->Add(button1);
window->SetDefaultButton(button1);
```

```
virtual void Destroy(void);
```

This function inherits features of the base ZafList::Destroy() function and adds functionality to redisplay the window, if visible on the screen, and to delete OS specific references to the children if they exist. This is a destructive call that not only removes children from the window's list, but also deletes them.

Note, this function does not remove any of the window's support children. This must be done with a call to SupportDestroy().

```
bool Destroyable(void) const;
virtual bool SetDestroyable(bool destroyable);
```

If Destroyable() is true, the window is considered non-static and is maintained by ZAF. In other words, when the window is closed (either by the end user, or by the programmer), it is destroyed with the delete operator, causing its destructor to be called. On the other hand, if Destroyable() is false, ZAF will not destroy the window, and the programmer assumes responsibility for it. The default value of this attribute is true, but it may be changed by calling SetDestroyable().

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function handles all events that are sent to the ZafWindow object, either by processing the events itself, or by passing the event down for base class processing. Refer to ZafWindowObject::Event() for complete details. In addition to events handled by its base classes, the following are handled by ZafWindow:

Event type	Description
S_ADD_OBJECT	Causes event.windowObject to be added as a child
S_MAXIMIZE	Causes the window to be maximized
S_MINIMIZE	Causes the window to be minimized
S_RESTORE	Causes a maximized or minimized window to be restored
S_SUBTRACT_OBJECT	Causes event.windowObject to be subtracted as a child
N_CLOSE	Notifies that the window is about to be closed
L_NEXT	Causes the next child to receive focus
L_PREVIOUS	Causes the previous child to receive focus

```
virtual ZafWindowObject *FocusObject(void);
```

FocusObject() returns the leaf object that has focus, the object that receives keyboard events. If there is no object with focus, such as when all the objects on a window are either Disabled() or Noncurrent(), FocusObject() returns null.

```
ZafGeometryManager *GeometryManager(void);
```

This function returns a pointer to a ZafGeometryManager child object, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A ZafGeometryManager object will exist only if one of the following conditions have been met:

- it has previously been added with the ZafWindow::Add() function
- it has been added by sending an S_ADD_OBJECT message with event.windowObject pointing to an instantiated ZafGeometryManager object
- the window was constructed using the persistent constructor and a ZafGeometryManager object was loaded as a child object

Otherwise, this function returns a null pointer.

```
ZafScrollBar *HorizontalScrollBar(void);
```

This function returns a pointer to a horizontal ZafScrollBar child object, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A horizontal ZafScrollBar object will exist only if one of the following conditions have been met:

- it has previously been added with the ZafWindow::Add() function
- it has been added by sending an S_ADD_OBJECT message with event.windowObject pointing to an instantiated horizontal ZafScrollBar object
- the window was constructed using the persistent constructor and a horizontal ZafScrollBar object was loaded as a child object

Otherwise, this function returns a null pointer.

```
bool Locked(void) const;
virtual bool SetLocked(bool locked);
```

If Locked() is true, the window may not be closed, but is to remain on the screen. A Locked() window's system button will reflect the fact that the end user may not close the window by disabling the appropriate menu item, if it exists. However, in some environments the window may be closed down unexpectedly by the environment. If Locked() is false, the window may be closed normally. The default value of this attribute is false, but it may be changed by calling SetLocked().

```
ZafMaximizeButton *MaximizeButton(void);
```

This function returns a pointer to a ZafMaximizeButton child object, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A ZafMaximizeButton object will exist only if one of the following conditions have been met:

- it has previously been added with the ZafWindow::Add() function
- it has been added by sending an S_ADD_OBJECT message with event.windowObject pointing to an instantiated ZafMaximizeButton object
- it has been added by the AddGenericObjects() function
- the window was constructed using the persistent constructor and a ZafMaximizeButton object was loaded as a child object

Otherwise, this function returns a null pointer.

```
bool Maximized(void) const;  
virtual bool SetMaximized(bool maximized);
```

If `Maximized()` is true, the window is presented on the screen as maximized, otherwise the window's position and size are specified by `Region()`. The default value of this attribute is false, but it may be changed at any time by calling `SetMaximized()`.

To make the window appear maximized when it first shows up on the screen, call `SetMaximized(true)` before adding the window to the window manager. Later calling `SetMaximized(false)` restores the window to its normal position and size. Both `Maximized()` and `Minimized()` should never be set to true at the same time, as the result is undefined.

```
ZafMinimizeButton *MinimizeButton(void);
```

This function returns a pointer to a `ZafMinimizeButton` child object, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A `ZafMinimizeButton` object will exist only if one of the following conditions have been met:

- it has previously been added with the `ZafWindow::Add()` function
- it has been added by sending an `S_ADD_OBJECT` message with `event.windowObject` pointing to an instantiated `ZafMinimizeButton` object
- it has been added by the `AddGenericObjects()` function
- the window was constructed using the persistent constructor and a `ZafMinimizeButton` object was loaded as a child object

Otherwise, this function returns a null pointer.

```
bool Minimized(void) const;  
virtual bool SetMinimized(bool minimized);
```

If `Minimized()` is true, the window is presented on the screen as minimized, represented on the screen only by a minimize icon (or something similar), otherwise the window's position and size specified by `Region()`. The default value of this attribute is false, but it may be changed at any time by calling `SetMinimized()`.

To make the window appear minimized when it first shows up on the screen, call `SetMinimized(true)` before adding the window to the window manager. Later calling `SetMinimized(false)` restores the window to its normal position and size. Both `Maximized()` and `Minimized()` should never be set to true at the same time, as the result is undefined.

```
ZafIcon *MinimizeIcon(void);
```

This function returns a pointer to a minimize ZafIcon child object, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A minimize ZafIcon object will exist only if one of the following conditions have been met:

- it has previously been added with the ZafWindow::Add() function
- it has been added by sending an S_ADD_OBJECT message with event.windowObject pointing to an instantiated minimize ZafIcon object
- it has been added by the AddGenericObjects() function, and a minimize ZafIcon object was specified
- the window was constructed using the persistent constructor and a minimize ZafIcon object was loaded as a child object

Otherwise, this function returns a null pointer.

```
bool Moveable(void) const;
virtual bool SetMoveable(bool moveable);
```

If Moveable() is true, the window may be moved around on the screen by the end user. A non-Moveable() window's system button will reflect the fact that the end user may not move the window by disabling the appropriate menu item, if it exists. The default value of this attribute is true, but it may be changed by calling SetMoveable().

```
ZafPullDownMenu *PullDownMenu(void);
```

This function returns a pointer to a ZafPullDownMenu child object, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A ZafPullDownMenu object will exist only if one of the following conditions have been met:

- it has previously been added with the ZafWindow::Add() function
- it has been added by sending an S_ADD_OBJECT message with event.windowObject pointing to an instantiated ZafPullDownMenu object
- the window was constructed using the persistent constructor and a ZafPullDownMenu object was loaded as a child object

Otherwise, this function returns a null pointer.


```
ZafSelectionType SelectionType(void) const;  
virtual ZafSelectionType  
    SetSelectionType(ZafSelectionType selectionType);
```

These functions set the selection parameters for a window's children as they operate in the context of their parent. There are three types of selection operations permitted with windows:

Selection type	Description
ZAF_SINGLE_SELECTION	Allows just one child object to be selected at a time
ZAF_MULTIPLE_SELECTION	Allows zero or more child objects to be selected at a time
ZAF_EXTENDED_SELECTION	Allows zero or more child objects to be selected at a time, with extended environment-specific selection rules

The following code shows the proper use of these functions:

```
// Get the attribute  
if (window->SelectionType() == ZAF_SINGLE_SELECTION)  
    break;  
  
// Set the attribute for a new vertical list.  
ZafVtList *vtList = new ZafVtList(0, 0, 50, 10);  
vtList->SetSelectionType(ZAF_MULTIPLE_SELECTION);
```

```
bool Sizeable(void) const;  
virtual bool SetSizeable(bool sizeable);
```

If `Sizeable()` is true, the window may be sized by the end user. A non-`Sizeable()` window's system button will reflect the fact that the end user may not size the window by disabling the appropriate menu item, if it exists. The default value of this attribute is true, but it may be changed by calling `SetSizeable()`.

```
virtual ZafWindowObject *Subtract(ZafWindowObject  
    *object);  
ZafWindow &operator-(ZafWindowObject *object) {  
    Subtract(object); return(*this); }
```

This function and operator overload base `ZafList::Subtract()` functionality to handle advanced subtraction operations typical of derived `ZafWindow` classes.

For example, ZafVtList has widely different implementations on the Microsoft Windows and Motif platforms. On Windows, the objects are represented internally by the OS and the only interface is accomplished through LB_* messages. On Motif, there is no widget representation at all! Thus the exact handling, deletion and updating of an object is performed uniquely by each overloaded Subtract() function.

For objects subtracted from ZafWindow, these three operations are performed:

- they are subtracted from the list of support or non-support children (either the support member or the base ZafList part of the class, respectively)
- their parent pointer is cleared
- they are removed from the screen if the parent window is already visible to the user

This overloaded function and operator return a typesafe ZafWindowObject pointer. This is generally the object that was passed to the Subtract() function, but can be null if the object isn't a child of the window.

The following code demonstrates correct use of this function and operator:

```
// Subtract children from a window.
window1->Subtract(string1);
window1->Subtract(string2);
window1->Subtract(button1);
window1->Subtract(button2);

// Do the same thing with the - operator.
*window2
- new string1
- new string2
- new button1
- new button2;

ZafList support;
ZafWindowObject *SupportCurrent(void) const;
void SupportDestroy(void);
ZafWindowObject *SupportFirst(void) const;
ZafWindowObject *SupportLast(void) const;
```

The support list of a ZafWindow object maintains a list of support children whose purposes are simply to support the ZafWindow object. Some support objects commonly used on a ZafWindow object are ZafBorder, ZafTitle, and ZafScrollBar. Typical decorations of a ZafWindow object reside in its support list. Operations on support objects are redirected to the parent ZafWindow

object. For example, dragging a ZafTitle object has the effect of moving the parent window.

Just as Current(), Destroy(), First(), and Last() operate on the normal children of a window, SupportCurrent(), SupportDestroy(), SupportFirst(), and SupportLast() operate on the support children of a window. SupportCurrent() returns the current support child, SupportDestroy() destroys all the children in the support list, SupportFirst() returns the first child in the support list, and SupportLast() returns the last child in the support list. The following example demonstrates correct use of some these functions:

```
// Search for the title bar in the support list.
for (ZafWindowObject *object = SupportFirst(); object; object =
    object->Next())
    if (object->IsA(ID_ZAF_TITLE))
        break;
...
// Destroy the support list.
SupportDestroy();
```

```
ZafSystemButton *SystemButton(void);
```

This function returns a pointer to a ZafSystemButton child object, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A ZafSystemButton object will exist only if one of the following conditions have been met:

- it has previously been added with the ZafWindow::Add() function
- it has been added by sending an S_ADD_OBJECT message with event.windowObject pointing to an instantiated ZafSystemButton object
- it has been added by the AddGenericObjects() function
- the window was constructed using the persistent constructor and a ZafSystemButton object was loaded as a child object

Otherwise, this function returns a null pointer.

```
ZafPopupMenu *SystemButtonMenu(void);
```

This function returns a pointer to a ZafSystemButton child object's menu member, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A ZafSystemButton object will exist only if one of the following conditions have been met:

- it has previously been added with the `ZafWindow::Add()` function
- it has been added by sending an `S_ADD_OBJECT` message with `event.windowObject` pointing to an instantiated `ZafSystemButton` object
- it has been added by the `AddGenericObjects()` function
- the window was constructed using the persistent constructor and a `ZafSystemButton` object was loaded as a child object

Otherwise, this function returns a null pointer.

```
bool Temporary(void) const;
virtual bool SetTemporary(bool temporary);
```

A `Temporary()` window will be removed from the screen under any of the following circumstances:

- An `S_CLOSE_TEMPORARY` is received by the window manager
- The end user clicks the mouse on another window
- The <Escape> key is pressed, if the window is a pop-up menu
- A menu item is selected, if the window is a pop-up menu

An example of a temporary window is a pop-up menu, which closes when the end user makes a selection. If `Temporary()` is false, the window behaves as a normal window. The default value of this attribute is false, but it may be changed by calling `SetTemporary()`.

```
virtual const ZafIChar *Text(void);
virtual ZafError SetText(const ZafIChar *text);
```

These functions overload the base `ZafWindowObject::Text()` and `ZafWindowObject::SetText()` functions by returning or changing the window's title bar, if there is one. If the `ZafWindow` object contains a child `ZafTitle` object, the *text* parameter of `SetText()` is passed to this child object, reflecting changes to the title bar portion of the window, and `Text()` returns the textual information from the title bar. If no `ZafTitle` object exists, these functions return null.

```
ZafTitle *Title(void);
```

This function returns a pointer to a `ZafTitle` child object, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A `ZafTitle` object will exist only if one of the following conditions have been met:

- it has previously been added with the `ZafWindow::Add()` function
- it has been added by sending an `S_ADD_OBJECT` message with `event.windowObject` pointing to an instantiated `ZafTitle` object
- it has been added by the `AddGenericObjects()` function
- the window was constructed using the persistent constructor and a `ZafTitle` object was loaded as a child object

Otherwise, this function returns a null pointer.

```
ZafScrollBar *VerticalScrollBar(void);
```

This function returns a pointer to a vertical `ZafScrollBar` child object, if the instantiated object exists in the window's list of support children (refer to the support section of this chapter for more information about support children). A vertical `ZafScrollBar` object will exist only if one of the following conditions have been met:

- it has previously been added with the `ZafWindow::Add()` function
- it has been added by sending an `S_ADD_OBJECT` message with `event.windowObject` pointing to an instantiated vertical `ZafScrollBar` object
- the window was constructed using the persistent constructor and a vertical `ZafScrollBar` object was loaded as a child object

Otherwise, this function returns a null pointer.

```
ZafWindow &operator+(ZafWindowObject *object);
```

See `Add()`.

```
ZafWindow &operator-(ZafWindowObject *object);
```

See `Subtract()`.

ZafWindowManager

Center	DragObject	Event
ExitFunction	helpObject	mouseObject

ZafWindowManager is the top-level class used to manage all the windows on the screen. To cause the window manager to manage a window, simply add the window to the window manager's list with the Add() function. The window will then appear on the screen (unless the window's Visible() attribute is false). The window manager allows no interaction with the user, but transparently manages the windows on the screen.

Declaration `#include <z_win.hpp>`

Inheritance `ZafWindowManager : ZafWindow : ((ZafWindowObject : ZafElement), ZafList)`

Constructor The ZafWindowManager constructor initializes the member variables associated with an instantiated ZafWindowManager object. The default values set by the ZafWindowManager and its base class constructors follow, if they differ from those set by the base class constructor, or if a blocking function is implemented in ZafWindowManager. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafWindowManager

exitFunction	null
dragObject	null
helpObject	null
mouseObject	null
ZafWindow	
SelectionType()	ZAF_SINGLE_SELECTION [†]
Temporary()	false [†]

ZafWindowObject

AcceptDrop()	false [†]
AutomaticUpdate()	true [†]
Bordered()	false [†]
OSDraw()	true [†]
ParentDrawBorder()	false [†]
ParentDrawFocus()	false [†]
ParentPalette()	false [†]

Member Initializations

<code>Region()</code>	<code>ZAF_PIXEL, 0, 0, display->columns - 1, display->lines - 1</code>
<code>RegionType()</code>	<code>ZAF_AVAILABLE_REGION[†]</code>

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_WINDOW_MANAGER</code>
<code>ClassName()</code>	<code>"ZafWindowManager"</code>

```
ZafWindowManager(ZafExitFunction exitFunction =  
    ZAF_NULLF(ZafExitFunction));
```

This constructor should normally not be called by the programmer, since it is called by the `ZafApplication` constructor. *exitFunction* specifies the function to be called when the application is about to close down (see `ExitFunction()` for more information). Static members of `ZafWindowObject`, including `display`, `eventManager` and `windowManager`, are initialized in this constructor. The mouse cursor is also initialized to `DM_VIEW` in this constructor (the default pointer).

Destructor

```
virtual ~ZafWindowManager(void);
```

This destructor is used to free the memory associated with a `ZafWindowManager` object. It chains to the `ZafWindow`, `ZafList`, `ZafWindowObject` and `ZafElement` destructors, first removing all windows from the screen and then deleting them, if they are `Destroyable()`.

Generally, the programmer will not directly destroy a `ZafWindowManager` object, since it is automatically destroyed when the `ZafApplication` object is destroyed.

Members

Unless otherwise noted, member functions that set or get class attributes will return the final or current value of the attribute. Under normal circumstances, this will be the value passed into the `Set*()` function. However, if the `Set*()` function does not successfully change the state as requested, it will instead return the current state.

```
void Center(ZafWindowObject *object);
```

The `Center()` function centers the window passed into the *object* parameter on the screen. This function may be used before adding the window to the window manager, as well as after it has appeared on the screen. Consider the following example:

```
// Create a window centered on the screen.
ZafWindow *window = new ZafWindow(5, 5, 50, 10);
window->AddGenericObjects(new ZafStringData("Main Window"));
windowManager->Center(window);
// Add the window to the screen.
windowManager->Add(window);
```

```
ZafWindowObject *dragObject;
```

The `dragObject` member is used during drag and drop and points to the object being dragged. This advanced member is used internally by ZAF, and should not be modified by the programmer.

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This overloaded function handles all events sent to the `ZafWindowManager` object, whether by processing the events itself, or by passing them to a base class for processing. See `ZafWindowObject` for more information.

In addition to events handled by its base classes, `ZafWindowManager` handles the following events:

Event	Description
A_CHANGE_LANG_LOC	causes the application to change its language and locale bases specified in <code>event.text</code> (the object handling the event deletes <code>event.text</code>)
A_CLOSE_WINDOW	causes the application to close the window whose <code>stringID</code> matches <code>event.text</code> (if the window is <code>Destroyable()</code> it is deleted, and the object handling the event deletes <code>event.text</code>)
A_MINIMIZE_WINDOWS	causes the window manager to minimize all the windows on the display
A_OPEN_WINDOW	causes the application to open the persistent window whose storage pathname is in <code>event.text</code> (the object handling the event deletes <code>event.text</code>)
A_RESTORE_WINDOWS	causes the window manager to restore all the minimized windows on the display
S_CLOSE	causes all the temporary windows on the display and the top-most window to be closed (and deleted if it is <code>Destroyable()</code>)

Event	Description
S_CLOSE_TEMPORARY	causes all the temporary windows on the display to be closed
S_EXIT	causes the application to exit (if there is an ExitFunction(), it is called first)
S_NEXT_WINDOW	causes the next window to become the top-most

```
ZafExitFunction ExitFunction(void) const;  
ZafExitFunction SetExitFunction(ZafExitFunction  
    exitFunction);
```

If ExitFunction() is not null, it is called before the application may close down. The application may be requested to close down programmatically, by the end user, or by the native environment. Note that there are some cases that the native environment will close an application without giving it a chance to call its exit function.

Programmatically, an S_EXIT message may be posted on the event manager's queue. The end-user may request the application to close down by attempting to close the main window. (The main window is specified by setting the window manager's screenID to be the same as the window's screenID.) Either of these two methods will cause the ExitFunction() to be called.

If the ExitFunction() returns zero, the application is allowed to close down, otherwise the message is ignored. The default value of this attribute is null, but the user may call SetExitFunction() to change it. An example of specifying an exit function follows:

```
// Create the main window.  
ZafWindow *window = new ZafWindow(5, 5, 50, 10);  
window->AddGenericObjects(new ZafStringData("Main Window"));  
// Add the window to the screen. This gives it a screenID.  
windowManager->Add(window);  
// Specify the window as the main window.  
windowManager->screenID = window->screenID;  
// Set up the exit function.  
extern ZafExitFunction MyExitFunction;  
windowManager->SetExitFunction(MyExitFunction);
```

```
ZafWindowObject *helpObject;
```

The `helpObject` member is used by `ZafHelpTips` and points to the object used to display help messages. This advanced member is used internally by ZAF, and should not be modified by the programmer.

`ZafWindowObject *mouseObject;`

The `mouseObject` member is used by `ZafHelpTips` and points to the object under the mouse. This advanced member is used internally by ZAF, and should not be modified by the programmer.

ZafWindowObject

AcceptDrop	AutomaticUpdate	BackgroundColor
BeginDraw	Bordered	Changed
ClassID	ClassName	ConvertCoordinates
ConvertRegion	ConvertToDrawRegion	ConvertToOSPosition
ConvertToZafPosition	ConvertToOSRegion	ConvertToZafRegion
ConvertToZafEvent	CopyDraggable	DefaultUserFunction
Disabled	display	DragDropEvent
Draggable	Draw	DrawBackground
DrawBorder	DrawFocus	DrawShadow
Duplicate	EndDraw	Error
Event	eventManager	Focus
FocusObject	Font	GetObject
HelpContext	HelpObjectTip	InitialDelay
IsA	LinkDraggable	LogicalEvent
memberUserFunction	MemberUserFunction	MoveDraggable
MoveEvent	Next	NonCurrent
NotifyFocus	NotifySelection	OSDraw
OSScreenID	PaletteState	Parent
ParentDrawBorder	ParentDrawFocus	ParentPalette
Previous	QuickTip	Read
Redisplay	Region	RegionType
RepeatDelay	RootObject	ScreenID
ScrollEvent	Selected	ToggleSelected
Text	TextColor	userFunction
UserFunction	userObject	userPaletteData
UserPaletteData	Visible	windowManager
Write	zafRegion	

ZafWindowObject defines the basic functionality necessary to display objects on the screen. All ZAF displayable objects including windows, strings, buttons, prompts, borders, etc., derive from this class.

ZAF displayable classes share certain characteristics. For example, all displayable objects occupy a certain region on the screen. Many objects display text in a variety of font types and sizes. Some objects display borders and other visual characteristic in a multitude of foreground and background colors. Frequently, displayable objects handle events such as gaining or losing focus. Finally, displayable objects sometimes allow drag and drop functionality to other displayable objects.

ZafWindowObject is important as a base class because it defines the basic functionality that provides the underlying definition of a graphical user interface system. Although this class cannot be instantiated directly, other classes can be derived from ZafWindowObject.

Since this is the core class for all user interface objects, the descriptions in this chapter are more comprehensive and descriptive. In each case, examples apply to ZafWindowObject, but also may apply to the creation and use of all user interface objects. Refer to this and the ZafWindow section of this manual when questions arise regarding the general operation of user interface objects.

Declaration `#include <z_win.hpp>`

Inheritance `ZafWindowObject : ZafElement`

Constructors All ZafWindowObject constructors initialize the member variables associated with an instantiated ZafWindowObject object. Default values set by the ZafWindowObject and its base class constructors are listed below.

Member Initializations

ZafWindowObject

<code>AcceptDrop()</code>	<code>false</code>
<code>AutomaticUpdate()</code>	<code>true</code>
<code>Bordered()</code>	<code>false</code>
<code>Changed</code>	<code>false</code>
<code>CoordinateType()</code>	<code>ZAF_CELL</code>
<code>CopyDraggable</code>	<code>false</code>
<code>Disabled()</code>	<code>false</code>
<code>EditMode()</code>	<code>false</code>
<code>Error()</code>	<code>ZAF_ERROR_NONE</code>
<code>Focus()</code>	<code>false</code>
<code>HelpContext()</code>	<code>NO_HELP_CONTEXT</code>
<code>HelpObjectTip()</code>	<code>null</code>
<code>LinkDraggable()</code>	<code>false</code>
<code>MoveDraggable()</code>	<code>false</code>
<code>memberUserFunction</code>	<code>ZafWindowObject::DefaultUserFunction</code>
<code>Noncurrent()</code>	<code>false</code>
<code>OSDraw()</code>	<code>true</code>
<code>parent</code>	<code>null</code>
<code>ParentDrawBorder()</code>	<code>false</code>
<code>ParentDrawFocus</code>	<code>false</code>
<code>ParentPalette()</code>	<code>false</code>
<code>QuickTip()</code>	<code>null</code>
<code>Region()</code>	<code>ZAF_CELL, left, top, left + width - 1, top + height - 1</code>
<code>RegionType()</code>	<code>ZAF_INSIDE_REGION</code>
<code>screenID()</code>	<code>null</code>

Member Initializations

Selected()	false
SupportObject()	false
SystemObject()	true
userObject	null
userFlags	0
userStatus	0
UserFunction()	null
UserPaletteData()	null
Visible()	true

ZafElement

ClassID()	ID_ZAF_WINDOW_OBJECT
ClassName()	"ZafWindowObject"

Since the constructors for `ZafWindowObject` are protected, you cannot instantiate a `ZafWindowObject` directly—it must be initialized through a derived class such as `ZafButton`, `ZafWindow`, or `ZafText`. The arguments and types of constructors for this class represent the basic construction techniques available to all derived window objects.

```
ZafWindowObject(int left, int top, int width, int height);
```

The first constructor is useful in straight-code situations. *left*, *top*, *width*, and *height* define the absolute size and position of the window object (relative to the top-left corner of its parent's "client" region). By default, these values are given in cell coordinates, but minicell, pixel, point, or twip coordinates can be used if you set the `CoordinateType()` attribute immediately after the constructor as follows:

```
ZafWindowObject *object = new ZafWindow(50, 30, 200, 95);  
object->SetCoordinateType(ZAF_PIXEL);
```

The following examples demonstrate how derived objects call the base `ZafWindowObject` constructor:

```
ZafButton::ZafButton(int left, int top, int width, int height,  
    ZafBitmapData *zBitmapData, const ZafIChar *text,  
    ZafButtonType tButtonType) : ZafWindowObject(left, top,  
    width, height)
```

```

...

ZafWindow::ZafWindow(int left, int top, int width, int height) :
    ZafWindowObject(left, top, width, height)
...

ZafString::ZafString(int left, int top, int width, const
    ZafIChar *text,
    int maxLength) :
    ZafWindowObject(left, top, width, 1)
...

// Show actual object creation using the constructors shown
// above.
ZafButton *button = new ZafButton(0, 0, 10, ZAF_NULLP(ZafIChar),
    "button");
ZafWindow *window = new ZafWindow(0, 0, 50, 10);
ZafString *string = new ZafString(2, 2, 20, "message1", 100);

```

ZafWindowObject(const ZafWindowObject ©);

The copy constructor calls the overload Duplicate() to create a new ZafWindowObject and initialize its data from *copy*. Note that the copy constructor “copies” all field information including attributes, sizes, and text information, so variables such as QuickHelpText(), UserPaletteData() and derived object’s data components that do not have the “StaticData() == true,” will depth copy their information, not just repoint the internal variables to the original object’s values.

The following examples show how derived objects use the base ZafWindowObject copy constructor to duplicate base class information. Also included is a code sample that shows how to duplicate another window object.

```

ZafButton::ZafButton(const ZafButton &copy) :
    ZafWindowObject(copy)
...

ZafWindow::ZafWindow(const ZafWindow &copy) :
    ZafWindowObject(copy)
...

ZafString::ZafString(const ZafString &copy) :
    ZafWindowObject(copy)
...

// Example 1: Create, then copy a string object.
ZafString *string1 = new ZafString(2, 2, 20, "message1", 100);

```

```
string1->SetBordered(false);
string1->SetTextColor(ZAF_CLR_BLUE);
string1->SetAutoClear(true);

ZafString *string2 = new ZafString(*string1); // A lot easier
        than the code above.
string2->SetText("message2");
```

```
ZafWindowObject(const ZafIChar *name,
        ZafObjectPersistence &persist);
```

The final constructor is used for persistence. *name* specifies the name of the window or window object to be read from a persistent file. *persist* contains persistent information such as a pointer to the file-system and object constructors, both necessary for object creation.

Below are several code snippets that show how derived objects use the base `ZafWindowObject` persistent constructor to read base class information. Also included is a code sample that shows how to use persistence in general window creation.

```
ZafButton::ZafButton(const ZafIChar *name, ZafObjectPersistence
        &persist) :
        ZafWindowObject(name, persist.PushLevel(className, classID,
        ZAF_PERSIST_DIRECTORY))
...

```

```
ZafWindow::ZafWindow(const ZafIChar *name, ZafObjectPersistence
        &persist) :
        ZafWindowObject(name, persist.PushLevel(className, classID,
        ZAF_PERSIST_ROOT_DIRECTORY)),
...

```

```
ZafString::ZafString(const ZafIChar *name, ZafObjectPersistence
        &persist) :
        ZafWindowObject(name, persist.PushLevel(className, classID,
        ZAF_PERSIST_DIRECTORY)),
...

```

```
// Load a persistent window.
ZafStorage *storage = new ZafStorage("myfile.dat");
ZafObjectPersistence persist(storage);
windowManager->Add(new ZafWindow("MyWindow", persist));
```

Destructor

```
virtual ~ZafWindowObject(void);
```

This destructor is used to free the memory associated with a ZafWindowObject object. The ZafWindowObject portion of the destructor deletes the object's QuickTip(), HelpObjectTip(), and UserPalette() members if memory has been allocated. It then chains to the ZafElement destructor.

Generally, the programmer will not directly destroy a ZafWindowObject object since it is automatically destroyed when its parent window is destroyed. For more information on child object deletion see ZafWindow::~ZafWindow().

Below is an example of object deletion that chains the destruction sequence to the ZafWindowObject class.

```
// Create a simple window.
ZafWindow *window = new ZafWindow(0, 0, 40, 10);
window->Add(new ZafBorder);
window->Add(new ZafMaximizeButton);
window->Add(new ZafMinimizeButton);
window->Add(new ZafSystemButton(ZAF_NATIVE_SYSTEM_BUTTON));
window->Add(new ZafTitle(new ZafStringData("Text Window")));
...
ZafWindowObject *object = new ZafText(2, 1, 35, 6, "text",
    1000);
window->Add(object);
...
// This call selectively deletes a child object.
window->Subtract(object);
delete object;
...
// This call deletes the whole window with all its children.
delete window;
```

Members

```
bool AcceptDrop(void) const;
bool SetAcceptDrop(bool acceptDrop);
```

These functions are part of a suite of drag/drop functions and attributes that allow the user to move, copy, or link application specific data from one object to another. Although different in actual presentation, all environments provide visual queues to the user when they permit drag and drop operations from one object to another. For example, the Windows environment shows a copy or move folder when a draggable object is picked-up in a dragging operation, then interactively shows either the continued copy/move image when positioned over a dropable object, or a cancel image if the mouse is positioned over an object that does not allow drop operations.

Calling `SetAcceptDrop(true)` tells ZAF that in your application the specified object allows dropping operations. When this is done, ZAF first communicates with the native environment, then with your application, by sending drop messages to your object's virtual `Event()` function.

`ZafWindowObject` provides simple handling of the copy and move drag/drop operations, but does not have any special handling for link drag/drop. Simple copying or moving of the object's `Text()` information for objects derived from `ZafWindowObject` is accomplished as follows:

```
// Perform the specified drop operation.
object = windowManager->dragObject;
const ZafIChar *text = object->Text();
if (ccode == S_DROP_COPY)
    SetText(text);
else if (ccode == S_DROP_MOVE)
{
    SetText(text);
    object->SetText(ZafLanguageData::blankString);
}
```

Most ZAF objects add additional native drag/drop capabilities into their run-time operation. Below is a partial list of drag/drop operations that are automatically handled when the `AcceptDrop()` is set to “true.”

Operations	Description
<code>ZafString</code>	Accepts the <code>Text()</code> portion of the drag object.
<code>ZafDate</code>	Converts the <code>Text()</code> portion of the drag object to a date value if the value is in a format recognized by the <code>ZafDate</code> object.
<code>ZafVtList</code>	Duplicates the drag object by calling the virtual <code>Duplicate()</code> function when the <code>S_DROP_COPY</code> is sent, or moves the actual object (by sending the message <code>S_SUBTRACT_OBJECT</code> to the drag object's parent and then <code>S_ADD_OBJECT</code> to itself) when the <code>S_DROP_MOVE</code> message is sent.

Normally, programmers will not intercept drop messages since most Zinc objects have drag and drop capabilities built into their native operation. If you wish to intercept drop messages, however, these are the messages to look for:

Message	Description
S_DROP_MOVE	<p>Sent to the destination object's Event() function when the user initiates a drag-move operation from one object to another. If this message is received the programmer should perform a move operation from the windowManager->dragObject to the destination. Thus, when the drag contents are moved, they should be removed from the drag object. The following picture illustrates the results of a move operation from one string to another.</p> <p>(picture here)</p>
S_DROP_COPY	<p>Sent to the destination object's Event() function when the user initiates a drag-copy operation from one object to another. If this message is received the programmer should perform a copy operation from the windowManager->dragObject to the destination. Thus, when the drag contents are copied, they should not be removed from the drag object. The following picture illustrates the results of a copy operation from one string to another.</p> <p>(picture here)</p>
S_DROP_LINK	<p>Sent to the destination object's Event() function when the user initiates a drag-link operation from one object to another. This is the least common mode of dragging and dropping, but is useful when you want to access data from a particular drag source, but do not want to copy the drag object's data. If this message is received the programmer should not copy or replace the data contents of windowManager->dragObject. The following picture illustrates the results of a link operation from one view object to an associated database record.</p> <p>(picture here)</p>

Message	Description
S_DROP_DEFAULT	<p>This message should only be intercepted when needing to change the default sequence of drag/drop messages. ZafWindowObject provides an algorithm for S_DROP_DEFAULT that sends messages in the following order:</p> <ul style="list-style-type: none">• S_DROP_COPY if the CopyDraggable() attribute is true• S_DROP_MOVE if S_DROP_COPY fails (returns an S_ERROR) or if the CopyDraggable() attribute is false and if the MoveDraggable() attribute is true• S_DROP_LINK if conditions in steps (1) or (2) are not met and if the LinkDraggable() attribute is true. <p>The order of drop messages can be modified by overriding the S_DROP_DEFAULT functionality in a derived object's Event() or DropEvent() function.</p>

The original drag object pointer is contained in windowManager->dragObject. The following code demonstrates how to copy the text associated with a dragged object when a custom target object has been marked with the accept drop capability.

```
void MyFunction(void)
{
    MyDropObject *target = new MyDropObject;
    target->SetAcceptDrop(true);
}

ZafEventStruct MyDropObject::Event(const ZafEventStruct &event)
{
    switch (event.type)
    {
        case S_DROP_DEFAULT:
        case S_DROP_COPY_OBJECT:
            // Copy the source text information.
            SetText(windowManager->dragObject->Text());
            break;

        case S_DROP_MOVE_OBJECT:
        case S_DROP_LINK_OBJECT:
            // Don't handle these cases.
            return (S_ERROR);
    }
}
```

The return value for `AcceptDrop()` and `SetAcceptDrop()` is the final, or current drop attribute associated with the object (true or false). Under normal circumstances, this value will be passed into the `SetAcceptDrop()` function. The following example demonstrates proper use of these functions and argument:

```
// Get the attribute.
if (object->AcceptDrop())
    break;

// Set the object as "drop capable."
ZafString *string = new ZafString(0, 0, 10, "string", 100);
string->SetAcceptDrop(true);
```

```
bool AutomaticUpdate(void) const;
virtual bool SetAutomaticUpdate(bool automaticUpdate);
```

`SetAutomaticUpdate()` turns on or off an object's ability to immediately draw itself on the screen. The following function calls are affected by the state of `AutomaticUpdate():Add()` - for `ZafWindow` objects

- `Add()` - for `ZafWindow` objects
- `Subtract()` - for `ZafWindow` objects
- `SetBackgroundColor()`
- `SetFont()`
- `SetTextColor()`
- `SetUserPaletteData()`

`SetAutomaticUpdate()` should be set to false to change several color or font attributes at once without causing screen flashes to the object with each attribute change. The following example shows how to efficiently change these attributes on a window object.

```
// Change an object's text and background color.
object->SetAutomaticUpdate(false);
object->SetBackgroundColor(ZAF_CLR_RED);
object->SetTextColor(ZAF_CLR_WHITE);
object->SetAutomaticUpdate(true);
```

`AutomaticUpdate()` is a temporary setting. When an object's automatic update is set to "false," immediately make necessary color and font changes, then set the value back to "true." Delayed restoration of this attribute may cause unwanted screen updates, if other attributes, not defined above, are changed while the object's automatic update status is "false."

Setting this attribute back to true does three things:

- Matches the object’s internal attributes with the particular environment values.
- Causes the displayed object to show any changes made while the attribute was set to false (by automatically redrawing the object where necessary).
- Ensures any future application activity which affects the appearance of the object will occur immediately and automatically.

The derived ZafWindow class provides additional Add()/Subtract() optimization to this function. For more information on these changes see the ZafWindow section of this manual.

The return value for AutomaticUpdate() and SetAutomaticUpdate() is the final or current update status associated with the object (true or false). Under normal circumstances this will be the update value passed into the SetAutomaticUpdate() function but may be different if the object does not allow changes to automatic updating (e.g. ZafMessageWindow always sets the value to true).

```
ZafLogicalColor BackgroundColor(ZafLogicalColor *color =  
    ZAF_NULLP(ZafLogicalColor),ZafLogicalColor *mono =  
    ZAF_NULLP(ZafLogicalColor));  
ZafLogicalColor SetBackgroundColor(ZafLogicalColor  
    color, ZafLogicalColor mono = CLR_DEFAULT);
```

Background color is the visual presentation area that encompasses the region behind an object’s text or image information, excluding the object’s border and shadow. The following picture identifies the BackgroundColor() area of three window objects.

(picture here)

SetBackgroundColor() changes the background color associated with the normal presentation of an instantiated object. Two types of colors can be passed to SetBackgroundColor(): a color value and a monochrome value. The first parameter specifies the color for normal operation, the second specifies the black/white value for monochrome or black/white modes of operation. Here is a list of predefined ZAF color values:

Color Values	Monochrome Values
ZAF_CLR_PARENT	ZAF_MONO_PARENT
ZAF_CLR_DEFAULT	ZAF_MONO_DEFAULT
ZAF_CLR_NULL	ZAF_MONO_NULL
ZAF_CLR_BACKGROUND	ZAF_MONO_BACKGROUND
ZAF_CLR_BLACK	ZAF_MONO_BLACK

Color Values	Monochrome Values
ZAF_CLR_BLUE	ZAF_MONO_DIM
ZAF_CLR_GREEN	ZAF_MONO_NORMAL
ZAF_CLR_CYAN	ZAF_MONO_WHITE
ZAF_CLR_RED	ZAF_MONO_HIGH
ZAF_CLR_MAGENTA	
ZAF_CLR_BROWN	
ZAF_CLR_LIGHTGRAY	
ZAF_CLR_DARKGRAY	
ZAF_CLR_LIGHTBLUE	
ZAF_CLR_LIGHTGREEN	
ZAF_CLR_LIGHTCYAN	
ZAF_CLR_LIGHTRED	
ZAF_CLR_LIGHTMAGENTA	
ZAF_CLR_YELLOW	
ZAF_CLR_WHITE	

In addition to the pre-defined colors described above, users can define and use their own logical colors. For more information on these color specifications, and for more details on derived color entries, see the `ZafPaletteStruct` and `ZafDisplay` sections of this manual or `ZafWindowObject::UserPaletteData()`.

This example demonstrates the correct use of `SetBackgroundColor()`.

```
// Change the background color of the object.
object->SetBackgroundColor(ZAF_CLR_WHITE, ZAF_MONO_WHITE);

// Check the current color of an object, and change where
// necessary.
if (object->BackgroundColor() == ZAF_CLR_NULL)
    object->SetBackgroundColor(object->parent-
        >BackgroundColor());

// Change an object's text and background color.
object->SetAutomaticUpdate(false);
object->SetBackgroundColor(ZAF_CLR_RED);
object->SetTextColor(ZAF_CLR_WHITE);
object->SetAutomaticUpdate(true);
```

The return value for `BackgroundColor()` and `SetBackgroundColor()` is the final or current color value associated with the object. Under normal circumstances, this will be the color value passed into the `SetBackgroundColor()`, but may be

different if the object does not allow for a particular type of color specification or if the system is running in black/white mode.

```
ZafRegionStruct BeginDraw(void);  
void EndDraw(void);
```

ZAF enables you to customize the appearance of an object by deriving from ZafWindowObject or some other displayable ZAF class and overriding the Draw() function. Before a Draw() function calls any functions that affect the appearance of a GUI object, such as DrawBorder(), DrawFocus(), or DrawShadow(), it must first determine the available drawing region, and set up any special environment values. This is done with calls to the object's BeginDraw() function.

BeginDraw() returns a region structure that defines the available drawing region of the object. The actual values in the region structure's top, left, bottom, and right fields will differ depending on the particular platform you are running under, so you should not make any assumptions about the nature of these values. Here is an example that shows the proper method for finding the horizontal center of a draw region:

```
ZafEventType MyObject::Draw(const ZafEventStruct &event,  
                             ZafEventType ccode)  
{  
    ZafRegionStruct region = BeginDraw();  
    int centerColumn = region.left + region.Width() / 2;  
    ...  
    EndDraw();  
    return (ccode);  
}
```

EndDraw() tells the system that you are finished with your drawing operation, and frees up any operating system specific memory used as part of the drawing operation.

```
bool Bordered(void) const;  
bool SetBordered(bool bordered);
```

SetBordered() is used to set the visual presentation of an instantiated object. Setting this attribute to true causes a platform-specific border to be drawn around the object. Typically, this is a 3-dimensional shadowed border that is prevalent on newer versions of GUI environments (e.g. Windows95 and Motif), but may also be a one or two pixel black band that surrounds the object, seen on other GUI systems. If not set, the object will not have any border that

frames the data portion of an object. Here is a picture of two ZafString objects, the left calling SetBordered(true), the right SetBordered(false).

(pict)

The following code shows the proper use of these functions:

```
// Check the attribute for drawing.
if (object->Bordered())
    object->DrawShadow(drawRegion, ZAF_BORDER_WIDTH, ccode);

// Turn the border off.
ZafString *string = new ZafString(0, 0, 10, "string", 100);
string->SetBordered(false);
```

The return value for Bordered() and SetBordered() is the final, or current border attribute associated with the object (true or false). Under normal circumstances, this will be the value passed to the SetBordered() function, but may be different if the object does not allow changes to the Bordered() attribute, or if the SetBordered() function is called after the object is visible on the screen.

```
bool Changed(void) const;
bool SetChanged(bool changed);
```

These functions are used to set or evaluate the changed status of an object. Many objects derived from ZafWindowObject set this attribute to true each time a user modifies the object's data. For example, ZafString sets this attribute each time a user types a character into the string buffer. ZafVtList sets the attribute each time a user adds, deletes, or selects an item from the list.

You can determine whether an end-user has changed the contents of an object by calling the Changed() member. In addition, you can clear the changed status by calling SetChanged(false). Note, ZAF only sets the changed attribute with SetChanged(true) calls, it never clears the attribute with SetChanged(false) calls. This operation is left for the programmer's discretion. The following code shows the proper use of these functions:

```
// Check the changed attribute.
if (myString->Changed())
{
    // Save the data.
    SaveMyString(myString->Text());
    myString->SetChanged(false);
}
```


The return value for `Changed()` and `SetChanged()` is the final, or current changed attribute associated with the object (true or false). Under normal circumstances, this will be the value passed into the `SetChanged()` function.

```
static ZafClassID classID;  
static ZafClassNameChar ZAF_FARDATA className[];  
ZafClassID ClassID(void) const;  
ZafClassName ClassName(void) const;
```

These two member functions overload the `ZafElement::ClassID()` and `ZafElement::ClassName()` functions, by returning the added class identification `ID_ZAF_WINDOW_OBJECT` and string “ZafWindowObject.”

The static members `classID` and `className` are used as internal place-holders for the class’s name and identification. They should not be used or set by the programmer.

Convert Functions

This group of advanced functions is used to convert ZAF information to OS information, or vice-versa. Typically, you will not need to use these functions. They are provided as convenience functions for derived window objects that create and manipulate OS information directly. Nevertheless, their functionality is described briefly here, to help you understand the relationship of events, coordinates and regions from ZAF to environment specific GUIs.

ZAF provides a mid-layered product that insulates you from the specific nuances of each operating environment. These differences are particularly acute in the area of coordinates and keyboard translation. The convert routines described in this section give Zinc the ability to “hide” most of the OS specific mechanics to provide a consistent cross-platform application environment. With this introduction, here are specific descriptions of the various convert functions:

```
virtual void ConvertCoordinates(ZafCoordinateType  
    coordinateType);
```

ConvertCoordinates This function converts `zafRegion` using `ConvertRegion()` with the specified coordinate type.

```
virtual void ConvertRegion(ZafRegionStruct &region,  
    ZafCoordinateType newType);
```

ConvertRegion This function encapsulates the conversion of all regions relevant to the window object. This is not a ZAF to OS conversion, rather a current type (cell, mini-cell, pixel, point, twips) to OS type (generally pixel based). This function is generally used to encapsulate a `zafRegion.ConvertCoordinates()` call, but for advanced or composite classes, such as scrolled window, it not only encapsulates the `zafRegion` conversion, but also provides a mechanism to convert private variables, such as `scrollRegion`.

```
virtual ZafRegionStruct ConvertToDrawRegion(const
    ZafWindowObject *object, const ZafRegionStruct
    *zafRegion = ZAF_NULLP(ZafRegionStruct)) const;
```

ConvertToDrawRegion This function converts the object's `zafRegion` to a region compatible with the environments display. This function is used in conjunction with `BeginDraw()` to determine the actual coordinates you need to use when calling display functions, such as: `Text()`, `Rectangle()`, `Line()`, etc. This return region can also be used directly with the environment specific display APIs. For example, the Windows API to draw a rectangle is `Rectangle()` or `FillRect()` depending on the fill specification. The region passed back from `ConvertToDrawRegion()` can be used directly with these native API calls, or with `ZafDisplay` calls.

```
virtual ZafPositionStruct ConvertToOSPosition(const
    ZafWindowObject *object, const ZafPositionStruct
    *zafPosition = ZAF_NULLP(ZafPositionStruct)) const;
virtual ZafPositionStruct ConvertToZafPosition(const
    ZafWindowObject *object, const ZafPositionStruct
    *osPosition = ZAF_NULLP(ZafPositionStruct)) const;
```

ConvertToOSPosition/ConvertToZafPosition These functions convert a ZAF position to an OS position, and vice-versa. Other sections of this manual describe the ZAF coordinate system. It is a 0,0 left-top based system that places objects either in a “client” region (if it is a normal window object), or in a “frame” region (if it is a support object).

(picture here)

Sometimes these ZAF positions match the OS, other times, they do not. These function allow Zinc to readjust the position based on potential coordinate shifting needs, including: shadowing, bordering, native vs. non-native OS objects, etc.

```
virtual ZafRegionStruct ConvertToOSRegion(const  
    ZafWindowObject *object, const ZafRegionStruct  
    *zafRegion = ZAF_NULLP(ZafRegionStruct)) const;  
virtual ZafRegionStruct ConvertToZafRegion(const  
    ZafWindowObject *object, const ZafRegionStruct  
    *osRegion = ZAF_NULLP(ZafRegionStruct)) const;
```

ConvertToOSRegion/ConvertToZafRegion These functions are very similar to the position functions, except that they convert full regions, not just a single point. Normally, this function simply makes two calls to **ConvertToOSPosition()**, to compute the top-left and bottom-right positions of the region, but occasionally, the region needs additional computations. For instance, Motif automatically sizes the region of toolbars to account for a small border area around its children. This is done by computing the actual OS region, based on **zafRegion**, then by adding a 2-pixel boundary to the region. The overload of **ConvertToOSRegion()** allows for this modification.

```
virtual bool ConvertToZafEvent(ZafEventStruct &event);
```

ConvertToZafEvent This is the most comprehensive of the convert functions. The main aspects of this function are to convert OS specific keyboard and mouse event data (such as **KeyPress**, **WM_KEYPRESS**, **keyDown**) to ZAF portable key and position interpretations. The two parts of the event structure that may be converted are **event.key** and **event.position**.

If the passed event is an OS specific keyboard event, ZAF determines the modifiers (**S_SHIFT**, **S_CTRL**, **S_ALT**) and key values. For instance, on Motif, a keyboard's shift state information is passed by **xEvent.xkey.state** and the pressed key is represented by **xEvent.xkey**. To get ZAF equivalents, we must look at each **xkey.state** and map to the ZAF equivalent, then call **XLookupString()** to get the ascii value of the key.

If the event is an OS specific mouse event, ZAF must not only determine the keyboard's shift state, but also convert the native OS position to a coordinate understood and relevant to ZAF. This is done by looking at OS position, looking at the intended target, and then converting the position based on our understanding of the environment specific coordinates, versus the coordinate required by portable ZAF. In Windows, this means calling the **ClientToScreen()** API, then adjusting the coordinate to not be Windows "client" based.

Most of these functions take **object** as a parameter. This can be a little confusing since the functions are, themselves, member functions that have a this pointer. The reason for this parameter is to give context to the requesting object. You should always make a position or region request to your parent, or to the window manager if there is no parent.

Here are some sample code snippets from ZAF's library that show the correct use of these convert functions.

```
void ZafWindowObject::OSSize(void)
{
    // Convert to the os region.
    ZafRegionStruct newRegion;
    if (parent)
        newRegion = parent->ConvertToOSRegion(this, &zafRegion);
    else
        newRegion = windowManager->ConvertToOSRegion(this,
            &zafRegion);
    ...
}

void ZafScrolledWindow::ConvertCoordinates(ZafCoordinateType
    newType)
{
    ConvertRegion(zafRegion, newType);
    ConvertRegion(scrollRegion, newType);
}

// Check for a converted event.
if (event.converted != this)
    ConvertToZafEvent(event);

CoordinateType
ZafCoordinateType CoordinateType(void) const;
virtual ZafCoordinateType SetCoordinateType(ZafCoordinateType
    coordinateType);
```

ZAF allows you to specify an object’s initial coordinates in terms of cells, minicells, pixels, points, or twips. Here is a description of what each enumerated type means:

Type	Description
ZAF_CELL	<p>A value based on the size of ZAF_DIALOG_FONT. This value calculates the average height and width of a dialog font, and adds information such as pre- and post-spacing, margin widths, border widths, etc. to determine the optimal size of a cell on the particular environment. Here is a picture that roughly shows how a cell’s height and width are determined.</p> <p>(picture here)</p> <p>The pixel conversion values for ZAF_CELL are contained in two ZafCoordinateStruct members, cellHeight and cellWidth which are determined at run-time by the ZafDisplay constructor.</p> <pre>int ZafCoordinateStruct::cellHeight; int ZafCoordinateStruct::cellWidth;</pre>
ZAF_MINICELL	<p>A fractional value of a cell. ZAF defines mini-numerator and denominator values in ZafCoordinateStruct.</p> <p>(picture here)</p> <p>These ZafCoordinateStruct values are pre-defined to be the following values:</p> <pre>long ZafCoordinateStruct::miniNumeratorX = 1; long ZafCoordinateStruct::miniDenominatorX = 10; long ZafCoordinateStruct::miniNumeratorY = 1; long ZafCoordinateStruct::miniDenominatorY = 10;</pre>
ZAF_PIXEL	<p>A pixel based value. Most systems run in “pixel-based” coordinates which, if specified, means your object’s region will not need conversion by the operating environment. This allows the exact placement of objects in a window, sometimes essential when presenting graphical information to the display. The use of this type is discouraged on text information objects, because the size of font is environment dependent, thus causing size problems when the exact font size for a particular environment is unknown.</p>

Type	Description
ZAF_POINTS	<p>A value based on the size of the default system font. The relationship of this value to pixels is determined at run-time, but is based on a general formula that 72 points equal a theoretical inch.</p> <p>(picture here)</p> <p>The pixel conversion values for ZAF_POINTS are contained in two ZafCoordinateStruct members, pixelsPerInchX and pixelsPerInchY and are determined at run-time by the ZafDisplay constructor.</p> <pre>long ZafCoordinateStruct::pixelsPerInchX; long ZafCoordinateStruct::pixelsPerInchY;</pre>
ZAF_TWIPS	<p>Defined to be 1/20 of a point. This value is computed at run-time, based on the value of the ZafCoordinateStruct pixelsPerInchX and pixelsPerInchY members.</p>

By default, all coordinates passed to an object's constructor are considered to be ZAF_CELL based. Thus, specifying a different coordinate type requires you to pass initial coordinate values to the constructor and then to specify the changed coordinate type through the SetCoordinateType() function. Here is some sample code that show how this is accomplished.

```
// Set the window to pixel based.
ZafWindow *window = new ZafWindow(0, 0, 500, 100);
window->SetCoordinateType(ZAF_PIXEL);

// Set all children to be twip based coordinates.
for (ZafWindowObject *object = First(); object; object = object->Next())
    object->SetCoordinateType(ZAF_TWIPS);
```

Resetting the coordinate type is valid up to the point when the object is presented to the screen. Actual coordinate conversion takes place when the S_INITIALIZE message is sent to the object (this message is sent when the object is added to the screen), and thus changes after this conversion are meaningless. Therefore, once the object's coordinates have been converted to native system values, calls to SetCoordinateType() are rejected.

Note, the original CoordinateType() value is preserved with the object, even though the actual coordinates are changed at run-time. Thus, even though an object may have converted its position and size to be pixel-based, the return

value of `CoordinateType()` will still be `ZAF_CELL`. To view the current run-time coordinate of an object, you should evaluate the `zafRegion.CoordinateType()` variable. Here is some code that shows these two usages.

```
// Create a point-based object.
ZafString *string = new ZafString(20, 20, 100, "string", 100);
string->SetCoordinateType(ZAF_POINTS);
...

// Evaluate the coordinates.
if (string->CoordinateType() != string-
    >zafRegion.CoordinateType())
    ...
    // string->CoordinateType() is ZAF_POINTS, but
    // string->zafRegion.CoordinateType() will vary depending
    // on the current state of the string object.
```

The return value for `CoordinateType()` and `SetCoordinateType()` is the initial, or enumerated value associated with the object (cell, mini-cell, pixel, point, or twips). Under normal circumstances, this will be the value passed into the `SetCoordinate()` function, but may be a different value if one of the conditions described above is met.

```
bool CopyDraggable(void) const;
bool SetCopyDraggable(bool copyDraggable);
```

See `ZafWindowObject::Draggable()`.

```
ZafEventType DefaultUserFunction(const ZafEventStruct
    &event, ZafEventType ccode);
```

`DefaultUserFunction` provides a simple notification mechanism that chains to a programmer specified user-function when the following events occur:

Event	Description
When <code>Focus()</code> changes from “true” to “false.”	When this event occurs, the user has either tabbed onto another field, or has moved the focus to another object using the mouse. If you have specified a user-function, it will be called with an <code>N_NON_CURRENT</code> message.

Event	Description
Focus() changes from “false” to “true.”	When this event occurs, the user has selected the receiving object, moving the focus to the object. If you have specified a user-function, it will be called with an N_CURRENT message.
User presses a selection key (usually the space bar, or <enter> key).	When this occurs and if you specify a user-function, the function is called with an L_SELECT message, but the contents of the event argument will be event.InputType() == E_KEY, signifying that a key event caused the selection.
User clicks the mouse button while being positioned over the object.	When this occurs and if you specify a user-function, the function is called with an L_SELECT message, but the contents of the event argument will contain an event.InputType() of E_MOUSE, signifying that a mouse event caused the user-function to be called.
User double-clicks on the object.	When this occurs and if you specify a user-function, the user-function is called with an L_DOUBLE_CLICK message. The contents of the event structure will be of type E_MOUSE, signifying that a mouse event caused the user-function to be called.

If you have not supplied a user-function with the object, this function has no effect in the application. It simply provides a “hooking” mechanism for user callbacks, rather than a more advanced concept known as derived pointers to member functions. These advanced concepts are discussed more fully in ZafWindowObject::memberUserFunction.

Here are some sample code snippets that show how the DefaultUserFunction() is set, and used in an application.

```
// Reset an object's member function.
object->memberUserFunction = DefaultUserFunction;

// Advanced ZafButton code that resets notification.
bool ZafButton::SetSendMessageWhenSelected(bool
    setSendMessageWhenSelected)
{
    // Make sure the attribute has changed.
    if (sendMessageWhenSelected != setSendMessageWhenSelected &&
        !screenID)
    {
        sendMessageWhenSelected = setSendMessageWhenSelected;
    }
}
```



```
        if (sendMessageWhenSelected)
            memberUserFunction =
                (MemberUserFunction)&ZafButton::SendMessage;
        else
            memberUserFunction = ZafWindowObject::DefaultUserFunction;
    }

    // Return the current attribute.
    return (sendMessageWhenSelected);
}
```

```
bool Disabled(void) const;
bool SetDisabled(bool disabled);
```

When an object is “disabled” it is typically presented as a grayed out object and does not allow any user interaction. It therefore cannot receive the input focus, cannot be tabbed on, and cannot receive keyboard or mouse events. The exact visual presentation depends on the operating environment, but is typically shown as a “dithered” text with shaded, rather than full-dark borders. This immediately tells the user, the object is not available for user selection or input.

Here is some sample code that shows the correct use of `SetDisabled()`.

```
// Place two objects in the window setting one disabled.
ZafButton *button1 = new ZafButton(2, 2, 20,
    ZAF_NULLP(ZafBitmapData), "Save");
window->Add(button1);
ZafButton *button2 = new ZafButton(22, 2, 20,
    ZAF_NULLP(ZafBitmapData), "Delete");
button2->SetDisabled(true);
window->Add(button2);

// Check the status of a object.
if (!object->Disabled())
    object->SetHelpContext("ObjectHelp");

// Disable a window, with all its children.
window->SetDisabled(true);
```

Note, setting the disabled attribute on a parent object causes all of the children to become disabled, even though their individual disabled states may not be “true.” `SetDisabled()` functionality is thus propagated to children, grandchildren, etc., but only through inheritance, not by value replacement (i.e. the child object’s disabled variable is not reset to be the same as the parent’s

value). Once the Disabled() state is set back to false, the visual and input settings of children are restored.

The return value for Disabled() and SetDisabled() is the final, or current disabled state of the object (true or false). Under normal circumstances, this will be the value passed into the SetDisabled() function.

```
static ZafDisplay *display;
```

This is a static pointer to the application's display. It is initialized when the ZafWindowManager constructor is called and should not be modified. All derived window objects use this member when drawing information to the screen. They do not use the global zafDisplay member.

Here is a code snippet that shows how the Motif ZafButton::Draw() function uses the display member variable.

```
ZafEventType ZafButton::Draw(const ZafEventStruct &,
    ZafEventType ccode)
{
    ZafRegionStruct drawRegion = BeginDraw();
    ...

    // Draw the border for flat buttons.
    if (Bordered() && buttonType != ZAF_3D_BUTTON)
    {
        display->SetPalette(LogicalPalette(ZAF_PM_OUTLINE, state));
        display->Rectangle(drawRegion, 1, false);
        drawRegion--;
    }
    ...
}
```

This member, as well as the static ZafWindowObject::eventManager and ZafWindowObject::windowManager members are duplicate copies of the global variables zafDisplay, zafEventManager, and zafWindowManager. They are defined in the base ZafWindowObject class to allow advanced ZAF programmers the opportunity of removing the static definition, thus allowing particular instance variables to be associated with each window object; a feature useful in some multi-display and embedded system applications.

```
virtual ZafEventType DragDropEvent(const ZafEventStruct
    &event);
```

This function is used as a dispatch function for the full group of system defined drag/drop messages. These messages include:

- S_DRAG_DEFAULT
- S_DRAG_MOVE
- S_DRAG_COPY
- S_DRAG_LINK
- S_DROP_DEFAULT
- S_DROP_COPY
- S_DROP_MOVE
- S_DROP_LINK
- S_BEGIN_DRAG
- S_END_DRAG

Whenever an object receives a drag/drop sequence of messages, the messages are first intercepted in the object's `Event()` function. To allow for more encapsulation and a more efficient handling of similar drag/drop functionality, ZAF defines a virtual `DragDropEvent()` which is called by the base `ZafWindowObject` class whenever one of the aforementioned messages is generated. This allows you to more efficiently handle drag/drop messages in a derived `DragDropEvent()` rather than the more generalized, and usually more cumbersome `Event()` function.

The following code shows how you might derive an object that overrides the `DragDropEvent()` function.

```
class MyButton : public ZafButton
...

ZafEventType MyButton::DragDropEvent(const ZafEventStruct
    &event)
{
    // Check for non-override messages.
    if (event.type != S_DROP_COPY)
        return (ZafButton::DragDropEvent(event));

    // Override the S_DROP_COPY functionality.
    ...
    return (event.type);
}
```

The return value for `DragDropEvent()` should be the passed `event.type` if processing is successful. Otherwise, the function should return the values

S_ERROR or S_UNKNOWN indicating the object either detected and error on the message, or that the function did not recognize the specified message.

```
bool Draggable(void) const;
bool CopyDraggable(void) const;
bool SetCopyDraggable(bool copyDraggable);
bool MoveDraggable(void) const;
bool SetMoveDraggable(bool moveDraggable);
bool LinkDraggable(void) const;
bool SetLinkDraggable(bool linkDraggable);
```

These functions are part of a suite of drag/drop functions and attributes that allow the user to move, copy, or link application specific data from one object to another. Although different in actual presentation, all environments provide visual queues to the user when they permit drag and drop operations from one object to another. For example, the Windows environment shows a copy or move folder when a draggable object is picked-up in a dragging operation, then interactively shows either the continued copy/move image when positioned over a dropable object, or a cancel image if the mouse is positioned over an object that does not allow drop operations.

Draggable() is a “read-only” function that indicates whether the MoveDraggable(), CopyDraggable() or LinkDraggable() attributes are set for the object. Generally, ZAF objects provide enough drag/drop functionality so that you will not need to evaluate the drag capability of an object. But in cases where you derive window objects, or where you are working on specific drag/drop capabilities, you will want to determine whether an object can be dragged, but will not be interested in exactly what type of drag operation can be performed. This function provides an efficient method for determining this drag state without calling each drag function separately. Its “read-only” status means you must use the SetMoveDraggable(), SetCopyDraggable() and SetLinkDraggable() functions to set the drag option on an object, there is no SetDraggable() function.

The following code shows how you might use the Draggable() function in your application.

```
// Example 1: Turn on the drag capability of run-time objects.
ZafEventType MyWindow::MemberCallback(ZafEventStruct &event,
    ZafEventType ccode)
{
    // Turn on the drag capabilities of all string objects.
    if (ccode == L_SELECT && allowDragButton->Selected())
```

```
        for (ZafWindowObject *object = First(); object; object =
            object->Next())
            if (object->IsA("ZafString") && !object->Draggable())
                object->SetCopyDraggable(true);
    return (0);
}

// Example 2: A derived window object.
ZafEventType MyObject::Event(const ZafEventStruct &event)
{
    switch (LogicalEvent(event))
    {
        case L_BEGIN_SELECT:
            // Determine the drag capability of the object.
            if ((event.rawCode & M_MIDDLE) && Draggable())
                ... // object can be dragged.
    }
}
```

The return value for each of these Draggable() functions is the final, or current drag state of the object (true or false). Under normal circumstances, this will be the value passed into the Set*Draggable() functions.

```
virtual ZafEventType Draw(const ZafEventStruct &event,
    ZafEventType ccode);
virtual ZafEventType DrawBackground(ZafRegionStruct
    &region, ZafEventType ccode);
```

ZAF enables you to customize the appearance of an object by deriving from ZafWindowObject or some other displayable ZAF class and overriding the Draw() function. Draw() is called whenever the system needs the object to update part, or all of its information to the screen. DrawBackground() is called whenever the system needs the clear the area behind an object. DrawBorder(), DrawFocus() and DrawShadow() are ZAF defined functions that perform common drawing operations. They should only be used within the context of a drawing operation according to the following specifications:

```
virtual ZafEventType DrawBorder(ZafRegionStruct &region,
    ZafEventType ccode);
```

DrawBorder() typically draws an OS specific border, generally a chiseled 3-d shape; but also detects for other GUI environments that draw a 1 pixel rectangle. Note, the region value passed to DrawBorder() is not a constant argument. Thus, the value associated with this region is changed according to the border width. For example, a 2-pixel border will adjust the specified region in, by 2

pixels. This allows you to automatically know what area of the object is still available for drawing operations.

```
virtual ZafEventType DrawFocus(ZafRegionStruct &region,
    ZafEventType ccode);
```

DrawFocus() draws an OS specific focus rectangle that surrounds the object. This is typically a one- or two-pixel black rectangle that encompasses the object, showing it as the “point-of-focus.” Note, as with **DrawBorder()**, the specified region is not a constant argument. On most environments, whether the focus rectangle is drawn, or not, the value of region is adjusted by the environment’s natural focus rectangle size.

```
virtual ZafEventType DrawShadow(ZafRegionStruct &region,
    int depth, ZafEventType ccode);
```

DrawShadow() differs from **DrawBorder()** because you specify the pixel depth size of the shadow to be drawn, and also because the function always draws a 3-dimensional shadow, not just the type of shadow specified by the native operating environment. The range of depth can be positive or negative. A negative value represents an indented shadow, generally associated with an object that has been “pushed-in” or that needs to be shown in a “depressed” state. As with **DrawBorder()** and **DrawFocus()**, the value of region is adjusted by the absolute depth of the shadow being drawn.

As you can see, most of the **Draw*()** functions described above, modify the specified region. Thus, calls to these functions should be done in a systemized manner, and should anticipate the adjustment of the region value. Here is some code that shows how one implementation of the **ZafButton::Draw()** function anticipates these region adjustments:

```
ZafEventType ZafButton::Draw(const ZafEventStruct &,
    ZafEventType ccode)
{
    // Compute the actual draw region.
    ZafRegionStruct drawRegion = BeginDraw();

    // Draw the focus.
    DrawFocus(drawRegion, Focus() ? S_CURRENT : S_NON_CURRENT);

    // Draw the shadow and fill the region.
    if (ButtonType() != ZAF_RADIO_BUTTON && ButtonType() !=
        ZAF_CHECK_BOX)
        DrawShadow(drawRegion, (Depressed() || Selected()) ? -depth
            : depth, ccode);
```

```
...
EndDraw();
return (ccode);
}
```

Here are a few more important notes associated with the Draw() and related functions:

- DrawBackground() should not be used within the Draw() function. All ZAF supported environments clear the background of an object with a method independent of the Draw() function. Thus, this function is generally called immediately before the Draw() function is called.
- The Draw() operation is called from the operating environment, and should not be initiated by the programmer. The correct way to cause an object's refresh is to call Redisplay() or RedisplayData(). Calling these functions causes Draw() to be called, but does it in coordination with the operating environment.
- Finally, the members BeginDraw() and EndDraw() must be called within your Draw() function to ensure the proper initialization of drawing arguments. For more information on these functions see ZafWindowObject::BeginDraw().

Here is some sample code that shows the correct use, and sequence of drawing operations within the Draw() function.

```
class TicTacToeCell : public ZafButton

ZafEventType TicTacToeCell::Draw(const ZafEventStruct &,
    ZafEventType ccode)
{
    // Compute the actual draw region.
    ZafRegionStruct drawRegion = BeginDraw();

    // Draw the focus.
    DrawFocus(drawRegion, Focus() ? S_CURRENT : S_NON_CURRENT);

    // Draw the shadow.
    DrawShadow(drawRegion, (Depressed() || Selected()) ? -depth :
        depth, ccode);

    // Draw the X or O.
    ZafPaletteState state = PaletteState();
    display->SetPalette(LogicalPalette(ZAF_PM_OUTLINE, state));
    if (MarkedWithAnX())
    {
        display->Line(drawRegion.left, drawRegion.top,
            drawRegion.right,
            drawRegion.bottom);
    }
}
```

```

        display->Line(drawRegion.left, drawRegion.bottom,
        drawRegion.right,
        drawRegion.top);
    }
    else if (MarkedWithAnO())
        display->Ellipse(drawRegion.left, drawRegion.top,
        drawRegion.right,
        drawRegion.bottom, 0, 360);

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

```

Note, the Draw() function will not be called for derived ZafWindowObjects on some environments unless “OSDraw() == false.” See ZafWindowObject::OSDraw() for more information on this function and its required use in derived class Draw() functions.

The return value for all Draw*() functions should be the passed ccode if processing is successful. Otherwise, the function should return the values S_ERROR or S_UNKNOWN indicating the object either detected an error on the message, or that the function did not recognize the specified message.

```
virtual ZafWindowObject *Duplicate(void);
```

This function creates a full-depth copy of “this” and returns a pointer to the copy. This function is declared virtual so derived objects can add their own copy constructors. The actual definition of this function by each ZAF object is a call to the object’s copy constructor:

```

virtual ZafWindowObject *Duplicate(void) { return (new
    ZafWindowObject(*this)); }
virtual ZafWindowObject *Duplicate(void) { return (new
    ZafWindow(*this)); }
virtual ZafWindowObject *Duplicate(void) { return (new
    ZafVtList(*this)); }

```

There is no particular advantage to using Duplicate() if you know the type of the object being copied. But if you are dealing with abstract lists of window objects or multiple-layered windows, calling duplicate is the only effective method of performing a depth copy on the object. Here is an example that shows this functionality.

```
// Example 1: Create, then copy a string object.
```



```
ZafString *string1 = new ZafString(2, 2, 20, "message1", 100);
string1->SetBordered(false);
string1->SetTextColor(ZAF_CLR_BLUE);
string1->SetAutoClear(true);

ZafString *string2 = new ZafString(*string1); // A lot easier
        than the code above.
string2->SetText("message2");

ZafString *string3 = string2->Duplicate(); // Almost like
        calling the copy constructor.
string3->SetText("message 3");

// Example 2: Depth-copy the children from one window to
        another.
extern ZafWindow *srcWindow, *dstWindow;
for (ZafWindowObject *object = srcWindow->First(); object;
        object = object->Next())
    dstWindow->Add(object->Duplicate());
```

```
void EndDraw(void);
```

See ZafWindowObject::BeginDraw().

```
ZafError Error(void) const;
ZafError SetError(ZafError error);
```

These functions get/set the error state of a window object. The types of errors that can be set are defined in `z_env.hpp`. Generally, however, only the following error values will be used by a ZafWindowObject object:

Error Value	Description
ZAF_ERROR_NONE	No error exists.
ZAF_ERROR_INVALID	The contents of the window object are invalid, meaning the object's value can be shown on the screen, but that the data is incorrect in the context of the application. For instance, the value 45 is a legitimate value for an integer field, but is invalid when used to describe the total number of days permitted in the month of February.

Error Value	Description
ZAF_ERROR_OUT_OF_RANGE	An error occurred while trying to convert data from one type to another or where the argument was too big for the return value. For example, a value of 10E+100 will not fit into a ZafInteger type field.
ZAF_ERROR_LESS_THAN_RANGE	
ZAF_ERROR_GREATER_THAN_RANGE	

In addition to the error types described above, error values greater than or equal to 10,000 are reserved for your use on user defined objects. The following code fragments show how to define and use your own error value with a derived data object.

```
// Define the class and constant.
const ZafError MY_APPLICATION_IS_DYING = 10000;
class AvailableMemory : public ZafInteger
...

// Create a new eject button.
AvailableMemory *memory = new AvailableMemory;

// Check for memory error.
if (memory->IntegerData()->Value() < someRandomValue)
    memory->SetError(MY_APPLICATION_IS_DYING);
...

// Check the error status.
if (memory->Error())
    exit(); // Die Hard!
```

```
virtual ZafEventType Event(const ZafEventStruct &event);
```

This function provides the core connection between event driven environment specific architectures and the object-oriented architecture supported by ZAF. This function receives four general types of events:

- ZAF system events represented by S_* messages.
- ZAF notification events represented by N_* messages.
- ZAF logical interpreted events represented by L_* messages.
- Environment specific events where event.type is E_OSEVENT and the environment specific message is specified by event.osEvent.

Here is the specific handling of these types of messages within the ZafWindowObject::Event() function.

Message	Handling
S_COMPUTE_SIZE	Causes the object to recalculate its zafRegion. The region is not modified if RegionType() is ZAF_INSIDE_REGION, but is recomputed by the parent's MaxRegion() function if any other type is specified.
S_CREATE	Causes the object to be created by dispatching an S_INITIALIZE, S_REGISTER_OBJECT and S_COMPUTE_SIZE message.
S_CURRENT/S_NON_CURRENT	Causes the Focus() member to be reset to "true" if S_CURRENT is sent, or "false" if the message is S_NON_CURRENT.
S_DEINITIALIZE	Causes the object to clear the screenID variable. This message updates the ZAF environment to match the deleted OS object.
S_DESTROY	Destroys the OS part of the object. This message dispatches an S_DEINITIALIZE message and causes the OS object to be destroyed using environment specific API calls.
S_DRAG_DEFAULT	These messages are dispatched to DragDropEvent(). For more information, see the ZafWindowObject::DragDropEvent() section of this chapter.
S_DRAG_MOVE	
S_DRAG_COPY	
S_DRAG_LINK	
S_DROP_DEFAULT	
S_DROP_MOVE	
S_DROP_COPY	
S_DROP_LINK	
S_BEGIN_DRAG	
S_END_DRAG	

Message	Handling
S_INITIALIZE	Causes the object to (1) ensure the zafRegion coordinates are converted for the native environment and (2) guarantee that the object's StringID() and NumberID() are valid. The algorithm used to set these values is to traverse to the root window, use the window's current number identification, update the root identification, then to set the StringID() according to the contents of the NumberID() (e.g. a NumberID() of 10 results in a default StringID() of "FIELD_10"). The NumberID() and StringID() variables are not reset if you have already associated a NumberID() and StringID() value with the object.
S_REDISPLAY	Causes the whole object to be redrawn. This message ensures that the background is cleared and all pertinent information updated on the display.
S_REDISPLAY_DATA	Causes only the data portion of the object to be redrawn. Since ZafWindowObject has no data portion, the default operation of this message is to request the redisplay of all the object except the border and shadow area (if any).
S_REDISPLAY_REGION	Causes a particular area of the field to be updated. The updated area is contained in event.region and should be relative to the object's region (0,0 left-top coordinate based on the interior area of the object to be redisplayed).
S_REGISTER_OBJECT	Causes the object to be registered with the operating environment by calling the virtual RegisterObject() function.
S_SIZE	Causes the object to be re-positioned, and the size to be modified according to event.region.

Message	Handling
S_HELP	Causes the help system to be called with a requested help context contained in HelpContext().
S_VSCROLL S_HSCROLL S_VSCROLL_SET S_HSCROLL_SET S_VSCROLL_CHECK S_HSCROLL_CHECK S_VSCROLL_COMPUTE S_HSCROLL_COMPUTE N_VSCROLL N_HSCROLL	These messages are dispatched to ScrollEvent(). For more information, see the ZafWindowObject::ScrollEvent() section of this chapter.
N_NON_CURRENT	Called just before the object loses focus. Causes userFunction to be called with N_NON_CURRENT as the message type. If the object cannot lose the input focus, the user-function should return a non-zero value. A zero indicates the object can lose the input focus.
N_CURRENT	Called just after the object gains focus. Causes userFunction to be called with N_CURRENT as the message type.
N_MOUSE_ENTER	Causes the QuickTip() and HelpObjectTip() information to be updated on the screen, if you have attached a ZafHelpTips device to the application's event manager.
N_MOUSE_LEAVE	Causes the QuickTip() and HelpObjectTip() information associated with this object to be removed from the screen, if you have attached a ZafHelpTips device to the application's event manager.

Message	Handling
L_HELP	Causes the ZafHelpSystem object to be called with the object's HelpContext().
E_KEY	Ignored by the ZafWindowObject class, but available for use with derived objects. This value is typically returned by ZafEventStruct::InputType() whenever the OS generates a key-press or key-release message. If interpreted, you should only look at the event.key.value and event.key.shiftState portion of the event structure.
E_MOUSE	<p>Ignored by the ZafWindowObject class, but available for use by derived objects. Typically, this value is returned by ZafEventStruct::InputType(). If interpreted after a call to LogicalEvent(), the event structure contains portable ZAF coordinates and shift-state information in event.position, event.rawCode, and event.modifiers. The specific contents of the event structure are:</p> <p>event.position</p> <p>Contains a ZAF “0,0,left-top” based coordinate that identifies a position within the receiving window object.</p> <p>event.rawCode</p> <p>Contains a combination of the ZAF defined mouse states: M_LEFT, M_LEFT_CHANGE, M_MIDDLE, M_MIDDLE_CHANGE, M_RIGHT, and/or M_RIGHT_CHANGE. These states are converted into OS specific parameters that describe the mouse on the native environment.</p>

Message	Handling
	<code>event.modifiers</code>
	Contains a combination of the ZAF keyboard shift-states including: S_SHIFT, S_RIGHT_SHIFT, S_LEFT_SHIFT, S_CTRL, S_ALT, S_CMD, S_SCROLL_LOCK, S_NUM_LOCK, S_CAPS_LOCK, S_INSERT, and S_OPT.

Events that are tagged with “event.type == E_OSEVENT” are dispatched directly to the underlying operating environment. For example, this code dispatches Windows events from within the Event() function:

```
CallWindowProc((WINDOWSPROC)GetClassLong(event.osEvent.hwnd,  
GCL_WNDPROC), event.osEvent.hwnd, event.osEvent.message,  
event.osEvent.wParam, event.osEvent.lParam));
```

All other events are considered “unknown” to the object and result in the S_UNKNOWN message being returned from the Event() function. Here is some sample code that shows how you could use, or override the default operation of ZafWindowObject::Event().

```
ZafEventType ZafString::Event(const ZafEventStruct &event)  
{  
    ...  
    default:  
        // Defer to the immediate base class.  
        ccode = ZafWindowObject::Event(event);  
        break;  
}  
  
ZafEventType ZafWindow::Event(const ZafEventStruct &event)  
{  
    ...  
    case S_REGISTER_OBJECT:  
        // Register the object.  
        RegisterObject();  
  
        // Register all of the children.  
        BroadcastEvent(event);  
}
```

```

ZafEventType ZafButton::Event(const ZafEventStruct &event)
{
    ...
case S_INITIALIZE:
    // Override the base class functionality.
    ccode = ZafWindowObject::Event(event);
    if (ButtonType() == ZAF_RADIO_BUTTON && !bitmapData)
    {
        int size = display->cellHeight / 2;
        bitmapData = radioBitmap;
        bitmapData->SetBitmap(size, size, bitmapData->Array());
    }
    else if (ButtonType() == ZAF_CHECK_BOX && !bitmapData)
    {
        int size = display->cellHeight / 2 - display->cellHeight /
            10; // motif algorithm.
        bitmapData = checkBitmap;
        bitmapData->SetBitmap(size, size, bitmapData->Array());
    }
}

```

```
static ZafEventManager *eventManager;
```

This is a static pointer to the application's event manager. It is initialized when the ZafWindowManager constructor is called and should not be modified. All derived window objects use this member when making requests to the event manager. They do not use the global zafEventManager member.

Here is a code snippet that shows how a derived window object would use eventManager to change the state of the mouse cursor:

```

MyDerivedWindowObject::MemberFunction(void)
{
    // Change the mouse image.
    eventManager->Event(DM_MOVE, E_MOUSE);
}

```

This member, as well as the static ZafWindowObject::display and ZafWindowObject::windowManager members are duplicate copies of the global variables zafDisplay, zafEventManager, and zafWindowManager. They are defined in the base ZafWindowObject class to allow advanced ZAF programmers the opportunity of removing the static definition, thus allowing particular instance variables to be associated with each window object; a feature useful in some multi-display and embedded system applications.

```
bool Focus(void) const;
```



```
bool SetFocus(bool focus);
```

The term “focus” has slightly different definitions, depending on the type of environment you are running under. A good general description of getting “focus” is simply specifying where keyboard and mouse events will be processed and identifying where visual queues, such as highlights or blinking cursors will be presented. When an object has focus, it can generally be distinguished as being at the “center of attention”, whereas, objects that do not have focus, are presented in a manner that does not distinguish them from other objects on the screen.

If you pass “true” as the argument to `SetFocus()`, the object you specified will become the new focal point on the screen. As mentioned earlier, this will cause a visual and interactive change in your application, so that the specified object is seen as both current, and as receiving all user interaction.

There are two proper ways to call `SetFocus()`. First, you can always call `SetFocus()` for the exact object you want to receive the focus.

```
// Give an object focus.  
object->SetFocus(true);
```

Second, you can call a parent class instance to give focus to a composite class such as a window or group, but allow the actual focus object to be determined by the window or group that is receiving the focus.

```
// Attach a vertical list to the window.  
ZafVerticalList *vtList = new ZafVerticalList(0, 0, 20, 5);  
for (int i = 0; i < 10; i++)  
    vtList->Add(new ZafString(0, 0, 20, "item", -1));  
window->Add(vtList);  
...  
  
// Give the vertical list focus.  
vtList->SetFocus(true);
```

When such a call is made to any type of object that has children, the focus is pushed down to the current child, or the last child that previously may have had the focus. In this manner, you can programmatically specify a parent object, but really have the focus be applied to a particular child. An extreme example of this would be to give the focus to a particular window, that may have many levels of children. The specification of the root window will ensure the window gets general focus (which may be distinguished by the user by a highlighted title bar), but will cause the focus to be progressively pushed down to

the window's current child, grandchild, etc. until a specific focus change is performed.

On most environments, you cannot pass "false" to an object that currently has the focus. The reason is that nearly all environments force a point of focus somewhere on the screen. For portability reasons, ZAF does not define the results of a SetFocus(false) function call.

The return value for Focus() and SetFocus() is the final, or current focus state of the object. Under normal circumstances, this value will be the value passed into the SetFocus() function, but may differ if the object is disabled, or does not allow focus changes for a particular reason.

```
virtual ZafWindowObject *FocusObject(void);
```

FocusObject() returns a pointer to the current object that has the keyboard input focus. The algorithm to determine focus is a depth traversal function that first determines if the focus is on, or within the specified object, then progressively works its way through children, grandchildren, etc. to find the current focus object. If the focus is on another object that is not in the specified object's inheritance tree, then the return value is null.

This function is useful when you want to determine exactly where the focus is in a given application. In DOS, one of the windows attached to the window manager will always have input focus. On other environments, such as Windows, Motif, and Macintosh, the focus may actually be on another application, thus rendering a return value of null.

(picture here)

Here are some simple code samples that shows how to determine the focus of an application, a window, and of a particular window object.

```
// Determine the application's focus.
if (zafWindowManager->FocusObject())
    ... // application has focus.

// Change the visual presentation of a window's current object.
case N_CURRENT:
    if (Current() && FocusObject() == Current())
        Current()->SetBackgroundColor(ZAF_CLR_YELLOW);
    ...

// Check the current object's focus.
if (FocusObject() == this)
    printf("Current focus is %s\n", Text());
```

```
ZafLogicalFont Font(void) const;  
virtual ZafLogicalFont SetFont(ZafLogicalFont font);
```

Fonts affect the visual presentation of an object's textual information. Each logical font contains implied information about the font, such as the font family, weight, slant, and point size. Here is a picture indicating the `Font()` values of three window objects.

(picture here)

`SetFont()` changes the logical font associated with the normal presentation of an instantiated object. The following predefined ZAF fonts are supported:

- ZAF_FNT_PARENT
- ZAF_FNT_DEFAULT
- ZAF_FNT_NULL
- ZAF_FNT_SMALL
- ZAF_FNT_DIALOG
- ZAF_FNT_APPLICATION
- ZAF_FNT_SYSTEM
- ZAF_FNT_FIXED

In addition to the pre-defined fonts described above, users can define and use their own logical fonts. For more information on these font specifications, and for more details on derived font entries, see the `ZafPaletteStruct` and `ZafDisplay` sections of this manual or `ZafWindowObject::UserPalette()`.

Here is some sample code that shows the correct use of `SetFont()`.

```
// Change the text font of the object.  
object->SetFont( ZAF_FNT_SMALL);  
  
// Check the current font of an object, and change where  
// necessary.  
if (object->Font() == ZAF_FNT_NULL)  
    object->SetFont(object->parent->Font());  
  
// Change an object's text font.  
object->SetAutomaticUpdate(false);  
object->SetBackgroundColor( ZAF_CLR_RED);  
object->SetTextColor( ZAF_CLR_WHITE);  
object->SetFont( ZAF_FNT_FIXED);  
object->SetAutomaticUpdate(true);
```

The return value for `Font()` and `SetFont()` is the final, or current font associated with the object. Under normal circumstances, this will be the font passed into

SetFont(), but may be different if the object does not allow for a particular type of font specification.

```
virtual ZafWindowObject *GetObject(ZafNumberID numberID);
virtual ZafWindowObject *GetObject(const ZafIChar
    *stringID);
```

These overloaded functions get a child object using the object's NumberID() or StringID() as the matching data. The algorithm for these functions is a depth first search of the children. For example, here is a partial listing of the ZafWindow and ZafWindowObject code used in GetObject:

```
ZafWindowObject *ZafWindow::GetObject(ZafNumberID matchID)
{
    // Try to match on the current object.
    ZafWindowObject *match = ZafWindowObject::GetObject(matchID);

    // All others are depth first searches.
    ZafWindowObject *object;
    for (object = SupportFirst(); object && !match; object =
        object->Next())
        match = object->GetObject(matchID);
    for (object = First(); object && !match; object = object-
        >Next())
        match = object->GetObject(matchID);

    // Return the matching item.
    return (match);
}

ZafWindowObject *ZafWindowObject::GetObject(ZafNumberID
    matchID)
{
    return ((numberID == matchID) ? this :
        ZAF_NULLP(ZafWindowObject));
}
```

Be careful not to associate the same NumberID() or StringID() with two children in the window's search hierarchy. Only the first matching object will be returned.

The following code shows the proper use of this overloaded function:

```
// See if an object exists.
if (window->GetObject("MyTable"))
    break;
```

```
// Use the contents of a found class.
ZafWindowObject *object = window->GetObject(ID_NAME_FIELD);
if (object)
    printf("Object found: %s\n", object->StringID());

// Check the condition of an object.
ZafButton *button = DynamicPtrCast(window-
    >GetObject("OK_BUTTON"), ZafButton);
if (button && button->Selected())
    windowManager->Add(new ZafWindow(0, 0, 40, 10));
```

```
ZafIChar *HelpContext(void) const;
virtual ZafIChar *SetHelpContext(ZafIChar *helpContext);
```

SetHelpContext() allows you to specify a particular help context with an object that will be shown in the ZafHelpSystem anytime a user moves to the object and presses the help key (<F1> on most environments).

(picture here)

The help context string needs to be a pre-defined context defined by the ZafHelpSystem. (For more information on creating help contexts see the ZafHelpSystem section of this manual.)

The return value for HelpContext() and SetHelpContext() is the current help context string associated with the object (null if no help context is associated with the object). This will always be the value passed into the SetHelpContext() function. The following code shows how to correctly use these functions.

```
// Associate a help context with the window's children.
window->Add(new ZafPrompt(2, 1, 8, "name:"));
ZafString *name = new ZafString(10, 1, 40, ZAF_NULLP(ZafIChar),
    100);
name->SetHelpContext("NameHelp");
window->Add(name);

window->Add(new ZafPrompt(2, 2, 8, "address:"));
ZafString *address = new ZafString(10, 2, 40,
    ZAF_NULLP(ZafIChar), 100);
address->SetHelpContext("AddressHelp");
window->Add(address);
```

```
const ZafIChar *HelpObjectTip(void) const;
```

```
virtual const ZafIChar *SetHelpObjectTip(const ZafIChar
    *helpObjectTip);
```

SetHelpObjectTip(), along with SetQuickTip() allows you to associate help messages with the run-time presentation of an object. HelpObjectTip() generally presents a “status-bar” update of the current window object and is generally more descriptive than a quick-tip. It is important to note that a ZafHelpTips device must be added to the event manager for help object tips to function (see ZafHelpTips for more information). Here is a picture that shows a help-object-tip associated with an instantiated ZafButton object as displayed on the window’s status bar.

(picture here)

The initialization of HelpObjectTip() is slightly different than that of SetQuickTip() because a receiving object must be specified to present the help-tip information. Here is some sample code the shows the initialization of both HelpObjectTip() and QuickTip() for a button object.

```
// Stage 1-Create the help-tip device.
ZafHelpTips *helpTip = new ZafHelpTips(D_ON, ZAF_HELPTIPS_BOTH);
zafEventManager->Add(helpTip);

// Stage 2-Set up the help-object tip.
ZafWindow *window = new ZafWindow(0, 0, 50, 10);
window->AddGenericObjects(new ZafStringData("File Operation"));
ZafStatusBar *helpBar = new ZafStatusBar(0, 0, 0, 1);
ZafString *helpBarString = new ZafString(0, 0, 32, "", 64);
helpBar->Add(helpBarString);
window->Add(helpBar);

helpTip->SetHelpObject(helpBarString);

// Stage 3-Create two objects with quick- and help-tip
information.
ZafButton *save = new ZafButton(25, 0, 20,
    ZAF_NULLP(ZafBitmapData), "Save");
save->SetQuickTip("Save the application.");
save->SetHelpObjectTip("Select this button to save the
    application.");
window->Add(save);

ZafButton *cancel = new ZafButton(0, 0, 20,
    ZAF_NULLP(ZafBitmapData), "Cancel");
cancel->SetQuickTip("Cancel the save operation.");
cancel->SetHelpObjectTip("Select this button to cancel the save
    operation.");
window->Add(cancel);
```

Note, the help-tip device created above (`helpTip`) received an associated help object (`helpBarString`). This object is called with the help-tip information (through the `SetText()` function) whenever an object needs to update the help-tip information presented to the window.

The return value for `HelpObjectTip()` and `SetHelpObjectTip()` is the current string associated with the object. Under normal circumstances, this will be the value passed into the `SetHelpObjectTip()` function.

```
static int InitialDelay(void);  
static int SetInitialDelay(int initialDelay);  
static int RepeatDelay(void);  
static int SetRepeatDelay(int repeatDelay);
```

These environment specific functions set the initial and repeat time period, in milliseconds, that a user must press a mouse button and wait, before a repeat signal is sent to a particular object. The most common use of these values is in advanced programming such as that found with the DOS and Motif implementations of `ZafScrollBar` and `ZafSpinControl` objects. Although these values are not generally set or used by programmers, the information presented here will help you understand the nuances of user interface.

When a user presses the mouse key over the down-arrow on a scrollbar, there is an initial delay before the text or visual presentation of an associated object begins a continuous scrolling motion. In a multi-line text field, the information will scroll one line up, then, after a short delay will begin repeating the scroll motion, as long as the mouse continues to be pressed over the down-arrow. The initial period of time between the down-click, and the auto-repeated motion is called the initial delay. Once the object begins scrolling, the initial delay is replaced by a repeat delay. It may not seem like there is a difference between these values, but when you analyze this interaction, you will notice there is a slight difference. Repeat delays are typically shorter than initial delays, giving the user time to end a clicking operation before the auto-repeat operation takes effect.

Here is some advanced code that shows how Motif hooks initial and repeat delays with the `AutoRepeat()` functionality of the `ZafButton` object.

```
ZafEventType ZafButton::Event(const ZafEventStruct &event)  
{  
    case L_BEGIN_SELECT:  
        // Set the repeat process.  
        if (AutoRepeatSelection() && SystemObject())  
        {
```

```

        lastMousePosition = event.position;
        if (intervalID)
            XtRemoveTimeOut(intervalID);
        intervalID = XtAppAddTimeOut(display->xAppContext,
            InitialDelay(),
            (XtTimerCallbackProc)Repeat, (XtPointer)this);
    }
    break;

case L_CONTINUE_SELECT:
    // Update the button information.
    if (AutoRepeatSelection() && SystemObject() && Depressed())
    {
        // Continue the repeat process.
        lastMousePosition = event.position;
        if (intervalID)
            XtRemoveTimeOut(intervalID);
        intervalID = XtAppAddTimeOut(display->xAppContext,
            RepeatDelay(),
            (XtTimerCallbackProc)Repeat, (XtPointer)this);
    }
    break;

case L_END_SELECT:
    // End the repeat process.
    if (intervalID)
        XtRemoveTimeOut(intervalID);
    intervalID = 0;
}

```

Note, the use of these variables is environment specific. On systems and classes where default initial and repeat values are used (e.g. all of Windows, ZafScrollBar support object on Motif, etc.) these values have no effect.

The return value for InitialDelay() and RepeatDelay() is the current time interval, in milliseconds, of the delay. These values are static, so a call to SetInitialDelay() and SetRepeat(delay), affect the time intervals on all objects that use the variables.

```

virtual bool IsA(ZafClassID compareID) const;
virtual bool IsA(ZafClassName compareName) const;

```

These overloaded functions add the const value ID_ZAF_WINDOW_OBJECT and string “ZafWindowObject” to the hierarchical chain of inheritance relationships. Thus, a ZafWindowObject object will not only match IsA() queries for ZafElement, but also for the added ZafWindowObject identification.


```
// Check for a window object.  
if (object->IsA(ID_ZAF_WINDOW_OBJECT))  
    break;
```

The value returned is “true” if the object is an instantiation of the `ZafWindowObject` class, or a derived window object class. Otherwise, the return value is “false.”

```
bool LinkDraggable(void) const;
```

See `ZafWindowObject::Draggable()`.

```
ZafLogicalEvent LogicalEvent(const ZafEventStruct  
    &event);
```

This function determines the “logical interpretation” of a native OS event, for use in ZAF programming. It should not be confused with the concepts contained in `DragDropEvent()`, `MoveEvent()`, and `ScrollEvent()`. In particular, this function looks at particular OS events, such as `WM_KEYDOWN` and `WM_KEYUP` events on Windows, `KeyPress` and `KeyRelease` events on Motif, and `keyDown` and `keyUp` events on Macintosh to determine their logical ZAF interpretation (`E_KEY`, `L_SELECT`, `L_DOWN`, etc.) The type of interpretation depends on the receiving object (e.g. `ZafString`, `ZafButton`) and the type of event being dispatched (e.g., `WM_KEYDOWN`, `MotionNotify`, `mouseUp`). Here is a partial list of the default event table associated with the `ZafWindow` and `ZafWindowObject` classes:

```
ZafEventMap ZAF_FARDATA ZafWindow::defaultEventMap[] =  
{  
    { L_NEXT, E_KEY, TAB, S_KEYDOWN },  
    { L_PREVIOUS, E_KEY, TAB, S_KEYDOWN | S_SHIFT },  
    { L_NONE, 0, 0, 0 }  
};  
ZafEventMap ZAF_FARDATA ZafWindowObject::defaultEventMap[] =  
{  
    { L_BEGIN_SELECT, E_MOUSE, M_LEFT | M_LEFT_CHANGE, 0 },  
    { L_CONTINUE_SELECT, E_MOUSE, M_LEFT, 0 },  
    { L_DOUBLE_CLICK, E_MOUSE, M_LEFT | M_LEFT_CHANGE,  
        S_DOUBLE_CLICK },  
    { L_END_SELECT, E_MOUSE, M_LEFT_CHANGE, 0 },  
    { L_VIEW, E_MOUSE, 0, 0 },  
    { L_NONE, 0, 0, 0 }  
};
```

When a user presses a key, or moves the mouse, an OS specific event is generated. This event has definitions that only apply to the currently running operating environment, but are of little use to ZAF programmers. LogicalEvent() matches these OS specific messages against class event tables, to produce a logical event. For instance, the pressing of a <tab> key produces the following information on Windows and Motif:

```
int ZafEventManager::Get(ZafEventStruct &event, ZafQFlags flags)
{
    #if defined(ZAF_Windows)
    MSG msg;
    if ((Blocked(flags) && !queueBlock.First()) ||
        (!queueBlock.Full() && PeekMessage(&msg, 0, 0, 0,
            PM_NOREMOVE)))
    {
        // Get a Windows message and place it in the Zinc event
        queue.
        GetMessage(&msg, 0, 0, 0);
        Put(ZafEventStruct(E_Windows, &msg));
    }
    #elif defined (ZAF_MOTIF)
    if ((Blocked(flags) && !queueBlock.First()) ||
        (!queueBlock.Full() && XtAppPending(ZafDevice::display-
            >xAppContext)))
    {
        // Block if necessary and process one X event.
        XEvent message;
        XtAppNextEvent(ZafDevice::display->xAppContext, &message);
        ZafEventStruct event(E_MOTIF, &message);
        Put(event, Q_BEGIN);
    }
    #endif
    ...
}

// Windows.
msg.message == WM_KEYDOWN
msg.wParam == 0x0009

// Motif.
message.type == KeyPress
message.key == XLookupString() which passes back a value of
    XK_Tab.
```

Keyboard definitions, such as TAB, SPACE, ESCAPE, etc. are defined in z_keymap.hpp and contain environment specific values that allow ZAF to

match OS specific information with ZAF const values, in order to determine what type of key is pressed:

```
#if defined(ZAF_Windows)
const ZafRawCode ESCAPE = 0x001B;
const ZafRawCode ENTER = 0x000D;
const ZafRawCode TAB = 0x0009;
const ZafRawCode SPACE = 0x0020;
const ZafRawCode BACKSPACE = 0x0008;
#elif defined(ZAF_MOTIF)
const ZafRawCode ESCAPE = XK_Escape;
const ZafRawCode ENTER = XK_Return;
const ZafRawCode TAB = XK_Tab;
const ZafRawCode SPACE = XK_space;
const ZafRawCode BACKSPACE = XK_BackSpace;
#endif
```

These raw OS values are then matched against <defaultMap>.rawCode to determine a match. If a match exists then <defaultMap>.logicalValue is returned.

There are many more details associated with LogicalEvent() that are not discussed in this section. Hopefully, you have a broad understanding of the concepts associated with LogicalEvent() and can just understand the use of this function in your application.

Here is some code that shows how the derived ZafDate and ZafTable use LogicalEvent() to determine the child movement.

```
ZafEventType ZafDate::Event(const ZafEventStruct &event)
{
    // Check for logical events.
    ZafEventType ccode = event.type;
    if (event.InputType() == E_KEY)
        ccode = LogicalEvent(event);

    switch (ccode)
    {
        case L_SELECT:
            ...
            break;
    }
}

ZafEventType ZafTable::Event(const ZafEventStruct &event)
{
    // Check for logical events.
```

```

ZafEventType ccode = LogicalEvent(event);

// Check for zinc events.
switch (ccode)
{
case L_BEGIN_SELECT:
...
break;
}
}

```

The ZafWindowObject class provides logical mapping for the following logical events:

Logical Event	Description
L_BEGIN_ESCAPE	Mapped in response to a right mouse button down-click event.
L_CONTINUE_ESCAPE	Mapped when the mouse moves while the right mouse button is still depressed.
L_END_ESCAPE	Mapped in response to a right mouse button up-click event.
L_BEGIN_SELECT	Mapped in response to a left mouse button down-click event.
L_CONTINUE_SELECT	Mapped when the mouse moves while the left mouse button is still depressed.
L_END_SELECT	Mapped in response to a left mouse button up-click event.
L_CANCEL	Mapped in response to an <escape> event.
L_DOUBLE_CLICK	Mapped when the left mouse button is quickly clicked twice in a row. The maximum time between the first up-click and the second down-click is determined by the native environment, or ZafWindowObject::doubleClickRate.
L_VIEW	Mapped when an unpressed mouse is positioned over a window object.

```

virtual ZafPaletteStruct LogicalPalette(ZafPaletteType
type, ZafPaletteState state);

```

This function returns a portable color/font description given a requested palette type and state. The allowed arguments for “ZafPaletteType type” are:

Argument	Description
ZAF_PM_OUTLINE	This requests a palette entry that contains border colors. This request should only be made if the object’s Bordered() flag is “true,” and when you want to draw an encompassing border around the object’s specified region. This value should not be used when drawing a 3-dimensional border around the object.
ZAF_PM_BACKGROUND	Used when you want to clear or draw to the background portion of the object. The background palette specifies a pattern, and foreground/background color that can be used in clearing operations.
ZAF_PM_FOREGROUND	Used when you want to draw graphic information, such as lines, rectangles, and ellipses within a window object. This palette is used after the background has been cleared, and additional information, such as a check-mark, or radio-button still need to be rendered on the display. Note, both the ZAF_PM_FOREGROUND and ZAF_PM_BACKGROUND colors are combined only when the fill pattern is a non-solid type fill (e.g., ZAF_PTN_INTERLEAVE_FILL).
ZAF_PM_TEXT	This requests a palette that has font, pattern, foreground and background entries that will be used in text drawing operations.
ZAF_PM_HOT_KEY	This requests an entry to be used when drawing the “hotkey” portion of a string value. Normally, this value is only used on text based environments, since GUI environments generally show hotkey information with an underline.

Argument	Description
ZAF_PM_LIGHT_SHADOW	This requests the light colors of a 3-dimensional shadow. This is used in conjunction with ZAF_PM_DARK_SHADOW to present a shadowed appearance on the object. When the object appears raised, this entry is used on the left and top sides of the object. If the object appears depressed, this entry is used for the right and bottom sides of the object.
ZAF_PM_DARK_SHADOW	This requests the dark colors of a 3-dimensional shadow. This is used in conjunction with ZAF_PM_LIGHT_SHADOW to present a shadowed appearance on the object. When the object appears raised, this entry is used on the right and bottom sides of the object. If the object appears depressed, this entry is used for the left and top sides of the object.
ZAF_PM_FOCUS	This requests a color value to be used when drawing the focus rectangle around a window object. Typically, this entry is used with ZAF_PM_ANY_STATE so the focus is drawn in a consistent color.
ZAF_PM_ANY_TYPE	

The associated values of “ZafPaletteState state” are generally determined at run-time (see the example below) but may include:

Value	Description
ZAF_PM_ANY_STATE	Specifying this value will match any palette request, as long as the specified type is ZAF_PM_ANY_TYPE or the value matches the palette map entry.
ZAF_PM_ACTIVE	This state is typically used when you want to show the window, or window object, in an active state; meaning the window has input focus and objects can be immediately selected or focused.
ZAF_PM_CURRENT	This state is typically used when you want to show that the object has the focus. This is particularly useful when you want to distinguish items that have the input focus in a more distinctive way than just drawing a focus rectangle.

Value	Description
ZAF_PM_INACTIVE	This is a special request that is used when the object distinguishes between a parent that does, or does not have the input focus. For example, ZafTitle is drawn with different colors that have a unique presentation to show the window either does or does not have the immediate input focus.
ZAF_PM_SELECTED	This state is used when the object's Selected() attribute is "true." Many objects, such as vertical and horizontal lists, show their "selected" children in a highlighted state, giving immediate visual queues to the user, that the object has been selected from among the list of siblings.
ZAF_PM_DISABLED	This state is used when the object's Disabled() attribute is "true." Typically, this palette contains a "dithered" font and color setting that shows the object in a diminished, or incapacitated state.
ZAF_PM_ENABLED	This state is used when the object's Disabled() attribute is "false." This palette typically contains the normal presentation of font and color settings.

The returned ZafPaletteStruct, is the class or instance palette that matches the type and state argument supplied by the programmer. You are guaranteed a valid palette definition as the return argument for LogicalPalette() because ZAF automatically provides default matching palettes for all their ZAF classes.

The palette structure contains the following information:

Palett Attribute	Description
lineStyle	This is the type of line (solid, dotted) that can be drawn.
fillPattern	This is the type of pattern (solid, interleaved) that can be used either in the clearing of background information, or in the display of text information.
colorForeground/ colorBackground	These are the foreground/background color pairs used on color systems when drawing the object's visual information.
monoForeground/ monoBackground	These are the foreground/background monochrome color pairs used on black/white systems when drawing the object's visual information.
font	This is the logical font to be used when rendering text information to the display.

Palett Attribute	Description
osPalette	This is an OS specific value that provides optimization of palette information. For instance, Motif uses objects called “graphic contexts” that contain information similar to ZafPaletteStruct. Whenever default color and font information is used with a Motif widget, the optimal drawing mechanism is through the graphic contexts, not ZAF’s palettes. This “opaque” handle allows ZAF to optimize drawing operations on Motif, while providing override information in the ZafPaletteStruct.

This function is normally used in conjunction with an object’s draw operations. In general, the type argument of the request does not change, but the state argument depends on the current run-time status of the object. Here is some sample code from the Windows implementation of ZafString::Draw() function.

```
ZafEventType ZafString::Draw(const ZafEventStruct &,
    ZafEventType ccode)
{
    // Begin drawing operation.
    ZafRegionStruct drawRegion = BeginDraw();

    ...
    // Set the text palette.
    ZafPaletteState state = PaletteState();
    display->SetPalette(LogicalPalette(ZAF_PM_TEXT, state));

    // Draw the text.
    display->Text(drawRegion.left + 1, drawRegion.top + 1,
        stringData->Text(), -1);
    ...
}
```

```
typedef ZafEventType
    (ZafWindowObject::*MemberUserFunction)(const
        ZafEventStruct &event, ZafEventType ccode);
MemberUserFunction memberUserFunction;
```

The variable memberUserFunction provides a pointer-to-member C++ capability at the window object level. This member is generally used in conjunction with userFunction to “hook” between functional techniques and the more powerful member replacement. To illustrate its use, let’s look at three default override member functions that are declared in ZAF:

- ZafButton::SendMessage(),

- `ZafString::DefaultValidateFunction()`
- `ZafWindowObject::DefaultUserFunction`.

Each one of these functions provides a specific type of interface that is used with derived window objects. For instance, the `ZafButton` member `SendMessage()` simply packages up a message and places it in the event queue, whenever the `L_SELECT` or `L_DOUBLE_CLICK` messages are received. The member `DefaultUserFunction()` calls the associated user-function, whenever focus changes from one object to another, or when a selection sequence is processed. And finally, `DefaultValidateFunction()` maintains all the functionality of `DefaultUserFunction()`, but also provides simple validation for dates, times, numbers, etc. whenever the user presses <enter> or tabs off the field.

Each member provides a unique mechanism for user callback, depending on the state and type of message being processed. As a developer, you can also replace `memberUserFunction` to give program specific handling of events through derived classes, rather than a flat function based interface (the architecture supported by the `userFunction` variable). It is important that you understand the subtle difference between the member pointer, and the simple function pointer methods supported by ZAF. To illustrate this difference, examine the following section of code:

```
class MyButton : public ZafButton
{
    MyDatabase *database;
public:
    MyButton(void);
    ZafEventType MyNotification(const ZafEventStruct &event,
                               ZafEventType ccode);
};

ZafEventType MyButton::MyNotification(const ZafEventStruct
                                     &event, ZafEventType ccode)
{
    database->CallTheDatabase(ccode);
    return (ccode);
}

ZafEventType MyCallback(MyButton *myButton, ZafEventStruct
                       &event, ZafEventType ccode)
{
    myButton->database->CallTheDatabase(ccode); // oops! database
    is private!
    return (ccode);
}

MyButton *button1 = new MyButton;
```

```
button1->userFunction = (ZafUserFunction)MyCallback;
MyButton *button2 = new MyButton;
button2->memberUserFunction =
    (MemberUserFunction)MyButton::MyNotification;
```

Obviously, this code is simplistic. Things could be moved around `MyCallback` could be made a static member of `MyButton`, database could be made public, etc. But hopefully you see the added benefit of using a pointer-to-member. These benefits include, but are not limited to:

- the class member function gives you real access to the data, at any level,
- the member does not require the use of a globally visible function,
- there are no typecasts necessary for the member pointer.

```
bool MoveDraggable(void) const;
```

See `ZafWindowObject::Draggable()`.

```
virtual ZafEventType MoveEvent(const ZafEventStruct
    &event);
```

This function provides filtered input of movement events, allowing you to reduce the code complexity typically associated with the `Event()` function. The following events can be dispatched to a derived object's `MoveEvent()` function:

- `L_LEFT`
- `L_RIGHT`
- `L_UP`
- `L_DOWN`
- `L_FIRST`
- `L_LAST`
- `L_NEXT`
- `L_PREVIOUS`
- `L_PGDN`
- `L_PGUP`.

A special warning accompanies this function. As described in other sections of this manual, `L_*` messages are not defined to be cross-platform portable. Thus, overriding this function requires specific knowledge of the environment on which you are running. For instance, the Motif implementation of ZAF provides special functionality in the `ZafVtList::MoveEvent()` function, which

allows you to intercept all of the L_* events, but the Windows implementation does not. You cannot guarantee the proper handling of an L_* message in MoveEvent() function, if you derive the base functionality from ZafVtList, since the environment specific implementations are different.

The following code snippet shows the derived functionality of MoveEvent() for Motif's implementation of ZafVtList.

```
ZafEventType ZafVtList::MoveEvent(const ZafEventStruct &event)
{
    ZafWindowObject *object = Current();
    ZafWindowObject *pObject = object ? object->Previous() :
        ZAF_NULLP(ZafWindowObject);
    ZafWindowObject *nObject = object ? object->Next() :
        ZAF_NULLP(ZafWindowObject);
    ...
    case L_UP:
    case L_PREVIOUS:
    case L_EXTEND_PREVIOUS:
        if (pObject)
            object = AdjustPrevious(pObject);
        break;

    case L_DOWN:
    case L_NEXT:
    case L_EXTEND_NEXT:
        if (nObject)
            object = AdjustNext(nObject);
        break;
    ...
}
```

The return value for MoveEvent() should be the passed event.type if processing is successful. Otherwise, the function should return the values S_ERROR or S_UNKNOWN indicating the object either detected an error on the message, or that the function did not recognize the specified message.

```
ZafWindowObject *Next(void) const;
```

This overloaded function adds a type-safe cast of “ZafWindowObject*” to the object while accessing the next sibling in a window’s list of children. The overload allows you to initialize and manipulate a list of associated window objects with the proper base type declaration. Here is a code snippet that shows the proper use of this overloaded member.

```
// Select all of the objects in the group.
for (ZafWindowObject *object = group->First(); object; object =
    object->Next())
    object->SetSelected(true);
```

```
bool Noncurrent(void) const;
virtual bool SetNoncurrent(bool noncurrent);
```

The term “non-current” specifies whether an object can receive system focus and subsequent keyboard input. If an object sets `SetNoncurrent(true)`, then the user will not be able to tab or set the focus on the object, but will still be able to use the mouse to select or click within the specified object. For instance, one common use of the `Noncurrent()` attribute is with scrollbars. Normally, the end-user clicks the mouse to “activate” up- and down-arrows, to select page-up and page-down options, or to grab the thumb control on the center portion of the scrollbar. The visual focus never moves to the scrollbar, nor does the scrollbar ever accept keyboard control. It simply modifies the visual presentation of an adjacent object, such as a multi-line text field. Another typical application is the use of `Noncurrent()` with toolbars. Toolbars typically have button children that allow the user to select application options (open, close, cut, copy, paste, etc.). As with scrollbar, the mouse is used to select one of the toolbar items, causing application changes. You may have noticed that the focus is never changed to a field inside the toolbar, but rather, still resides on another field within the window. The toolbar simply processes mouse information, never receiving focus or keyboard control. This is accomplished by calling `SetNoncurrent(true)` on the toolbar object.

The use of `SetNoncurrent()` is somewhat restricted. For instance, `ZafBorder`, `ZafTitle`, `ZafMinimizeButton`, `ZafMaximizeButton` never allow you to change the value of `Noncurrent()` (they automatically set this value to true). You are encouraged to refer to the associated section of the object you are creating to understand any restrictions or limitations on this function.

Here is some sample code that shows the correct use of these functions:

```
// Set the button's noncurrent state.
button->SetNoncurrent(true);

// Check all the object's noncurrent status.
for (ZafWindowObject *object = First(); object; object = object->Next())
    if (object->Noncurrent())
        printf("NO! Object %s does not allow keyboard focus.\n",
            object->StringID())
    else
```

```
printf("YES! Object %s allows keyboard focus.\n", object-
>StringID())

// This code has no effect since you cannot reset the noncurrent
state of a ZafPrompt object.
extern ZafPrompt *prompt;
prompt->SetNoncurrent(false); // error!
```

Note, setting the noncurrent attribute on a parent object causes all of the children to become noncurrent, even though their individual noncurrent states may not be “true.” SetNoncurrent() functionality is thus propagated to children, grandchildren, etc., but only through inheritance, not by value replacement (i.e. the child object’s noncurrent value is not reset to be the same as the parent’s value). Once the Noncurrent() state is set back to false, the input and focus aspects of children are restored.

The return value for Noncurrent() and SetNoncurrent() is the final, or current state of the object (true or false). Under normal circumstances, this value will be the value passed into the SetNoncurrent() function, but may differ if the derived class restricts its use.

```
virtual ZafWindowObject *NotifyFocus(ZafWindowObject
*object, bool focus);
```

This is an advanced function that is used to notify a parent object, or set of hierarchal objects, that the focus of a child is in the process of being changed. Under normal conditions, you will not need to explicitly call this function. Subsequent discussion of this function is intended for advanced ZAF programmers, and as adeptly stated by many reference manuals as not indented for the “faint of heart.”

The consequences of changing focus are fairly dramatic for most windows and window objects. For instance, changing the focus from one window to another, requires all of the ancestors of the old focus object to be notified of a focus change (their Focus() attribute changes to false), and all the ancestors of the new focus object to be notified of their focus change (their Focus() attribute changes to true). A fairly complicated endeavor when you consider all the ways in which an object may gain focus! NotifyFocus() provides a consistent method of notification for these affected objects.

The illustration below demonstrates one simple aspect of focus notification from one child object to another:

```
// Sample code to move the focus on another window.
window1->Current()->SetFocus(true); // Assume this is the base
state.
```

```
...
window2->Current()->SetFocus(true); // This focus change is
    described by the algorithm.
```

When window2 resets the focus for its Current() object, the following algorithm is executed:

- SetFocus() is called on the current object of window2. If a focus change is allowed by the current object, it proceeds to step 2. Otherwise the function ends and SetFocus() returns false.
- Current() calls NotifyFocus() for itself, beginning the process of notification. This is done with the following arguments:

```
NotifyFocus(this, true);
```

The return value for NotifyFocus() will be null if the object can obtain the focus, or non-null if another object will become “invalid” if the focus is moved. A non-null return value is the final chance of the “old focus” object to request a cancellation of focus change.

- NotifyFocus() calls its parent’s NotifyFocus() function as follows:

```
parent->NotifyFocus(this, true);
```

The “this” pointer is a pointer to itself, whereas “true” tells the parent object that the focus is being turned on for the object.

- The parent propagates the “focus” up, if it does not currently have the focus, or notifies the old focus object that it will lose the focus. Notification of focus change is made by sending the N_NON_CURRENT and S_NON_CURRENT messages to the old focus object’s Event() function, in the following manner:

```
if (Event(N_NON_CURRENT) == 0)
    Event(S_NON_CURRENT);
```

These messages allow the object to call associated user or validate functions, in preparation of losing the input focus.

- If the return “invalidObject” argument is null, the parent finalizes the objects focus by calling the object’s Event() function with S_CURRENT and N_CURRENT. The default operation of these messages is to reset the Focus() variable and to call any associated user-functions.

One way to view this algorithm is to envision a pyramid.

(picture here)

We start on a corner of the pyramid and walk all the way to its peak. This represents traversing all the non-focus objects in our parent’s hierarchy. Once we reach the top, we have reached the root object that currently contains the system focus. Now we must walk all the way down the other side of the pyramid to find the exact object that currently has the system focus. Once this is

accomplished, we begin walking back up the pyramid, systematically turning off the focus of the old focus object's parent hierarchy. This is done by sending `N_NON_CURRENT` and `S_NON_CURRENT` messages to the objects, allowing them to turn off their focus attributes, to call associated user-functions, etc. Once we reach the top of the pyramid, the focus value changes from "false" to "true" and all ancestors are systematically notified of their new focus state. The process ends when we reach the beginning spot on the pyramid.

It would be impractical to describe all the combinations and possible variations of `NotifyFocus()`. The example above is intended as a simplified explanation to the types of operations that are occurring whenever the `NotifyFocus()` function is called.

In order to fully explain the nature of `NotifyFocus()`, a partial code snippet from `ZafWindow::NotifyFocus()` is shown below:

```
ZafWindowObject *ZafWindow::NotifyFocus(ZafWindowObject
    *focusObject, bool setFocus)
{
    ...
    // Check for algorithm direction.
    ZafWindowObject *invalidObject = ZAF_NULLP(ZafWindowObject);
    if (!setFocus) // down
    {
        // Remove focus from the current branch.
        if (Current())
            invalidObject = Current()->NotifyFocus(focusObject, false);

        // Remove focus from the "this"
        if (!invalidObject)
            invalidObject = ZafWindowObject::NotifyFocus(focusObject,
                false);
    }
    else if (!focus) // up
    {
        // Recurse the entire focus path. Give this object focus.
        invalidObject = ZafWindowObject::NotifyFocus(this, true);

        // This window is now the root of the focus path. Transition
        focus
        // if focusObject doesn't already have it.
        if (!invalidObject && Current() && Current() != focusObject)
            invalidObject = Current()->NotifyFocus(this, false);

        // Set focus if validation succeeded.
        if (!invalidObject)
            ZafList::SetCurrent(focusObject);
    }
}
```

```

else // transition
{
    // This window is the root of the focus path. Transition focus
    // if focusObject doesn't already have it.
    if (Current() && Current() != focusObject)
        invalidObject = Current()->NotifyFocus(this, false);

    // Set focus if validation succeeded.
    if (!invalidObject)
        ZafList::SetCurrent(focusObject);
}
...
}

```

```

virtual ZafWindowObject *NotifySelection(ZafWindowObject
    *object, bool selected);

```

This is an advanced function that is used to notify a parent object, or set of hierarchal objects, that the selection state of a child is in the process of being changed. Under normal conditions, you should not call this function directly. A description of this function follows as “useful information” in your programming endeavors.

For simple objects, a call to SetSelected() simply changes the state of the object and reflects the change on the display. It does not, however, take into account the consequences of such an action on other siblings, or as it may affect the state of an application. In addition, since some simple objects (list and tree items) do not know what consequence the change of its state will have, it does not update its visual representation if its SystemObject() status is “false.” This aspect of selection is deferred to the parent’s NotifySelection() function.

Let’s briefly look at the selection process of two objects: a basic window and a vertical list to give you better insight into the selection process.

(picture here)

In a basic window, the mechanics of selection are straight forward. If SelectionType() is ZAF_MULTIPLE_SELECTION or ZAF_EXTENDED_SELECTION no additional processing is needed on the window’s children. The object simply marks itself as selected, calls parent->NotifySelection() which returns without action, and redisplay its changes. The selection process then continues within the context of the application.

In the case of ZAF_SINGLE_SELECTION, however, ZafWindow::NotifySelection() traverses all children to either turn off their selection status (if the selection state of the notify child is “true”), or to check for another selected object (if the child requests the selection status to be turned off). Note the

ZafWindow class simply calls SetSelected() on the children, no special OS processing is needed.

```
ZafWindowObject *ZafWindow::NotifySelection(ZafWindowObject
    *selectedObject, bool setSelected)
{
    ...
    if (setSelected && SelectionType() == ZAF_SINGLE_SELECTION)
    {
        // Deselect all objects except "selectedObject."
        for (ZafWindowObject *object = First(); object; object =
            object->Next())
            if (object->Selected() && object != selectedObject)
                object->SetSelected(false);
        ...
    }
    // Return the object selection.
    return (selectedObject);
}
```

In the case of ZafVtList, there is an important component added to the selection process; the native operating system's update of list items. If the vertical list is set with a selection type of ZAF_SINGLE_SELECTION, the list must not only turn off the selection status of all other selected children, but must also redisplay the child's visual information in accordance with proper API calls, to reflect the changed status on the display. For this object, the particular aspects of redisplay are environment specific.

For instance, Windows must send an LB_SETCURSEL message to the parent list of a single-select list item, a message that automatically causes the item to be redisplayed in its new state.

```
ZafWindowObject *ZafVtList::NotifySelection(ZafWindowObject
    *object, bool setSelected)
{
    ZafWindow::NotifySelection(object, setSelected);
    int index = Index(object);
    ...
    SendMessage(screenID, LB_SETCURSEL, (WPARAM)index, (LPARAM)0);
}
```

If the list is ZAF_MULTIPLE_SELECTION or ZAF_EXTENDED_SELECTION, however, it must process an LB_SETSEL message for the selected child, in order to redisplay correctly:

```
SendMessage(screenID, LB_SETSEL, (WPARAM)setSelected,
            (LPARAM)index);
```

Each environment has specific methods for accessing and manipulating native lists and list items. Thus, the rendering of these changes is overridden through environment specific implementations of the `ZafVtList::NotifySelection()` member function. (Note, the children will not automatically update themselves when `SetSelected(true/false)` is called, because they are marked by the list as non-system objects.)

The methods of selection for other derived windows is similar. If the object has a native implementation, API calls are intermixed with ZAF functionality to properly reflect changes to the object's status. Otherwise, the derived window defers the notification process to the base `ZafWindow` class.

Here is some sample code that shows the use of `NotifySelection()`.

```
bool ZafWindowObject::SetSelected(bool setSelected)
{
    ...
    // Check the object selection with its parent.
    ZafWindow *window = DynamicPtrCast(parent, ZafWindow);
    if (window)
        window->NotifySelection(this, setSelected);
    ...
}
```

The return value for `NotifySelection()` is a pointer to the current selected item. Typically, this is the object you passed to `NotifySelection()`, but may be different if an associated user-function or derived object does not allow selection of the item specified.

```
bool OSDraw(void) const;
virtual bool SetOSDraw(bool osDraw);
```

This is a special attribute function that combines with derived `Draw()` functions to “reconnect” OS specific display calls. Some operating environments provide special mechanisms for drawing native objects. These methods should not be overridden unless you have specific drawing needs that are not automatically handled by the native object. You should only clear `OSDraw()` when you derive a particular window object and need to override the `Draw()` functionality of that object. Here is some sample code that shows how this is done:

```
class MyButton : public ZafButton
{
```

```
virtual ZafEventType Draw(const ZafEventStruct &event,
    ZafEventType ccode);
    ...
};

MyButton::MyButton(int left, int top, int width, int height) :
    ZafButton(left, top, width, height, ZAF_NULLP(ZafBitmapData),
        ZAF_NULLP(ZafIChar))
{
    // Make sure Draw() gets the proper update calls.
    SetOSDraw(false);
    ...
}
```

Note that overloading the draw capabilities requires two operations: the definition of a derived `Draw()` function, and the clearing of the `OSDraw()` variable. If both steps are not taken, the results are undefined and environment dependent.

The return value for `OSDraw()` and `SetOSDraw()` is the final, or current draw attribute associated with the object (true or false). Under normal circumstances, this will be the value passed into the `SetOSDraw()` function, but may be different if the object does not allow you to override this functionality.

```
virtual OSWindowID OSScreenID(ZafScreenIDType type =
    ZAF_SCREENID) const;
```

See `ZafWindowObject::screenID`.

```
ZafPaletteState PaletteState(void);
```

This function indicates the current state of an instantiated object, in values recognized by palette computation functions such as `ZafWindowObject::LogicalPalette()` and `ZafPaletteData::GetPalette()`. The following values are inclusively defined by ZAF:

- `ZAF_PM_ANY_STATE` (0x0000)
- `ZAF_PM_ACTIVE` (0x0001)
- `ZAF_PM_CURRENT` (0x0002)
- `ZAF_PM_INACTIVE` (0x0004)
- `ZAF_PM_SELECTED` (0x0008)
- `ZAF_PM_DISABLED` (0x0010)
- `ZAF_PM_ENABLED` (0x0020)

This function is generally used within drawing operations to determine the current drawing state of an object. Here is some code from `ZafString::Draw()`, that shows the use of `PaletteState()` to determine the proper color to use in drawing text information.

```
ZafEventType ZafString::Draw(const ZafEventStruct &,
    ZafEventType ccode)
{
    // Begin drawing operation.
    ZafRegionStruct drawRegion = BeginDraw();

    ...
    // Set the text palette.
    ZafPaletteState state = PaletteState();
    display->SetPalette(LogicalPalette(ZAF_PM_TEXT, state));

    // Draw the text.
    display->Text(drawRegion.left + 1, drawRegion.top + 1,
        stringData->Text(), -1);
    ...
}
```

Note, the function call `LogicalPalette()` uses two variables to associate the proper color/font information: `ZAF_PM_TEXT` and `state`. `ZAF_PM_TEXT` is a “static” request that tells `LogicalPalette()` that we want to retrieve the proper color associated with the textual information of the object. The `state` argument, however, is “dynamic,” meaning its value must be determined at run-time, (in this case as determined by the return value of `PaletteState()` function) in order to obtain the correct color and font information.

For more information on palette settings, see `ZafWindowObject::UserPaletteData()` or the `ZafPaletteMap` section of this manual.

```
ZafWindowObject *Parent(void) const;
ZafWindowObject *SetParent(ZafWindowObject *parent);
```

These functions set or provide access to an object’s instance hierarchy. `Parent()`, combined with the `Next()`, `Previous()`, `ZafWindow::First()` and `ZafWindow::Last()` give programmers the ability to move bidirectionally within a window. Here is a diagram that shows these functional components.

(picture here)

Here are some code snippets that show the use of the `Parent()` function.

```
ZafWindowObject *ZafWindow::Add(ZafWindowObject *object,
    ZafWindowObject *position)
{
    // Make sure add is allowed.
    if (object->Parent())
        return (object);

    // Set object's parent.
    object->SetParent(this);
    ...
}

// Find the root window object.
ZafWindowObject *ZafWindowObject::RootObject(void)
{
    ZafWindowObject *object = this;
    while (object->Parent())
        object = object->Parent();
    return (object);
}

// Notify the parent of a pending action.
if (buttonToDisableWindow->Selected())
    buttonToDisableWindow->Parent()->SetDisabled(true);
```

Under normal circumstances, you will not use the `SetParent()` function, because the `Parent()` value is automatically set when the `ZafWindow::Add()` function is called.

The return value for `Parent()` and `SetParent()` is the current parent associated with the object. This value will always be the value passed into `SetParent()`.

```
bool ParentDrawBorder(void) const;
virtual bool SetParentDrawBorder(bool parentDrawBorder);
bool ParentDrawFocus(void) const;
virtual bool SetParentDrawFocus(bool parentDrawFocus);
bool ParentPalette(void) const;
virtual bool SetParentPalette(bool parentPalette);
```

These functions are used to defer drawing or the retrieval of color information from a child to its parent. In general, you should not set these values because they are automatically set by advanced ZAF objects such as `ZafVtList` and `ZafTree`. These list objects set the `Parent*()` values, of children added to them, to “true” in order to give all their children a consistent presentation to the screen. Here is a brief explanation of these functions.

SetParentDrawBorder() Causes the object to defer the border drawing operation to its immediate parent. In this case, the corresponding colors of the immediate object are ignored.

SetParentDrawFocus() Causes the object to defer the focus drawing operation to its immediate parent. As with **SetParentDrawBorder()**, setting this attribute to “true” causes the object to ignore any colors that correspond to its own focus rendering.

SetParentPalette() Causes the child to ignore its default class palette information and use the parent object’s class or instance palette information. Note, this flag is ignored if you specify a **UserPalette()**. The child only refers to parent information if the user palette does not contain the information necessary for the requested drawing operations. Normally, this operation is desired when all the children of an object (**ZafVtList**, **ZafTreeList**) need to be presented in a uniform manner.

Here is some sample code that shows the proper use of these functions.

```
ZafWindowObject *ZafVtList::Add(ZafWindowObject *object,
    ZafWindowObject *position)
{
    ...
    // Add the object to the list.
    object->SetSystemObject(false);
    object->SetParentPalette(true);
    ZafWindow::Add(object, position);
    ...
}
```

All of these functions return the current attribute associated with the object (true or false). Normally, this will be the original value passed to the **SetParent*()** function, but may be different if the derived window object does not allow changes to the **Parent*()** attributes.

```
ZafWindowObject *Previous(void) const;
```

This overloaded function adds a type-safe cast of “**ZafWindowObject ***” to the object while accessing the previous sibling in a window’s list of children. The overload allows you to initialize and manipulate a list of associated window objects with the proper base type declaration. Here is a code snippet that shows the proper use of this overloaded member.

```
// Clear all the entries in a window.
for (ZafWindowObject *object = window->Last(); object; object =
    object->Previous())
```

```
object->SetText(ZAF_NULLP(ZafIChar));
```

```
const ZafIChar *QuickTip(void) const;
virtual const ZafIChar *SetQuickTip(const ZafIChar
    *quickTip);
```

SetQuickTip(), along with SetHelpObjectTip() allows you to associate particular help messages with the run-time presentation of an object. QuickTip() is the “pop-up” portion of the ZafHelpTips object that appears below the mouse cursor anytime the mouse is idle and positioned over an object that has a QuickTip() string. It is important to note that a ZafHelpTips device must be added to the event manager for quick tips to function (see ZafHelpTips for more information). Here is a picture that shows a quick-tip text associated with an instantiated ZafButton object.

(picture here)

The return value for QuickTip() and SetQuickTip() is the current string associated with the object. This will always be the value passed into the SetQuickTip() function. Here is a code snippet that shows the correct usage of these functions:

```
// Create two objects with quick-tip information.
ZafButton *save = new ZafButton(0, 0, 20,
    ZAF_NULLP(ZafBitmapData), "Save");
save->SetQuickTip("Save the application.");

ZafButton *cancel = new ZafButton(25, 0, 20,
    ZAF_NULLP(ZafBitmapData), "Cancel");
cancel->SetQuickTip("Cancel the save operation.");

static ZafElement *Read(const ZafIChar *name,
    ZafObjectPersistence &persist);
```

This static function defines a pointer to the persistent constructor (see ZafWindowObject::ZafWindowObject()) which reads a window object from a persistent file. This function should not be used directly with object construction. Rather, it is used by the ZafObjectPersistence class to allow the run-time determination of window object constructors. Here is a portion of the default ZafObjectPersistence table used when constructing a persistence object along with code that constructs a persistent window:

```
ZafObjectPersistence::ObjectConstructor
    ZafObjectPersistence::defaultObjectConstructor[] =
{
```

```

// --- Window objects ---
{ 0, ID_ZAF_BIGNUM, ZafBignum::className, ZafBignum::Read },
{ 0, ID_ZAF_BORDER, ZafBorder::className, ZafBorder::Read },
{ 0, ID_ZAF_BUTTON, ZafButton::className, ZafButton::Read },
{ 0, ID_ZAF_WINDOW, ZafWindow::className, ZafWindow::Read },
...
// --- End-of-array ---
{ 0, ID_END, 0, 0 }
};

// Load a persistent window.
ZafStorage *storage = new ZafStorage("myfile.dat");
ZafObjectPersistence persist(storage);
windowManager->Add(new ZafWindow("MyWindow", persist));

```

The return value for Read() is a newly instantiated object. If an error occurred during the creation of the object, the object's Error() value will be ZAF_ERROR_CONSTRUCTOR or ZAF_ERROR_FILE_READ.

```

void Redisplay(void);
void RedisplayData(void);

```

These two functions send appropriate ZAF messages (S_REDISPLAY and S_REDISPLAY_DATA) that cause the object to either redisplay the data portion of its area (e.g. The ZafString field redisplays only the text associated with the object) or the entire object's region.

```

ZafRegionStruct zafRegion;
virtual ZafRegionStruct Region(void) const;
virtual void SetRegion(const ZafRegionStruct &region);

```

This function and member gives the current position and size of a window object. In ZAF, all regions are defined to be positioned on a "0,0 left-top" based coordinate system, the coordinate system either specified as "client" based relative to their parent if they are normal window objects, or based on a "frame" area of their parent if they are support objects. The following pictures show the position and size of several ZAF objects within a simple window that contains both a frame and client area.

(picture here)

The zafRegion values for more complex objects also fit within the "frame/client" specification described above. Thus, the 0,0 left-top based coordinate not only applies to top-level windows, but also to sub-windows contained within a parent object.

(picture here)

Although this member is publicly available, it is recommended you never change its contents directly. Rather, the constructor for each object, the message `S_SIZE`, and the member functions `ZafWindowObject::SetRegion()`, `ZafWindowObject::SetCoordinateType()` and `ZafWindowManager::Center()` can be used to effect a change in the `zafRegion` structure. Here are some code snippets that show the various operations that can be used to modify the size and position of a window object.

```
// Create a string and reset its coordinate type.
// These calls set:
// string->zafRegion.left = 20,
// string->zafRegion.top = 20,
// string->zafRegion.right = 119,
// string->zafRegion.bottom depends
//   ZafCoordinateStruct::cellHeight, and
// string->zafRegion.coordinateType = ZAF_PIXEL.

ZafString *string = new ZafString(20, 20, 100, "string", 50);
string->SetCoordinateType(ZAF_PIXEL);

// Reset the position of a sub-object.
ZafEventStruct event(S_SIZE);
event.region.left = event.region.top = 100;
event.region.right = 600;
event.region.bottom = 400;
object->Event(event);

// Center a window on the screen.
zafWindowManager->Center(window);
zafWindowManager->Add(window);

ZafRegionType RegionType(void) const;
virtual ZafRegionType SetRegionType(ZafRegionType
    regionType);
```

SetRegionType() is used to specify the type of area that an object occupies within its parent. The initial region is defined by zafRegion, which is used or modified according to the following specifications:

Region Type	Description
ZAF_INSIDE_REGION	Tells the parent object that the object overlays its region within the parent according to the object's zafRegion member. This is the default type for most ZAF objects and is commonly referred to as a "field" object. Note, this type of region allows multiple objects to be overlaid on the same position in a window at one time. The presentation of overlapping window object's is environment specific. While not normally used in such a manner, it clarifies the non-ownership relation of the object with its specified zafRegion. Here is a picture that shows a normal window objects that use the ZAF_INSIDE_REGION type: (picture here)

Region Type	Description
ZAF_AVAILABLE_REGION	Specifies ownership of a particular screen area within its parent. A simple way of thinking about this type of region is to imagine an object that “hogs” or “reserves” a particular area of a window, not allowing other sibling objects to occupy the same area. Another way to think about this attribute is in conjunction with geometry management. Setting this flag would be equivalent to specifying geometry attributes that “pinned” the left, top, right, and bottom areas of the object to specific positions within its parent window and disallowed any other object to occupy the same area on the window. Here is a simple picture and code snippet that shows how the specification of ZAF_AVAILABLE_REGION or specific ZafConstraints could render a similar screen representation.

(picture here)

```
// Create an object with two region
specifications.
text->SetRegionType
    (ZAF_AVAILABLE_REGION);

// Do the same thing with geometry
management.
gmgr->Add(new ZafAttachment(text,
    ZAF_ATCF_LEFT));
gmgr->Add(new ZafAttachment(text,
    ZAF_ATCF_TOP));
gmgr->Add(new ZafAttachment(text,
    ZAF_ATCF_RIGHT));
gmgr->Add(new ZafAttachment(text,
    ZAF_ATCF_BOTTOM));
```

The ZAF_AVAILABLE_REGION attribute is mainly associated with support objects, but can also be associated with child objects such as ZafText and ZafVtList when you want the object to occupy all the remaining area of the window. Here are two additional pictures that show use of ZAF_AVAILABLE_REGION on support and normal objects.

(picture here) (picture here)

Region Type	Description
ZAF_OUTSIDE_REGION	<p>Specifies ownership of an area directly outside the specified zafRegion. This setting is similar to ZAF_AVAILABLE_REGION, but the ownership area is defined to be outside the zafRegion, not inside. Currently, only the ZafBorder class uses this type of region. Here is a picture that clarifies the use and meaning of this type.</p> <p>(picture here)</p> <p>The return value for RegionType() and SetRegionType() is the final, or current attribute associated with the object. Under normal circumstances, this will be the value passed into the SetRegionType() function but can differ if the specified object does not allow the resetting of this attribute.</p> <p>Here is some sample code that shows the correct use of these functions.</p> <pre>// Create a text view window. ZafWindow *window = new ZafWindow(0, 0, 50, 10); window->AddGenericObjects(new ZafStringData("View Text")); ZafText *text = new ZafText(0, 0, 0, 0, "text", 1000); text-> >SetRegionType(ZAF_AVAILABLE_REGION); window->Add(text); ZafEventType ZafWindowObject::Event(const ZafEventStruct &event) { ... case S_COMPUTE_SIZE: if (RegionType() != ZAF_AVAILABLE_REGION) zafRegion = parent ? parent- >MaxRegion(this) : windowManager- >MaxRegion(this); break; }</pre>

```
static int RepeatDelay(void);
```

See ZafWindowObject::InitialDelay().

```
ZafWindowObject *RootObject(void) const;
```

This function traverses an object's Parent() hierarchy to find the root ZAF object (typically a ZafWindow or derived window). The return value is the root, or final object in the instance hierarchy that can be attached to the window manager or MDI child. Here is some code that shows effective use of the RootObject() function.

```
ZafPaletteState ZafWindowObject::PaletteState(void)
{
    // Check for the matching palette state.
    ZafPaletteState state = ZAF_PM_ANY_STATE;
    if (RootObject()->Focus())
        state |= ZAF_PM_ACTIVE;
    ...
}

// Determine if my non-MDI window is attached to the window
// manager.
extern ZafButton *myButton;
ZafWindowObject *root = myButton->RootObject();
if (windowManager->Index(root) == -1)
    printf("My window is not attached to the window manager\n");
else if (root->Visible())
    printf("%s is visible to the user\n", root->StringID());
```

```
OSWindowID screenID;
```

```
virtual OSWindowID OSScreenID(ZafScreenIDType type =
    ZAF_SCREENID) const;
```

This variable and virtual function provide access to OS specific information associated with a window object. The screenID value represents a “hook” from the ZAF class hierarchy, to the underlying operating environment. The following definitions apply to screenID:

Platform	Description
MS Windows	Represents an HWND. For example, the Windows implementation of ZafString creates an “EDIT” with a call to the Windows API CreateWindowEx().

Platform	Description
Motif	Represents a Widget. For example, screenID for a ZafString object corresponds to a derived Motif “XmTextField” widget; ZafWindow corresponds to a derived “XmBulletinBoard” widget, and ZafButton corresponds to a derived “XmPushButton” widget.
Macintosh	Represents a class-specific API reference. For example, the Macintosh implementation of ZafString creates a TEHandle object by calling the Macintosh API TENew().
DOS	Represents a unique screen identifier generated by the display. This environment has no specific OS hook. The value is simply used to reserve a region of the display.

Normally, you will not use screenID directly in your application. For advanced ZAF programming, however, this provides a fundamental mechanism for deriving objects that ZAF does not provide, but that may have native implementation on certain GUI environments.

Here is some sample code snippets that show how screenID is used in the implementation of S_REDISPLAY to make native API calls on Windows and Motif.

```
// Windows implementation of S_REDISPLAY.
if (SystemObject())
    RedrawWindow(screenID, NULL, 0, RDW_INVALIDATE | RDW_ERASE |
        RDW_ALLCHILDREN);
else
{
    // Get the object region in OS coordinates.
    ZafRegionStruct region = parent ? parent-
        >ConvertToDrawRegion(this) :
        windowManager->ConvertToDrawRegion(this);
    RECT rect;
    region.ExportPoint(rect);

    // Invalidate the object region.
    RedrawWindow(screenID, &rect, 0, RDW_INVALIDATE | RDW_ERASE |
        RDW_ALLCHILDREN);
}

// Motif implementation of S_REDISPLAY.
if (SystemObject())
    XClearArea(XtDisplay(screenID), XtWindow(screenID), 0, 0, 0,
        0, true);
else
{
```

```
// Dismiss border area for non-system objects (Complex motif
// object
// such as list and tool-bar overlap their children by the
// border
// area of their children).
ZafRegionStruct updateRegion = zafRegion;
updateRegion -= ZAF_BORDER_WIDTH;
XClearArea(XtDisplay(screenID), XtWindow(screenID),
            updateRegion.left, updateRegion.top, updateRegion.Width(),
            updateRegion.Height(), true);
}
```

```
virtual ZafEventType ScrollEvent(const ZafEventStruct
    &event);
```

This function provides filtered input of scrolling events, allowing you to reduce the code complexity typically associated with a large, and generalized Event() function. The following events can be dispatched to a derived object's ScrollEvent() function:

- N_VSCROLL
- N_HSCROLL
- S_VSCROLL
- S_HSCROLL
- S_VSCROLL_SET
- S_HSCROLL_SET
- S_VSCROLL_CHECK
- S_HSCROLL_CHECK
- S_VSCROLL_COMPUTE
- S_HSCROLL_COMPUTE

Whenever an object receives a scroll message, the messages are first intercepted in the object's Event() function. To allow for more encapsulation and a more efficient handling of similar scroll functionality, ZAF defines a virtual ScrollEvent() which is called by the base ZafWindowObject class whenever one of the aforementioned messages is generated.

The following code snippet shows the derived functionality of ScrollEvent() for Motif's implementation of ZafVtList.

```
ZafEventType ZafVtList::ScrollEvent(const ZafEventStruct &event)
{
    ...
    case S_VSCROLL_CHECK:
```

```

// Scroll the object into view.
{
ZafScrollStruct vScroll;
vScroll.delta = 0;
if (object->zafRegion.top < firstVisible->zafRegion.top)
{
// Determine the downward scrolling delta.
ZafWindowObject *temp = object;
while (temp && temp != firstVisible)
{
vScroll.delta--;
temp = temp->Next();
}
}
else if (object->zafRegion.bottom > lastVisible-
>zafRegion.bottom)
{
// Determine the upward scrolling delta.
ZafWindowObject *temp = object;
while (temp && temp != lastVisible)
{
vScroll.delta++;
temp = temp->Previous();
}
}
}

// Scroll the list elements or adjust the values.
if (vScroll.delta)
ScrollEvent(ZafEventStruct(S_VSCROLL, 0, vScroll));
}
}

```

The return value for `ScrollEvent()` should be the passed event.type if processing is successful. Otherwise, the function should return the values `S_ERROR` or `S_UNKNOWN` indicating the object either detected an error on the message, or that the function did not recognize the specified message.

```

bool Selected(void) const;
virtual bool SetSelected(bool selected);
virtual bool ToggleSelected(void);

```

The term selected, specifies a programming state where an object becomes peculiar or “stands out” within its context with respect to other sibling objects. For example, you may want to create a vertical list that allows end-users to select a set of files to be deleted from a directory. If the user clicks the mouse over a particular item in the list, the item becomes selected. When the user fin-

ishes his/her selection, a set of programming code traverses the list of files to determine which items the user wants deleted. This is done by looking at the selected state of a particular item and deleting the associated file when its selected state is set to true.

If you pass true as the argument to `SetSelected()`, the object will become selected. In addition, setting this attribute causes the object to notify its parent, such as a group, list, or window, to tell the parent of its intended changed state. The parent then determines if other siblings need to have their states changed in response to this sibling's new state. Generally their selection states will be changed to false if the parent does not allow multiple objects to be selected at one time (specified by the `ZafWindow::SelectionType()` member).

If, on the other hand, you specify false as the argument to `SetSelected()`, the object will turn off its selected state, will notify its parent that the state has changed, and will then give the parent final control to determine whether the specified object can really be de-selected within its context to other sibling objects.

The return value for `Selected()` and `SetSelected()` is the final, or current selected state of the object. Under normal circumstances, this value will be the value passed into the `SetSelected()` function, but may vary if the parent object does not allow the de-selection or automatic selection of a particular object.

The following code shows the proper use of these functions:

```
// Modify the selected attribute on some combo-boxes.
if (userHasPhone)
    phone->SetSelected(true);
if (userKnowsPassword)
    password->SetSelected(true);
if (userHasOfficeKeys)
    keys->SetSelected(true);
...
userHasPhone = phone->Selected();
userKnowsPassword = password->Selected();
userHasOfficeKeys = keys->Selected();

virtual const ZafIChar *Text(void);
virtual ZafError SetText(const ZafIChar *text);
```

These functions allow you to retrieve or change the textual information associated with an object. The base `ZafWindowObject` class contains no textual information, but many derived objects such as `ZafString`, `ZafDate`, `ZafGroup`, etc., do contain textual input or output information. Here is a picture that identifies the textual information associated with various window objects.

(picture here)

The exact implementation of these functions depends on the derived object, but here are a few explanations of its use with some common ZAF objects:

ZafString Resets the string according to the specified “text” parameter. The argument is accepted “as is” and will only be truncated if the specified string is larger than the buffer allocated by the instantiated ZafString object (as determined by ZafString::MaxLength()).

ZafDate Interprets the contents of the string based on its internal ZafDate::InputFormatData(). The value is thus “filtered” by the ZafDate object, then presented to the screen in a format consistent with the ZafDate::OutputFormatData(). For example, a partial “dec 95” string may be formatted as “December 1, 1995.”

ZafFormattedString Interprets the contents of the string as either “compressed,” or “expanded” data. For instance, the string value “8017858900” may be re-formatted to be “(801) 785-8900,” as seen by the user.

The return value for Text() is a const pointer, which indicates the current textual information associated with the object. Under normal circumstances, this value will be the same as the value passed to SetText(), but may differ if the object filters the text information in a manner similar to that described with the ZafDate and ZafFormattedString object definitions given above.

The return value for SetText() is an error indicator. The following general error conditions apply to window objects:

Error	Description
ZAF_ERROR_NONE	The value passed to SetText() was accepted without any errors.
ZAF_ERROR_INVALID	The value passed to SetText() was invalid for the receiving object. An example of this type of error would be passing “10:00am” to a ZafDate object.

Here is some sample code that shows the correct usage of Text() and SetText() with derived window objects.

```
// Reset the string information.
if (strcmp(string->Text(), "Stop"))
    string->SetText("Stop");

// Reset the date.
date->SetText("January 1, 2001");

// Clear all the editable object fields of a window.
```

```
for (ZafWindowObject *object = First(); object; object = object->Next())
    if (object->IsA(ID_ZAF_STRING) && !object->Disabled())
        object->SetText("");

ZafLogicalColor TextColor(ZafLogicalColor *color =
    ZAF_NULLP(ZafLogicalColor), ZafLogicalColor *mono =
    ZAF_NULLP(ZafLogicalColor));

ZafLogicalColor SetTextColor(ZafLogicalColor color,
    ZafLogicalColor mono = CLR_NULL);
```

Text color is the color associated with the textual presentation of an object. Here is a picture indicating the TextColor() area of three window objects.
(picture here)

SetTextColor() changes the foreground color associated with the normal presentation of an instantiated object. There are two types of colors that can be passed to SetTextColor(): a color value and a monochrome value. The first parameter specifies the color for normal operation, the second specifies the black/white value for monochrome or black/white modes of operation. Here is a list of predefined ZAF color values:

Color Values	Monochrome Values
ZAF_CLR_PARENT	ZAF_MONO_PARENT
ZAF_CLR_DEFAULT	ZAF_MONO_DEFAULT
ZAF_CLR_NULL	ZAF_MONO_NULL
ZAF_CLR_BACKGROUND	ZAF_MONO_BACKGROUND
ZAF_CLR_BLACK	ZAF_MONO_BLACK
ZAF_CLR_BLUE	ZAF_MONO_DIM
ZAF_CLR_GREEN	ZAF_MONO_NORMAL
ZAF_CLR_CYAN	ZAF_MONO_WHITE
ZAF_CLR_RED	ZAF_MONO_HIGH
ZAF_CLR_MAGENTA	
ZAF_CLR_BROWN	
ZAF_CLR_LIGHTGRAY	
ZAF_CLR_DARKGRAY	
ZAF_CLR_LIGHTBLUE	
ZAF_CLR_LIGHTGREEN	
ZAF_CLR_LIGHTCYAN	
ZAF_CLR_LIGHTRED	

Color Values	Monochrome Values
ZAF_CLR_LIGHTMAGENTA	
ZAF_CLR_YELLOW	
ZAF_CLR_WHITE	

In addition to the pre-defined colors described above, users can define and use their own logical colors. For more information on these color specifications, and for more details on derived color entries, see the `ZafPaletteStruct` and `ZafDisplay` sections of this manual or `ZafWindowObject::UserPalette()`.

Here is some sample code that shows the correct use of `SetTextColor()`.

```
// Change the text color of the object.
object->SetTextColor(ZAF_CLR_BLACK, ZAF_MONO_BLACK);

// Check the current color of an object, and change where
// necessary.
if (object->TextColor() == ZAF_CLR_NULL)
    object->SetTextColor(object->parent->TextColor());

// Change an object's text and background color.
object->SetAutomaticUpdate(false);
object->SetBackgroundColor(ZAF_CLR_RED);
object->SetTextColor(ZAF_CLR_WHITE);
object->SetAutomaticUpdate(true);
```

The return value for `TextColor()` and `SetTextColor()` is the final, or current color value associated with the object. Under normal circumstances, this will be the color value passed into the `SetTextColor()`, but may be different if the object does not allow for a particular type of color specification or if the system is running in black/white mode.

```
typedef ZafEventType (*ZafUserFunction)(ZafWindowObject
    *, ZafEventStruct &, ZafEventType);
ZafUserFunction userFunction;
ZafUserFunction UserFunction(void) const;
virtual ZafUserFunction SetUserFunction(ZafUserFunction
    userFunction);
```

These functions provide a callback mechanism for operation with an instantiated window object. When the `userFunction` variable is set, the programmer does not need to derive an object to implement multiple types of selection, coloration, or application control. For instance, the `ZafButton` class is frequently

used to “Accept” changes to a dialog window, to “Cancel” changes, or the get “Help” for a specified operation. In traditional function based programming, these methods would be implemented using Accept(), Cancel(), and Help() functions. The use of userFunction allows you to chain these traditional methods of programming with a derived window object.

```
// Hook up normal functions.
ZafEventType Accept(ZafWindowObject *object, ZafEventStruct &,
    ZafEventType ccode)
{ ... }

ZafEventType Cancel(ZafWindowObject *object, ZafEventStruct &,
    ZafEventType ccode)
{ ... }

ZafEventType Help(ZafWindowObject *object, ZafEventStruct &,
    ZafEventType ccode)
{ ... }

// Connect the user functions.
button1->SetUserFunction(Accept);
button2->SetUserFunction(Cancel);
button3->SetUserFunction(Help);
```

The default callback sequence of an object is defined as follows:

- When Focus() changes from “true” to “false” the associated user-function is called with an N_NON_CURRENT message. When this event occurs, the user has either tabbed onto another field, or has moved the focus to another object using the mouse.
- When Focus() changes from “false” to “true” the associated user-function is called with an N_CURRENT message. When this event occurs, the user has selected the receiving object, moving the focus to the object.
- When the user presses a selection key (usually the space bar, or <enter> key) the user-function is called with an L_SELECT message. The contents of the event argument will be an event.InputType() of E_KEY, signifying that a key event caused the selection.
- When the user clicks the mouse button while being positioned over the object the user-function is called with an L_SELECT message. When this occurs, the contents of the event argument will contain an event.InputType() of E_MOUSE, signifying that a mouse event caused the user-function to be called.
- When the user double-clicks on the object the user-function is called with L_DOUBLE_CLICK. The contents of the event structure will be of type E_MOUSE, signifying that a mouse event caused the user-function to be called.

The user-function should return 0 on success, and if some error occurred, anything non-zero should be returned. Here are several sample code snippets that show the correct use of the userFunction variable.

```
// Specify the callbacks.
ZafEventType MyStringCallback(ZafWindowObject *object,
    ZafEventStruct &, ZafEventType ccode)
{
    if (ccode == N_NON_CURRENT)
        object->SetText("Oh No! I'm losing focus.\n");
    else if (ccode == N_CURRENT)
        object->SetText("Yea! I'm back in control.\n");
    return (0);
}

ZafEventType MyButtonCallback(ZafWindowObject *object,
    ZafEventStruct &, ZafEventType ccode)
{
    if (ccode == L_SELECT && object->Selected())
        object->SetBackgroundColor(ZAF_CLR_RED);
    else if (ccode == L_SELECT)
        object->SetBackgroundColor(ZAF_CLR_DEFAULT);
    else if (ccode == L_DOUBLE_CLICK && object->Selected())
        object->SetText("OK");
    else if (ccode == L_DOUBLE_CLICK)
        object->SetText("Cancel");
    return (0);
}

// Set the user-functions.
stringField->SetUserFunction(MyStringCallback);
buttonField->SetUserFunction(MyButtonCallback);
```

```
void *userObject;
```

This is a void pointer, reserved for your use during the run-time operation of your application. ZAF sets this value to null in the ZafWindowObject constructor, but does not evaluate or modify its contents during the instantiated object's run-time operation. Since this is a void pointer, you must dynamically cast the object in your application. Here is some sample code, showing two possible implementations of the userObject member.

```
// Derived object implementation.
class MyObject : public ZafButton
{
public:
```

```
MyDatabase *Database(void) const { return
    (dynamic_cast<MyDatabase *>(userObject)); }
MyDatabase *SetDatabase(MyDatabase *database) { userObject =
    database; return (database); }
};

// User callback implementation.
ZafEventType MyUserFunction(ZafWindowObject *object,
    ZafEventStruct &, ZafEventType ccode)
{
    MyDatabase *database = dynamic_cast<MyDatabase *>(object-
        >userObject);
    ...
    return (ccode);
}

ZafPaletteData *userPaletteData;
ZafPaletteData *UserPaletteData(void) const;
ZafPaletteData *SetUserPaletteData(ZafPaletteData
    *userPaletteData);
```

These functions provide an instance override for default object color and font attributes. On each environment, ZAF coordinates its color and font rendering with the underlying operating environment, giving you a native looking application whether you are running on Windows, Motif, Macintosh, or even DOS based applications. These functions allow you a rich set of mechanisms that override these system colors, in order to show specific object information, such as selection states, specific object errors, or particular points of emphasis. Whenever you define user palettes, the instantiated object's Draw() function uses these color and font definitions in its drawing process. A brief description of the palette color and attribute overrides follows (A more complete discussion can be read in the ZafPaletteData section of this manual).

There are three main aspects of a user palette that determine the override functionality of an object.

(1) Color/font/style attributes. This is the actual palette information that will apply in the drawing operation if the conditions set in (2) and (3) are met. This part of the palette map is specified by "ZafPaletteStruct palette," which contains the following members:

Member	Description
lineStyle	This is the type of line (solid, dotted) that can be drawn.

Member	Description
fillPattern	This is the type of pattern (solid, interleaved) that can be used either in the clearing of background information, or in the drawing of textual information.
colorForeground/ colorBackground	These are the foreground/background color pairs used on color systems when drawing the object's visual information.
monoForeground/ monoBackground	These are the foreground/background monochrome color pairs used on black/white systems when drawing the object's visual information.
font	This is the logical font to be used when rendering text information to the display.

(2) Object Type. This part of the ZafPaletteMap structure, defined by “Zaf-PaletteType type,” specifies the type of drawing that you want to override. The following values are defined by ZAF:

Value	Definition
ZAF_PM_ANY_TYPE	This matches any palette request. If used, this should be the last matching entry (just before ZAF_PM_NONE) in the palette array since its value is the OR'ed composite of all other types.
ZAF_PM_NONE	This is an end-of-array indicator. This entry must be the last entry of an array. It is used by ZAF when counting the number of palette entries available in a given ZafPaletteData object. It should not be used as a sole member of a palette table.
ZAF_PM_OUTLINE	This specifies a palette entry that contains border colors. This is only used when the Bordered() flag is set, and the requesting object wants to draw an encompassing border around its specified region. This value is not used when a 3-dimensional border is drawn around the object.
ZAF_PM_BACKGROUND	Used when clearing or drawing to the background portion of the object. The background clears to the pattern specified in this palette using the foreground, background and pattern for the cleared area.

Value	Definition
ZAF_PM_FOREGROUND	Used when drawing graphic information, such as lines, rectangles, and ellipses within a window object. This palette is used after the background has been cleared, and additional information, such as a check-mark, or radio-button still need to be rendered on the display.
ZAF_PM_TEXT	This indicates a palette to be used when drawing textual information. Window objects typically use the font, pattern, foreground and background entries of the associated palette entry when rendering text information to the screen.
ZAF_PM_HOT_KEY	This indicates an entry to be used when drawing the “hotkey” portion of a string value. Normally, this value is only set and used on text based environments, since GUI environments generally show hot key information with an underline.
ZAF_PM_LIGHT_SHADOW	This is the light area of a 3-dimensional shadow. This palette is used in conjunction with ZAF_PM_DARK_SHADOW to present a shadowed appearance on the object. When the object appears raised, this entry is used on the left and top sides of the object. If the object appears depressed, this entry is used for the right and bottom sides of the object.
ZAF_PM_DARK_SHADOW	This is the dark area of a 3-dimensional shadow. This palette is used in conjunction with ZAF_PM_LIGHT_SHADOW to present a shadowed appearance on the object. When the object appears raised, this entry is used on the right and bottom sides of the object. If the object appears depressed, this entry is used for the left and top sides of the object.
ZAF_PM_FOCUS	This specifies the color value to be used when drawing the focus rectangle around a window object. Typically, this entry is used with ZAF_PM_ANY_STATE so the focus is drawn in a consistent color.

(3) Object state. This part of the ZafPaletteMap structure, defined by “Zaf-PaletteState state,” is used in conjunction with ZafPaletteType, and specifies

the state the object must be in, in order for the palette to be used in the current drawing operation. The following values are defined by ZAF:

Value	Definition
ZAF_PM_ANY_STATE	This value matches any logical palette request, as long as the specified type is ZAF_PM_ANY_TYPE or the value matches the requested palette type. This is typically used as the “final” matching entry in a palette array, so objects automatically match a default request.
ZAF_PM_ACTIVE	This state is used when the window, or window object, is shown as the front window. The colors in this entry are typically the same as ZAF_PM_INACTIVE, but may be different for objects that show different colors when shown as the front window (e.g. a window’s title bar is typically shown in an active color when the window has input focus, or inactive when it does not have the focus).
ZAF_PM_CURRENT	This matches when the requested object has the focus. This entry can be used if you want to distinguish items that have the input focus in more dramatic ways than just drawing a focus rectangle.
ZAF_PM_INACTIVE	This state is used when the parent does not have the input focus. As stated in ZAF_PM_ACTIVE, most objects do not use this setting. It is provided for special visual objects such as ZafTitle, that have a unique presentation to show the window either does or does not have the immediate input focus.
ZAF_PM_SELECTED	This state is used when the object’s Selected() attribute is “true.” Many objects, such as vertical and horizontal lists, show their “selected” children in a highlighted state, giving immediate visual queues to the user that the object has been selected from among the list of siblings.
ZAF_PM_DISABLED	This state is used when the object’s Disabled() attribute is “true.” Typically, this palette contains a “dithered” font and color setting that shows the object in a diminished, or unselectable state.
ZAF_PM_ENABLED	This state is used when the object’s Disabled() attribute is “false.” This palette typically contains the normal font and color setting associated with window objects.

There are three pre-defined `ZafWindowObject` functions that actually make changes to the user palette. These functions are `SetBackgroundColor()`, `SetTextColor()` and `SetFont()`. Evaluation of the `SetTextColor()` function provides a good tutorial on the proper use of user palettes.

```
ZafLogicalColor ZafWindowObject::SetTextColor(ZafLogicalColor
    color, ZafLogicalColor mono)
{
    // Make sure there is a userPalette.
    if (!userPaletteData)
        SetUserPaletteData(new ZafPaletteData());

    // Add the new entry.
    ZafPaletteStruct textPalette = userPaletteData->
        GetPalette(ZAF_PM_TEXT, ZAF_PM_ANY_STATE);
    textPalette.colorForeground = color;
    textPalette.monoForeground = mono;
    userPaletteData->AddPalette(ZAF_PM_TEXT, ZAF_PM_ANY_STATE,
        textPalette);

    // Return the current color.
    return (color);
}
```

The code above shows two main pieces. The first part shows the creation and specification of a new default `ZafPaletteData` object, if no user palette currently exists. The `ZafPaletteData` object provides a “container” for all color and font modifications. The second part adds a particular palette entry by either retrieving the palette entry if it already exists, or by creating a new entry if it does not exist (both operations accomplished within the `GetPalette()` function). The new palette is then modified by setting the color and monochrome text values, and finally, added as a new entry with the `AddPalette()` function.

Here is another code snippet that reinforces the palette creation techniques used in an application.

```
// Create a new button.
ZafButton *button = new ZafButton(0, 0, 20,
    ZAF_NULLP(ZafBitmapData), "button");

// Modify the selected background of the button to be yellow.
ZafPaletteData *palette = new ZafPaletteData();
ZafPaletteStruct entry = palette->GetPalette(ZAF_PM_BACKGROUND,
    ZAF_PM_SELECTED);
entry.colorBackground = ZAF_CLR_YELLOW;
palette->AddPalette(ZAF_PM_TEXT, ZAF_PM_SELECTED, entry);
```

```
button->SetUserPaletteData(palette);
```

Note, `SetUserPalette()` is an instance replacement of color and font information, not a class replacement. Thus, setting a new user palette will only affect the instantiated object where the information is specified, not all objects associated with the instance's class (e.g. a new button object vs. the `ZafButton` class). Also, be careful to examine the present contents of `UserPalette()` before replacing those contents. As mentioned above, the `SetBackgroundColor()`, `SetTextColor()` and `SetFont()` functions create a user palette before adding a particular font or color specification.

The return value for `UserPalette()` and `SetUserPalette()` is the current user palette data associated with the object. This will always be the color value passed into the `SetUserPalette()`.

```
bool Visible(void) const;
virtual bool SetVisible(bool visible);
```

A visible object is one that can be viewed by the user from within a window or on the screen. There are many occasions where changing an object's visibility may be appropriate. These may include, but are not restricted to:

- Overlaying multiple objects on the same window location, then making one object visible according to the current state of the application.
- Making the object invisible while performing extensive changes to the object. These changes may include modifying text information, removing presentation bitmaps, changing the selected state of the object, etc. Changing a single piece of information would not require temporarily making the object invisible. Multiple changes in succession, however, are greatly enhanced while the object is invisible because it dramatically reduces the amount of object refreshing.
- Modifying the object while preserving the object's position, relative to other siblings. You can always remove an object using `Subtract()`, but this causes the object's OS representation to be destroyed, and its position, relative to other siblings to be removed. Thus, `Subtract()` should only be used when extensive changes, that cannot be accomplished by toggling either `AutomaticUpdate()` or `Visible()`, are performed.

As stated above, using `SetVisible(false)` simply removes the object's visual presentation from the screen. It does not affect the object's current settings, its position within the parent window, the space the object occupies, or its created state with the underlying GUI environment.

Here is some sample code that shows the correct use of `SetVisible()`.

```
// Make extensive changes to the object.
```

```
object->SetVisible(false);
object->SetText("Continue with the application?");
object->Disabled(false);
object->SetViewOnly(true);
object->SetHzJustification(ZAF_HZ_CENTER);
object->SetVisible(true);

// Place two objects on the same screen location, setting one
// invisible.
ZafButton *button1 = new ZafButton(2, 2, 20,
    ZAF_NULLP(ZafBitmapData), "No Error");
window->Add(button1);

ZafButton *button2 = new ZafButton(2, 2, 20,
    ZAF_NULLP(ZafBitmapData), "Kill Application!");
button2->SetBackgroundColor(ZAF_CLR_RED);
button2->SetTextColor(ZAF_CLR_WHITE);
button2->SetVisible(false);
window->Add(button2);
```

Note, you can also use the `SetAutomaticUpdate()` function to prevent flashing of an object when color, font, or child additions and subtractions are being performed in your application. It is recommended you review `ZafWindowObject::SetAutomaticUpdate()` to understand the benefits and limitations of this related function.

Finally, note that setting the visible attribute on a parent object to "false" causes all of the children to become invisible, even though their individual visible states may be preset to be "true." The functionality of `SetVisible()` is thus propagated to children, grandchildren, etc., but only through inheritance, not by value replacement (i.e. the child object's visible member is not reset to be the same as the parent's value). Once the `Visible()` state is set back to true, the visual aspects of the children are restored.

The return value for `Visible()` and `SetVisible()` is the final, or current visible state of the object (true or false). Under normal circumstances, this will be the value passed into the `SetVisible()` function. It is only if a derived object overrides the functionality of `SetVisible()`, that the return state may be different from the value passed to `SetVisible()`.

```
static ZafWindowManager *windowManager;
```

This is a static pointer to the application's window manager. It is initialized when the `ZafWindowManager`'s constructor is called and should not be modified. All derived window objects use this member when making requests to

the window manager. They do not use the global variable `zafWindowManager`.

Here is a code snippet that shows how a derived window object can use the window manager to determine the current drag object.

```
ZafEventType MyDerivedWindowObject::DragDropEvent(const
    ZafEventStruct &event)
{
    if (event.type == S_DROP_DEFAULT && windowManager->dragObject)
        SetText(windowManager->dragObject->Text());
    ...
}
```

This member, as well as the static `ZafWindowObject::display` and `ZafWindowObject::eventManager` members are duplicate copies of the global variables `zafDisplay`, `zafEventManager`, and `zafWindowManager`. They are defined in the base `ZafWindowObject` class to allow advanced ZAF programmers the opportunity of removing the static definition, thus allowing particular instance variables to be associated with each window object; a feature useful in some multiple-display and embedded-system applications.

```
virtual void Write(ZafObjectPersistence &persist);
```

This function is used to persist a ZAF object. Typically, objects are persisted to a Zinc specified .DAT file using the ZAF Designer. But ZAF objects may also be persisted “in-code,” if they have been instantiated and have received an `S_INITIALIZE` message (This message causes the object to register string and number identifications, essential to object persistence). The following code shows how this type of persistence is performed.

```
// Create a window, then persist it to the specified file.
ZafWindow *window = new ZafWindow(0, 0, 50, 10);
window->AddGenericObjects(ZafStringData("Window"));
window->Add(new ZafPrompt(2, 2, 10, "name:"));
window->Add(new ZafString(12, 2, 30, ZAF_NULLP(ZafIChar), 100));
...

window->Event(S_INITIALIZE);

// Create the persist object.
ZafStorage *storage = new ZafStorage("myfile.dat");
ZafObjectPersistence persist(storage);
window->Write(persist);
delete storage;
```

As shown above, this function is generally used when storing a complete window, not just a window object. Nevertheless, the advanced definition of .DAT files allows the persistence of single window objects, if the associated file system allows directory traversal and independent object storage (both features available with the ZafStorage object). Here is a code snippet that shows how to persist a ZafButton object inside a parent window's directory.

```
// Persist a ZafButton to ~ZafWindow~MyWindow~MyButton.
ZafButton *button = new ZafButton(2, 2, 10,
    ZAF_NULLP(ZafBitmapData), "OK");
button->SetStringID("MyButton");
button->Event(S_INITIALIZE);

ZafStorage *storage = new ZafStorage("myfile.dat");
ZafObjectPersistence persist(storage);
storage->ChDir(~ZafWindow~MyWindow);
button->Write(persist);
```

For additional information on object persistence, see the ZafWindow::Write() section of this manual and the ZafWindowObject constructor section of this chapter.

```
ZafRegionStruct zafRegion;
```

See Region().

Appendices

Event Definitions

ZAF 5 defines a set of portable messages that may be trapped and allow developers to take action in response to user and system events. Many of these events are designed for programmer use, but some are intended for internal library use only and these are documented as such. However, all messages are fully documented to allow maximum flexibility.

Events are separated into six broad categories:

- Notification events
- System events
- Logical events
- Raw events
- Device requests
- Application events

Detailed descriptions of each category follow later in this chapter.

Subcategories exist within some of these broad categories as well. When present, these subcategories generally separate portable events (intended for use by ZAF programmers) from non-portable events (intended for internal library use only).

To fully release the capabilities of Zinc Application Framework a developer should be familiar with all these events. In particular, the following aspects of each event are critical:

- *Sendability* refers to an event's ability to be sent by the programmer to trigger an action. Some events may be sent, others are expected to be generated only by the Operating System or by ZAF internally.
- *Portability* refers to the identical behavior of the event on all Operating Systems. Almost all ZAF events are fully portable, but some have been defined for the use in specific operating systems as part of ZAF's internal portable library implementation.
- *Guarantee* refers to the reliability of a particular event in an application context. To maximize ease-of-use for developers, many events are guaranteed. That is, they are generated internally by ZAF on Operating Systems that would not normally generate an event.

Notification Events

Notification events are normally generated by the Operating System. They notify the library that an OS internal operation has either been initiated or completed. Notification events are not requests to take action, and in general should be used only to synchronize ZAF with the changing state of the OS. However, some notification events indicate OS operations that may be can-

celed or reversed; In these cases, the library may intervene to reset the OS state. (Notification events are only defined in response to an OS / ZAF state discrepancy on a platform, and not for internal ZAF notifications.)

- Notification events should not be sent by the user. For simplicity, ZAF assumes these events to be generated only by the OS.
- Notification events are fully portable. They will be received at the same times, in response to the same actions on all platforms.
- Notification events are guaranteed. Where an OS does not generate a notification event on a particular platform, the library will emulate the behavior and generate the event internally. (This is the only time that a notification event should be sent by ZAF.)

Following is the set of notification events. *Note: where shown, class names refer to the indicated classes and all their subclasses.*

N_CHANGE_PAGE	Notifies ZafNotebook that the current notebook page has changed.
N_CLOSE	Notifies ZafWindow that it is about to be closed. If ZafWindow returns 0, it will be closed; otherwise it will not be closed. (Operation may not be cancellable on Motif.)
N_CURRENT	Notifies ZafWindowObject that it has received focus. Calls memberUserFunction(), if any.
N_EXIT	Notifies ZafWindowManager that the application is about to exit. ZafWindowManager responds by returning the result of exitFunction(), if any. If exitFunction() returns 0, it will exit; otherwise it will not exit. (Operation may not be cancellable on Motif.)
N_HSCROLL	Notifies ZafWindowObject that its corresponding horizontal ZafScrollBar has scrolled, if the scroll bar is SupportObject(). ZafWindowObject responds by scrolling its data. If the scroll bar is not SupportObject(), then memberUserFunction() is called.
N_MOUSE_ENTER	Notifies ZafWindowObject that the mouse has entered its region. windowManager::mouseObject already points to the object. Sends an initialization message to ZafHelpTips, if the object has a help tip or a quick tip.
N_MOUSE_LEAVE	Notifies ZafWindowObject that the mouse has left its region. Sends a deinitialization message to ZafHelpTips, if the object has a help tip or a quick tip.
N_MOVE	Notifies ZafWindowObject that it has been moved, and that its zafRegion reflects the new position. ZafWindowObject responds by updating its internal information to correspond with the operating system object.

N_NON_CURRENT	Notifies ZafWindowObject that it is about to lose focus. If ZafWindowObject returns 0, it will lose focus; otherwise it will not lose focus. Calls memberUserFunction(), if any. (Operation may not be cancellable on Motif.)
N_SIZE	Notifies ZafWindowObject that it has been sized, and that its zafRegion reflects the new size. ZafWindowObject responds by updating its internal information to correspond with the operating system object.
N_TIMER	Notifies ZafWindowObject that a timer event has occurred.
N_VSCROLL	Notifies ZafWindowObject that its corresponding vertical ZafScrollBar has scrolled, if the scroll bar is SupportObject(). ZafWindowObject responds by scrolling its data. If the scroll bar is not SupportObject(), then memberUserFunction() is called.

System Events

System events are normally generated internally by ZAF. They are requests for action to be taken, and generally should not be ignored.

- System events may be sent by the user. However, some system events are used internally by ZAF and are not safe to be sent by the casual user. System events that are specifically anticipated to be sent by users will be documented fully. All others will be documented briefly for those trapping the events, or deriving custom ZAF objects.
- System events are fully portable. They cause the same actions on all platforms.
- System events are guaranteed. ZAF will generate system events internally at the same times, in the same orders on all platforms.

The following is the set of system events that are designed to be sent by the user. *Note: where shown, class names refer to those classes and all their sub-classes.*

S_ADD_OBJECT	Causes ZafWindow to add event.windowObject as a child.
S_CLOSE	Causes ZafWindowManager to close its top-most non-temporary window, and any temporary windows above it.
S_CLOSE_- TEMPORARY	Causes ZafWindowManager to close its top-most window, if it is temporary.
S_CONTINUE	User hook. For example, may cause a latent thread to continue processing.
S_COPY	Causes ZafString or ZafText to copy its selected data to the clipboard.
S_COPY_DATA	Causes ZafWindowObject to copy event.windowObject's data values and update visually.
S_CUT	Causes ZafString or ZafText to cut its selected data to the clipboard.

S_DECREMENT	Causes ZafSpinControl to decrement its field object by the value of ZafSpinControl::delta.
S_EXIT	Causes ZafWindowManager to receive an N_EXIT event.
S_HELP	Causes ZafWindowObject to display its help context by calling DisplayHelp(). Handled at the child-most level that has a help context.
S_HSCROLL	Causes a horizontally scrollable object to scroll itself horizontally event.scroll.delta units.
S_HSCROLL_CHECK	Causes a horizontally scrollable object to ensure its current child is visible. If it isn't, it is scrolled horizontally into view.
S_HSCROLL_- COMPUTE	Causes a horizontally scrollable object to compute its horizontal scrolling values, based on its current child and horizontal data. Its corresponding horizontal scroll bar's data is set accordingly.
S_HSCROLL_SET	Causes a horizontal ZafScrollBar to replace its data with event.scroll and update visually.
S_INCREMENT	Causes ZafSpinControl to increment its field object by the value of ZafSpinControl::delta.
S_MAXIMIZE	Causes ZafWindow to maximize itself on the display.
S_MDI_CASCADE_- WINDOWS	Causes a parent ZafMDIWindow to cascade all its MDI child windows beginning with the bottom-most window at the top-left corner and stepping each window down and right.
S_MDI_CLOSE	Causes a parent ZafMDIWindow to close the top-most MDI child window.
S_MDI_MAXIMIZE	Causes a child ZafMDIWindow to maximize itself within its parent.
S_MDI_MINIMIZE	Causes a child ZafMDIWindow to minimize itself within its parent.
S_MDI_MOVE_- MODE	Causes keyboard moving mode to begin on the current MDI child window, if supported by the native environment.
S_MDI_NEXT_- WINDOW	Causes a parent ZafMDIWindow to bring the next MDI child window to the top.
S_MDI_RESTORE	Causes a maximized or minimized child ZafMDIWindow to restore itself within its parent.
S_MDI_SIZE_MODE	Causes keyboard sizing mode to begin on the current MDI child window, if supported by the native environment.

S_MDI_TILE_ WINDOWS	Causes a parent ZafMDIWindow to tile all its MDI child windows so that all child windows are visible, taking up as much of the parent's client space as needed.
S_MINIMIZE	Causes ZafWindow to minimize itself on the display.
S_MOVE_MODE	Causes keyboard moving mode to begin on the current window, if supported by the native environment.
S_NEXT_WINDOW	Causes ZafWindowManager to bring the next window to the top.
S_PASTE	Causes ZafString or ZafText to paste the clipboard's data to it at the current cursor position, replacing any selected data.
S_REDISPLAY	Causes ZafWindowObject to redraw itself. Higher-priority events (such as mouse or keyboard events) may be processed first.
S_REDISPLAY_DATA	Causes ZafWindowObject to immediately redraw only its data portions.
S_REDISPLAY_ REGION	Causes ZafWindowObject to redraw the portion of itself requested in event.region (specified as a region relative to the top left of the object). Higher-priority events (such as mouse or keyboard events) may be processed first.
S_RESTORE	Causes a maximized or minimized ZafWindow to restore itself on the display.
S_SET_DATA	Causes ZafWindowObject to reset its data pointers to share event.windowObject's data and update visually. The data manager updates its notification lists to reflect the change.
S_SIZE	Causes ZafWindowObject to set its zafRegion to event.region and update visually. May be used to size and/or move the object, since zafRegion is modified.
S_SIZE_MODE	Causes keyboard sizing mode to begin on the current window, if supported by the native environment.
S_SUBTRACT_ OBJECT	Causes ZafWindow to subtract event.windowObject as a child.
S_VSCROLL	Causes a vertically scrollable object to scroll itself vertically event.scroll.delta units.
S_VSCROLL_CHECK	Causes a vertically scrollable object to ensure its current child is visible. If it isn't, it is scrolled vertically into view.
S_VSCROLL_ COMPUTE	Causes a vertically scrollable object to compute its vertical scrolling values, based on its current child and vertical data. Its corresponding vertical scroll bar's data is set accordingly.

S_VSCROLL_SET Causes a vertical ZafScrollBar to replace its data with event.scroll and update visually.

System Events (Internal)

The following is the set of system events that may be trapped, but are generally not sent by the user. These are generally “advanced” events used internally by ZAF, but they may be exploited by expert ZAF programmers. *Note: where shown, class names refer to those classes and all their subclasses.*

S_BEGIN_DRAG Causes windowManager::dragObject to point to the object being dragged. Sent to the window object under the mouse when a drag operation is initiated, usually by moving the mouse several pixels after a down-click. Dragging begins only if windowManager::dragObject is set by this event.

S_COMPUTE_SIZE Causes ZafWindowObject to recompute its region. Normally sent to an object when it is being created, or when its parent is being resized.

S_CREATE Causes ZafWindowObject and its children to be created by processing the following events or functions in this order: SetVisible(false), S_INITIALIZE, S_REGISTER_OBJECT, S_COMPUTE_SIZE, OSSize(), SetVisible(oldVisible). Normally sent to an object when it is added to its parent (or ZafWindowManager), if the parent has already been created.

S_CURRENT Causes ZafWindowObject to complete the process of gaining focus by updating itself visually to indicate that it has focus, setting the focus member, and synchronizing the ZAF object with the operating system.

S_DEINITIALIZE Causes ZafWindowObject to deinitialize itself and its children, by clearing screenID, clearing windowManager::mouseObject if appropriate, and doing anything else needed by the native environment. Normally sent to an object when it is being destroyed.

S_DESTROY Causes ZafWindowObject to destroy itself and all its children, if any. Normally sent to an object when it is subtracted from its parent. Causes S_DEINITIALIZE to be sent to itself and all its children, if any.

S_DRAG_COPY Received by ZafWindowObject being copy-dragged over.

S_DRAG_DEFAULT Received by ZafWindowObject being default-dragged over.

S_DRAG_LINK Received by ZafWindowObject being link-dragged over.

S_DRAG_MOVE Received by ZafWindowObject being move-dragged over.

S_DROP_COPY Received by ZafWindowObject that is copy-dropped on.

S_DROP_DEFAULT Received by ZafWindowObject that is default-dropped on.

S_DROP_LINK	Received by ZafWindowObject that is link-dropped on. ZAF does not currently implement link-drop, but this message is provided as a hook for the user.
S_DROP_MOVE	Received by ZafWindowObject that is move-dropped on.
S_END_DRAG	Causes the object being dragged to clear windowManager::dragObject.
S_HLP_CLOSE	Used internally by ZafHelpSystem to close the window when the Close button is selected.
S_HLP_SELECT_ TOPIC	Used internally by ZafHelpSystem to display event.windowObject's help context, sent when an item in the list is double-clicked.
S_HLP_SHOW_ INDEX	Used internally by ZafHelpSystem to display the help context index when the Show Index button is selected.
S_HLP_SHOW_ TOPIC	Used internally by ZafHelpSystem to display the help context corresponding to the name displayed in the string field when the Show Topic button is selected.
S_HLP_UPDATE_ NAME	Used internally by ZafHelpSystem to set the string field corresponding to the list item that is selected.
S_HOT_KEY	Causes ZafWindowObject to perform an action appropriate to the hot key, if its hotKeyChar is the same as event.key.value. When a hot key is typed, the current window sends S_HOT_KEY to each of its children until a match is found, or all the objects have been checked.
S_INITIALIZE	Causes ZafWindowObject to initialize itself and its children, including converting its zafRegion to be in native coordinates, initializing oldRegion, and initializing its numberID and stringID to unique values (if they weren't already initialized). Normally sent to an object when it is being created.
S_NON_CURRENT	Causes ZafWindowObject to complete the process of losing focus by updating itself visually to indicate that it has lost focus, clearing the focus member, and synchronizing the ZAF object with the operating system.
S_REDISPLAY_ DEFAULT	Causes ZafButton to redisplay its border to indicate that it has gained or lost default status.
S_REGISTER_ OBJECT	Causes ZafWindowObject to register itself and its children with the operating system, including calling RegisterObject(). Normally sent to an object when it is being created.

**System
Events (Result
Codes)**

The following is the set of system event values that functions may return as result codes. These system event values are not used in actual events, but may be returned to indicate a status code or a user's response. They are included

here since they are processed in much the same way as “normal” events. *Note: where shown, class names refer to those classes and all their subclasses.*

S_DLG_ABORT	Returned by ZafDialogWindow after it is closed by the Abort button.
S_DLG_CANCEL	Returned by ZafDialogWindow after it is closed by the Cancel button.
S_DLG_HELP	Returned by ZafDialogWindow after it is closed by the Help button (Motif).
S_DLG_IGNORE	Returned by ZafDialogWindow after it is closed by the Ignore button.
S_DLG_NO	Returned by ZafDialogWindow after it is closed by the No button.
S_DLG_OK	Returned by ZafDialogWindow after it is closed by the OK button.
S_DLG_RETRY	Returned by ZafDialogWindow after it is closed by the Retry button.
S_DLG_YES	Returned by ZafDialogWindow after it is closed by the Yes button.
S_ERROR	Returned by some functions to indicate error status.
S_NO_OBJECT	Returned by ZafWindowManager::Event(), indicating that there are no windows open, and the application should exit.
S_UNKNOWN	Returned from an Event() function when the event passed in was unknown to the class, and thus not handled.

Logical Events

Logical events are portable mappings of OS events that represent raw end-user input. These events include mouse and keyboard events, for example, but do not include events triggered by ZAF objects or OS objects. (For example, a button press or menu selection should not send a logical event.)

- Logical events should not be sent by the user or ZAF. ZAF assumes these events are portable mappings of raw OS input events, and therefore do not need to be handled by ZAF library objects. (Typically, ZAF library objects will act on raw OS input events to optimize performance). A portable subset of possible logical mappings is documented below.
- Logical events may be portable. Logical events are normally returned by LogicalEvent() in response to raw OS events (or they may be natively available on some platforms). A subset of possible logical mappings will be documented as portable. By implication, a set of OS events that can be mapped to the documented logical event set must pass through the ZAF objects on all platforms. Other logical events are not portable and will be disclaimed by the documentation.
- Logical events are not guaranteed. However, the documented subset of OS Events that map to logical events are guaranteed to be generated, and therefore the

logical mapping is guaranteed to be accessible. (In the future we may want to guarantee full user extensibility of the event map table.)

The following is the set of portable logical event mappings designed to be trapped by the user. Mappings on some systems are “soft” and may not correspond exactly to the keystrokes listed. *Note: where shown, class names refer to those classes and all their subclasses, and “list object” means ZafHzList, ZafVtList, or ZafTreeList.*

L_BACKSPACE	Mapped for ZafString or ZafText in response to a backspace event.
L_BEGIN_ESCAPE	Mapped for ZafWindowObject in response to a right mouse button down-click event.
L_BEGIN_SELECT	Mapped for ZafWindowObject in response to a left mouse button down-click event.
L_BOL	Mapped for ZafString or ZafText in response to a home event.
L_CANCEL	Mapped for ZafWindowObject in response to an escape event.
L_CLOSE	Mapped for ZafWindowManager when a keystroke (such as <ALT>-<F4>) indicates that the top-most non-temporary window should be closed.
L_CLOSE_- TEMPORARY	Mapped for ZafWindowManager when a keystroke (such as escape) indicates that the top-most temporary window should be closed.
L_CONTINUE_- ESCAPE	Mapped for the ZafWindowObject under the mouse when the mouse moves while the right button is still depressed.
L_CONTINUE_- SELECT	Mapped for the ZafWindowObject under the mouse when the mouse moves while the left button is still depressed.
L_COPY	Mapped for ZafString or ZafText when a keystroke indicates a copy event.
L_CUT	Mapped for ZafString or ZafText when a keystroke indicates a cut event.
L_DELETE	Mapped for ZafString or ZafText in response to a delete event.
L_DOUBLE_CLICK	Mapped for ZafWindowObject when it is clicked on twice in a row quickly with the left mouse button. The maximum time between the first up-click and the second down-click is determined by the native environment, or ZafWindowObject::doubleClickRate.
L_DOWN	Mapped for ZafText or a list object in response to a down arrow event.
L_END_ESCAPE	Mapped for the ZafWindowObject under the mouse when the right mouse button is released.

L_END_SELECT	Mapped for the ZafWindowObject under the mouse when the left mouse button is released.
L_EXIT	Mapped for ZafWindowManager when a keystroke indicates that the application should exit.
L_EOL	Mapped for ZafString or ZafText in response to an end event.
L_FIRST	Mapped for a list object in response to a home event.
L_HELP	Mapped for ZafWindowObject in response to a help event.
L_LAST	Mapped for a list object in response to an end event.
L_LEFT	Mapped for ZafString, ZafText, or a list object in response to a left arrow event.
L_MDI_NEXT_WINDOW	Mapped for a parent ZafMDIWindow when a keystroke indicates that the top-most MDI child window should become bottom-most, and the next MDI child window should become top-most.
L_NEXT	Mapped for ZafWindow in response to a tab event. (Tabbing order of children is defined by the environment).
L_PASTE	Mapped for ZafString or ZafText when a keystroke indicates a paste event.
L_PGDN	Mapped for ZafText, ZafScrolledWindow, or a list object in response to a page down event.
L_PGUP	Mapped for ZafText, ZafScrolledWindow, or a list object in response to a page up event.
L_PREVIOUS	Mapped for ZafWindow in response to a shift-tab event. (Tabbing order of children is defined by the environment).
L_RIGHT	Mapped for ZafString, ZafText, or a list object in response to a right arrow event.
L_SELECT	Mapped for a selectable object (e.g. ZafButton, ZafIcon, or any object in a list) in response to an enter or return event.
L_UP	Mapped for ZafText or a list object in response to an up arrow event.
L_VIEW	Mapped for ZafWindowObject when the mouse moves over it.

Logical Events (Non-portable)

The following is the set of non-portable logical events that are platform-specific and unsafe for portable ZAF applications. These logical events are required by ZAF internally for some native environments' non-portable needs.

Programmers may trap these events on the platforms indicated, but will not have access to them on others. *Note: where shown, class names refer to those classes and all their subclasses, and “list object” means ZafHzList, ZafVtList, or ZafTreeList.*

L_ALT_KEY	Causes focus to switch to or from the pull-down menu when <ALT> is pressed and released (DOS).
L_DELETE_EOL	Mapped for ZafString or ZafText when a keystroke indicates to delete the data from the beginning of the selected range to the end of the current line (DOS).
L_DELETE_WORD	Mapped for ZafString or ZafText when a keystroke indicates to delete the word that the cursor is positioned on (DOS).
L_EXTEND_FIRST	Mapped for an ExtendedSelection list object when a keystroke indicates to extend the selection through the first child (Motif).
L_EXTEND_LAST	Mapped for an ExtendedSelection list object when a keystroke indicates to extend the selection through the last child (Motif).
L_EXTEND_NEXT	Mapped for an ExtendedSelection list object when a keystroke indicates to extend the selection through the next unselected child (Motif).
L_EXTEND_PREVIOUS	Mapped for an ExtendedSelection list object when a keystroke indicates to extend the selection through the previous unselected child (Motif).
L_INSERT_TOGGLE	Mapped for ZafString or ZafText when the insert key is typed (DOS).
L_MARK_BOL	Mapped for ZafString or ZafText when a keystroke indicates to extend the selection to the beginning of the current line (DOS).
L_MARK_DOWN	Mapped for ZafText when a keystroke indicates to extend the selection one line down (DOS).
L_MARK_EOL	Mapped for ZafString or ZafText when a keystroke indicates to extend the selection to the end of the current line (DOS).
L_MARK_LEFT	Mapped for ZafString or ZafText when a keystroke indicates to extend the selection one character to the left (DOS).
L_MARK_PGDN	Mapped for ZafText when a keystroke indicates to extend the selection one page down (DOS).
L_MARK_PGUP	Mapped for ZafText when a keystroke indicates to extend the selection one page up (DOS).
L_MARK_RIGHT	Mapped for ZafString or ZafText when a keystroke indicates to extend the selection one character to the right (DOS).

L_MARK_UP	Mapped for ZafText when a keystroke indicates to extend the selection one line up (DOS).
L_MARK_WORD_-LEFT	Mapped for ZafString or ZafText when a keystroke indicates to extend the selection one word to the left (DOS).
L_MARK_WORD_-RIGHT	Mapped for ZafString or ZafText when a keystroke indicates to extend the selection one word to the right (DOS).
L_MAXIMIZE	Mapped for ZafWindow when a keystroke indicates to maximize itself on the display (DOS).
L_MDI_MOVE_MODE	Mapped for ZafMDIWindow to cause keyboard moving mode to begin on the current MDI child window (DOS).
L_MDI_SIZE_MODE	Mapped for ZafMDIWindow to cause keyboard sizing mode to begin on the current MDI child window (DOS).
L_MINIMIZE	Mapped for ZafWindow when a keystroke indicates to minimize itself on the display (DOS).
L_MOVE_MODE	Mapped for ZafWindow to cause keyboard moving mode to begin on the current window (DOS).
L_NEXT_WINDOW	Mapped for ZafWindowManager when a keystroke indicates to make the next window top-most (Motif, DOS).
L_NONE	Not an event mapping. Used to denote the last entry of an event map table.
L_RESTORE	Mapped for a maximized ZafWindow when a keystroke indicates to restore itself on the display (DOS).
L_SIZE_MODE	Mapped for ZafWindow to cause keyboard sizing mode to begin on the current window (DOS).
L_SYSTEM_MENU	Mapped for ZafWindow when a keystroke (such as <ALT>-<space>) indicates to open the system menu (DOS).
L_TOGGLE_-EXPANDED	Mapped for ZafTreeList when a keystroke indicates to expand or close the current tree item (Motif).
L_WORD_LEFT	Mapped for ZafString or ZafText when a keystroke indicates to move the cursor one word to the left (Motif, DOS, Macintosh).
L_WORD_RIGHT	Mapped for ZafString or ZafText when a keystroke indicates to move the cursor one word to the right (Motif, DOS, Macintosh).

**Raw Events
(Device
Events)**

Raw events are OS specific raw input events including mouse and keyboard.

- Raw events should not be sent by the user or ZAF. These events are always placed on the event queue by a ZAF device. Because these events vary widely between systems, they will not be documented, except in the abstract.
- Raw events are not portable. However, these events can often be mapped to portable, logical events, and/or have their data made portable by calling `LogicalEvent()`. Data in mouse and keyboard events will be converted by `LogicalEvent()` to object specific position, and portable character format, respectively. Despite documented nonportability, the raw event structure itself is consistent across platforms. `event.type` should always be `E_OSEVENT`, and `event.InputType()` should return `E_KEY`, `E_MOUSE`, etc.
- Raw events are not guaranteed. These events are generally handled by the OS and are not guaranteed to be received, with the exception of those raw events required to yield documented portable logical event mappings.

The following is the set of raw event types that may be trapped by the user, based on the return value of `event.InputType()`.

<code>E_KEY</code>	Indicates the <code>ZafKeyboard</code> device.
<code>E_MOUSE</code>	Indicates the <code>ZafMouse</code> device.
<code>E_OSEVENT</code>	Indicates a system event on any native environment, and is assigned to <code>event.type</code> . <code>event.InputType()</code> may return <code>E_KEY</code> or <code>E_MOUSE</code> , as appropriate. The value of this constant is also used for <code>E_MACINTOSH</code> , <code>E_MOTIF</code> , <code>E_MSWINDOWS</code> , <code>E_OS2</code> , and <code>E_XT</code> .

**Raw Events
(Event Types)**

The following is the set of raw event types that are designed to pass to `ZafEventManager` in specifying a device object, such as when calling the `DeviceState()` method.

<code>E_CURSOR</code>	Indicates the <code>ZafCursor</code> device.
<code>E_DEVICE</code>	Indicates all devices.
<code>E_HELPTIPS</code>	Indicates the <code>ZafHelpTips</code> device.
<code>E_TIMER</code>	Indicates the <code>ZafTimer</code> device.

**Device
Requests**

Device messages are messages sent to ZAF devices to request an action. They are similar in function to system events, but are sent only to devices. `D_` messages apply to all devices. `D?_` messages apply only to the device whose first character matches the `?`.

- Device messages can be sent by the user. Portable device request messages will be documented.
- Device messages may be portable. However, due to OS differences, all device messages are not implemented on all platforms. Portable messages will be fully documented; others briefly.
- Device messages are not guaranteed. However, when possible ZAF UI objects attempt to use these messages to achieve desired results, rather than working around them with the native OS.

The following is the set of portable device request events designed to be sent by the user. Note: where shown, class names refer to those classes and all their subclasses.

D_STATE	Causes a device to return its state.
DH_SET_HELP_ OBJECT	Causes ZafHelpTips::helpObject to be set to event.windowObject.
DH_UPDATE_ HELP_OBJECT	Causes ZafHelpTips::helpObject to update with the help tip of event.windowObject, if event.windowObject is not nil.
DM_BOTTOM_ LEFT_CORNER	Causes ZafMouse to show the bottom-left corner image.
DM_BOTTOM_ RIGHT_CORNER	Causes ZafMouse to show the bottom-right corner image.
DM_BOTTOM_SIDE	Causes ZafMouse to show the bottom side image.
DM_CANCEL	Causes ZafMouse to show the cancel image.
DM_CROSS_HAIRS	Causes ZafMouse to show the cross-hairs image.
DM_EDIT	Causes ZafMouse to show the I-bar image.
DM_LEFT_SIDE	Causes ZafMouse to show the left side image.
DM_MOVE	Causes ZafMouse to show the move image.
DM_RIGHT_SIDE	Causes ZafMouse to show the right side image.
DM_SELECT	Causes ZafMouse to show the selection image.
DM_TOP_LEFT_ CORNER	Causes ZafMouse to show the top-left corner image.
DM_TOP_RIGHT_ CORNER	Causes ZafMouse to show the top-right corner image.
DM_TOP_SIDE	Causes ZafMouse to show the top side image.

DM_VIEW Causes ZafMouse to show the default pointer image.

DM_WAIT Causes ZafMouse to show the wait image.

**Device
Requests
(Non-portable)**

The following is the set of non-portable device request events that are platform-specific and unsafe for portable ZAF applications. These device events are used internally for some native environments' non-portable needs. *Note: where shown, class names refer to those classes and all their subclasses.*

D_ACTIVATE Causes a device to be activated (DOS, Macintosh).

D_DEACTIVATE Causes a device to be deactivated (DOS, Macintosh).

D_DEINITIALIZE Causes a device to deinitialize (MSWindows, Motif, DOS, OS/2, Macintosh).

D_HIDE Causes ZafMouse to be hidden (MSWindows, DOS, OS/2, Macintosh).

D_INITIALIZE Causes a device to initialize (MSWindows, Motif, DOS, OS/2, Macintosh).

D_OFF Causes a device to stop putting events on the event queue (DOS).

D_ON Causes a device to put events on the event queue as normal (DOS).

D_POSITION Causes a device to change its position to event.position (DOS).

DC_INSERT Causes ZafCursor to show the insert image (DOS).

DC_OVERSTRIKE Causes ZafCursor to show the overstrike image (DOS).

DH_HELP_TIPS_
TIMER Notifies ZafHelpTips that its timer has expired, meaning that it should display the help tip associated with the object that the mouse is over (Motif, Macintosh).

DM_DRAG Causes ZafMouse to show the drag image (MSWindows, Motif, DOS, OS/2, Macintosh).

DM_DRAG_COPY Causes ZafMouse to show the copy-drag image (MSWindows, Motif, DOS, OS/2, Macintosh).

DM_DRAG_COPY_
MULTIPLE Causes ZafMouse to show the multiple-item-copy-drag image (MSWindows, Motif, DOS, OS/2, Macintosh).

DM_DRAG_LINK Causes ZafMouse to show the link-drag image (MSWindows, Motif, DOS, OS/2, Macintosh).

DM_DRAG_LINK_
MULTIPLE Causes ZafMouse to show the multiple-item-link-drag image (MSWindows, Motif, DOS, OS/2, Macintosh).

DM_DRAG_MOVE	Causes ZafMouse to show the move-drag image (MSWindows, Motif, DOS, OS/2, Macintosh).
DM_DRAG_MOVE_MULTIPLE	Causes ZafMouse to show the multiple-item-move-drag image (MSWindows, Motif, DOS, OS/2, Macintosh).

Application Events

Application events are system wide events intended to be received and handled only by the ZAF Window Manager. As such, these events are usually used to implement macro level functionality such as loading a window, changing application language, etc. Among other possible functionality, application events allow Zinc Designer's test mode to invoke more sophisticated application flows.

- Application events can be sent by the user. In fact, this is the source of most application events. All application events are fully documented.
- Application events are fully portable. In addition, the code required to implement an application event should, in most cases be portable ZAF code.
- Application events are not guaranteed. Because they are sent by the user, ZAF objects do not expect to receive these events at any specific time.

The following is the set of application events that may be sent by the user.

Note: where shown, class names refer to those classes and all their subclasses.

A_CHANGE_LANG_LOC	Causes the application to change its language and locale bases specified in event.text. (The object handling the event deletes event.text.)
A_CLOSE_WINDOW	Causes the application to close the window whose stringID matches event.text. (The object handling the event deletes event.text. If the window is Destroyable(), the object handling the event also deletes the window.)
A_COLOR_DIALOG	Causes ZafColorDialog to come up. (***)Not currently implemented.(***)
A_FILE_DIALOG	Causes ZafFileDialog to come up. (***)Not currently implemented.(***)
A_FONT_DIALOG	Causes ZafFontDialog to come up. (***)Not currently implemented.(***)
A_HELP_CONTEXT	Causes ZafHelpManager to display event.helpContext.
A_MINIMIZE_WINDOWS	Causes ZafWindowManager to minimize all the windows on the display.
A_OPEN_DOCUMENT	User hook for opening the document whose file name is in event.text. If event.text is nil, the application is starting up, and a new "untitled" document should be opened. Some operating systems may send this event. (The object handling the event deletes event.text.)

A_OPEN_WINDOW	Causes the application to open the persistent window whose storage pathname is in event.text. (The object handling the event deletes event.text.)
A_PRINT_DIALOG	Causes ZafPrintDialog to come up. (***)Not currently implemented.(***)
A_PRINT_- DOCUMENT	User hook for printing the document whose file name is in event.text. Some operating systems may send this event. (The object handling the event deletes event.text.)
A_RESTORE_- WINDOWS	Causes ZafWindowManager to restore all the minimized windows on the display.

Zinc Coding Standards

Zinc Software specifies standards for all code written for internal or external distribution. These standards improve the readability, organization and maintenance of source code and header files and are used when writing library code, example programs, tutorial programs, etc.

Naming Conventions

CLASSES AND STRUCTURES

Class names should be self-explanatory and should be in name-case format: first letter in uppercase lettering, all remaining characters in lowercase lettering, with no underscores used to separate words. Some example class and structure names are shown below.

```
struct ZafEventStruct
struct ZafPaletteStruct
class ZafElement
class ZafEventManager : public ZafList
class ZafButton : public ZafWindowObject
```

All class and structure names unique to Zinc Application Framework use the prefix “Zaf”. Names unique to Zinc DataConnect use the prefix “Zdc”. Other Zinc products may introduce unique three-character prefixes. Shared names (used by multiple products) currently use the prefix “Zaf”.

In addition, many Zinc classes and structures use suffixes:

```
Struct    // denotes a general structure
Data      // denotes a data object
```

FUNCTIONS

Function names should be self explanatory and should be in name-case format (see above). In addition, the function name should describe what the function does. Some example class and regular function names are shown below:

```
ZafElement *Previous(void) const;
ZAF_EVENT_TYPE Event(const ZafEvent &event);
static ZafWindowObject *Read(const ZAF_ICHAR *name, ZafIO *io,
    ZafIOObject *ioObject, ZafPersistence *persist);
```

VARIABLES

Variable names should be self-explanatory and use lowercase lettering for the first word, then name case for each word thereafter. Global variables should always be prefixed by the three character product prefix. Some example variable names are shown below.

```
extern ZafIO *ZafDefaultStorage;  
static int virtualCount = 0;  
int ZafBorder::width = 4;
```

Each variable should be declared on a separate line when it is needed by the function. When declaring a list of variables, the following order should be followed:

- External variables
- Static variables
- Variables with complex structures
- All other variables according to need within the application

In addition, only one space, and not tabs, should exist between the type and the variable. Comments should be aligned evenly after the variable list.

CONSTANTS

Constant variables should be self-explanatory and should be in all uppercase, with an underscore separating the words. Some example constant names are shown below:

```
const ZAF_ERROR ZAF_ERROR_NONE = 0;        // comment  
const ZAF_LOGICAL_EVENT L_VIEW = 1001;    // comment
```

In addition to the information described above:

- Constants should be placed before the definition of the class for which they apply, or at the beginning of the module.
- If several related constants are defined, the definitions should be grouped together with a preceding comment.
- Constant values should be tab-aligned to the right.
- Comments for each line, if any, should be aligned to the right of the value.

TYPEDEFS

Typedefs should use the same naming conventions as classes and structures. (However, as of this printing, ZAF typedefs are not yet using this convention.)

Organization

CLASS SCOPES

The class declaration in an include file should list public members first, protected members next, and private members last. Each major section should list static member variables first, member variables next, and member functions last, listed in alphabetical order. (Be sure to list the constructor and destructor

first.) In addition, each scope section should contain a short comment telling where its members are documented. The following example shows a class containing the three scope sections:

```
class ZafExportClass ZafTimeData : public ZafUtimeData
{
public:
    // -- General members ---
    ZafTimeData(void);
    ZafTimeData(int hour, int minute, int second, int
        milliSecond);
    ZafTimeData(const ZAF_ICHAR *string, const ZAF_ICHAR *format =
        ZAF_NULLP(ZAF_ICHAR));
    ZafTimeData(const ZafTimeData &copy);
    virtual ~ZafTimeData(void);

protected:
    // --- Persistence ---
    friend class ZafExportClass ZafPersistence;
    static ZafElement *Read(const ZAF_ICHAR *name, ZafIO *io,
        ZafIOObject *ioObject) { return (new ZafTimeData(name, io,
        ioObject)); }
};
```

FILES

Source code modules that contain class member functions should contain the copyright notice, then any include files, static member variables, and member functions, described in alphabetical order. An example of BORDER.CPP's file layout is shown below:

```
// Zinc Application Framework - Z_BORDER.CPP
// COPYRIGHT (C) 1990-1996. All Rights Reserved.
// Zinc Software Incorporated. Pleasant Grove, Utah USA

#include "z_border.hpp"

// ----- ZafBorder -----
ZAF_CLASSID ZafBorder::classID = ID_ZAF_BORDER;
ZAF_CLASSNAME_CHAR ZafBorder::className[] =
    ZAF_ITEXT("ZafBorder");
static ZAF_STRINGID_CHAR _stringID[] =
    ZAF_ITEXT("ZAF_NUMID_BORDER");

ZafBorder::ZafBorder(void) :
    ZafWindowObject(0, 0, 0, 0,
        ZafSupportObject, true,
        ZafRegionType, ZAF_OUTSIDE_REGION,
        ZafNoncurrent, true,
        0)
```

```
{  
}  
  
ZafBorder::~ZafBorder(void)  
{  
}
```

Comments

FILES

Each source file (.CPP or .HPP) should contain a three-line comment that contains the library or program name, the name of the file and copyright information. A sample header is shown below:

```
// Zinc Application Framework - BUTTON.CPP  
// COPYRIGHT (C) 1990-1996. All Rights Reserved.  
// Zinc Software Incorporated. Pleasant Grove, Utah USA
```

The copyright information should be copied as shown above. The copyright year should include the original year when the product was created and all subsequent years when major revisions were made.

Code not copyrighted by Zinc Software should generally be placed in separate files from Zinc code whenever possible. The same three-line comment should begin the file, followed by whatever copyright information is required by the owner of the source code.

FUNCTIONS

Each routine may be preceded by a short description giving the routine's purpose and any related algorithms. If the routine name intuitively describes the routine, no comment is needed. The example below shows the use of a function comment:

```
// This member function displays the biorhythm information in  
// the window. As the size of the window object changes (by  
// changing the parent window), the size of the biorhythm chart  
// also changes. A horizontal change results in a change in the  
// number of days displayed. A vertical change results in a  
// dynamic change in the height of the biorhythm curve.  
  
void Biorhythm::UpdateBiorhythm(void)  
{  
    ...  
}
```

VARIABLES

Function arguments and local variables should only have descriptive comments if their names are not descriptive. These comments should be lined up on a right tab region. In addition, all comments should start with a capital letter and be followed by a period. An example of two variable declarations is shown below.

```
ZAF_EVENT_TYPE ccode;    // The control code for an event.
int cardFile;            // File handle for the disk file.
```

BLOCKS

Block comments are used to describe a group of related code. Most block comments should be one line, if possible, and reside immediately above the block being commented. If more than a one line comment is needed, the extra lines should each begin with the double slash. Block comments should be indented to match the indentation of the line of code following it. A single blank line should precede the comment and the block of code should follow immediately after. Small blocks of code that do a specific job should be commented but not individual lines, unless the line is complex or not intuitive). Some example block comments are shown below.

```
// Destroy all of the items within the list.
Destroy();

// When the user selects a button, ccode
// is checked to see what type of event was received.
switch (ccode)
{
    ...
}
```

PRIVATE
COMMENTS

Infrequently, it may be helpful to add source code comments for the personal reference of the Zinc developer, but that should not be shipped with production code. In general, private comments are discouraged and should be used sparingly. When necessary, these comments should use the following format:

```
////? This algorithm needs more attention before shipping.
```

A source code utility will be run on the code prior to creating master diskettes. The utility will remove any lines that begin with “//???”. Previous standards that allowed “//***” as an alternate prefix to private comments are superseded, and these comments should be removed or replaced.

Indentation

CLASSES AND STRUCTURES

Structures and classes should have all members listed on individual lines and should be indented with one tab from the left margin. Several sample indentations are shown below:

```
class ZafExportClass ZafIcon : public ZafButton
{
public:
    // --- General members ---
    ZafIcon(int left, int top, ZafIconData *iconData, const
        ZAF_ICHAR *title, ZAF_ATTRIBUTE attribute = 0, ...);
    virtual ~ZafIcon(void);
    virtual ZAF_EVENT_TYPE Event(const ZafEventStruct &event);

protected:
    // --- General members ---
    ZAF_ICON_TYPE iconType;
    ZafIconData *iconData;

    ZafIcon(const ZafIcon &copy);
    virtual ZAF_EVENT_TYPE DragDropEvent(const ZafEventStruct
        &event);
    virtual ZafWindowObject *Duplicate(void) { return (new
        ZafIcon(*this)); }
    virtual ZAF_EVENT_TYPE DrawFocus(ZafRegionStruct &region,
        ZAF_EVENT_TYPE ccode);
    virtual ZAF_EVENT_TYPE DrawItem(const ZafEventStruct &event,
        ZAF_EVENT_TYPE ccode);
    virtual ZAF_EVENT_TYPE DrawShadow(ZafRegionStruct &region, int
        depth, bool fillRegion, ZAF_EVENT_TYPE ccode);
    virtual ZafPaletteStruct *MapClassPalette(ZAF_PALETTE_TYPE
        type, ZAF_PALETTE_STATE state);

private:
    static ZafPaletteMap defaultPaletteMap[];
};
```

FUNCTIONS

The main body of routines should have braces below the function declaration. All function code should be indented one tab. An example of this indentation is shown below:

```
void ZafButton::SetText(const ZAF_ICHAR *string)
{
    // Reset the button's string information.
    ...
}
```

FUNCTION CALLS

Parameters in a function call should be listed with each argument, followed by a comma and one space. If a routine call cannot fit on one line on the screen, it should be broken with the next half of the call indented one tab farther over. It should be split after a comma or logic symbol if possible. Several examples of this calling convention are shown below:

```
ZafWindow *ZafWindow::Generic(int left, int top, int width,
    int height,
    ZAF_ICHAR *title, ZafWindowObject *minObject,
    ZAF_ATTRIBUTE attribute, ...)
{
    // Create the window.
    ZafWindow *window = new ZafWindow(left, top, width, height);
}

STRING_WINDOW::STRING_WINDOW(int left, int top) :
    ZafWindow(left, top, 40, 9)
{
    // Set the window information.
    SetStringID("String Window");

    // Create the window fields.
    *this
        + new ZafBorder
        + new ZafMaximizeButton
        + new ZafMinimizeButton
        + new ZafSystemButton
        + new ZafTitle("String Window")
        ...
    ;
}
```

CASE STATEMENTS

The reserved word case should be aligned with the switch statement, but all code information should be indented an additional tab. Each additional level of logic should be indented one tab. The colon should immediately follow each case and the statement(s) should start on a new line. The break should also be on a separate line. An example of this organization is shown below:

```
EventType ZafPrompt::Event(const ZafEvent &event)
{
    // Switch on the event type.
    ZAF_EVENT_TYPE ccode = event.type;
    switch (ccode)
    {
        case S_CREATE:
```



```
case S_SIZE:
    ...
    break;

case S_CURRENT:
case S_NON_CURRENT:
    if (ZafWindowObject::NeedsUpdate(event, ccode))
        ZafWindowObject::Text(prompt, 0, ccode, lastPalette);
    break;
default:
    ccode = ZafWindowObject::Event(event);
    break;
}

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

SCOPING

Normally, scoping is done with an expression, and indents with level.

```
if (expression)
{
    statement1
    statement2
}
```

Simple scoping should not indent, however.

```
{
statement1
statement2
}
```

IF AND FOR STATEMENTS

Statements following an if or for should be indented one tab, and simple conditionals should use the inline ? operator. An example of these statements is shown below:

```
left = (left < 1) ? 1 : right;

if (event->type == E_KEY &&
    (event->rawCode == ESCAPE || event->rawCode ==
    BACKSPACE || event->rawCode == ENTER))
{
    offset = length;
    length = 0;
}
```

```
for (number = 0; number < noOfCalls; number++)  
    ; // Do nothing.
```

Index

A

AcceptDrop
 WindowObject 256

Add
 PopUpItem 148
 PullDownItem 139
 Window 230

AddColor
 Display 57

AddDepthItem
 TreeList 215

AddFont
 Display 58

AddGenericObjects
 Window 231

AddNotification
 Notification 119

AddStaticModule
 Application 8

AllowDefault
 Button 31

AllowToggling
 Button 32

Append
 StringData 183

Application 7

argc
 Application 8

argv
 Application 8

AutoClear
 String 173
 Text 193

AutomaticUpdate

Window 232
WindowObject 260

AutoRepeatSelection
 Button 32

AutoSize
 Button 33
 Image 103
 Prompt 134
 ScrollBar 166

AutoSortData
 HzList 98
 TreeItem 208
 TreeList 216
 VtList 225

B

Background
 Display 59

BackgroundColor
 WindowObject 261

BeginDraw
 Display 59
 WindowObject 263

Bignum 12

BignumData 15
 Bignum 14

Bitmap
 Display 60

BitmapData
 Button 33

Blocked
 EventManager 84

Border 25
 Window 232

Bordered

WindowObject 263

BroadcastEvent
 Window 233

Button 28

ButtonType
 Button 34

C

CellHeight
 HzList 99

CellWidth
 HzList 99

Center
 WindowManager 247

Changed
 Window 234
 WindowObject 264

Char
 StringData 183

ClassID
 Element 76
 WindowObject 265

classID
 WindowObject 265

ClassName
 Element 76
 WindowObject 265

className
 WindowObject 265

Clear
 BignumData 17
 Data 43
 IntegerData 111
 RealData 160

ClearImage

Button 34
 ClearNotifications
 Notification 120
 ClearText
 Button 34
 ClientRegion
 Window 234
 ClipRegion
 Display 60
 colorTable
 Display 61
 columns
 Display 61
 Compare
 StringData 184
 CompareAscending
 Window 235
 CompareDescending
 Window 235
 Control
 Application 9
 ConvertCoordinates
 Button 35
 WindowObject 265
 ConvertRegion
 WindowObject 265
 ConvertToDrawRe-
 gion
 WindowObject 266
 ConvertToOSBitmap
 Display 61
 ConvertToOSIcon
 Display 61
 ConvertToOSMouse
 Display 61

ConvertToOSPosition
 WindowObject 266
 ConvertToOSRegion
 WindowObject 267
 ConvertToZafBitmap
 Display 62
 ConvertToZafEvent
 WindowObject 267
 ConvertToZafIcon
 Display 62
 ConvertToZafMouse
 Display 62
 ConvertToZafPosition
 WindowObject 266
 ConvertToZafRegion
 WindowObject 267
 CopyDraggable
 WindowObject 276
 CornerScrollBar
 Window 235
 Count
 PopUpItem 149
 PullDownItem 140
 Current
 PopUpItem 149
 ProgressBar 128
 PullDownItem 140
 CursorOffset
 String 174
 Text 194
 CursorPosition
 Text 194

D

Data 41
 Date 48
 DateData
 Date 50
 Decrement
 ProgressBar 128
 DefaultButton
 Window 236
 DefaultUserFunction
 WindowObject 271
 DefaultValidateFunc-
 tion
 String 174
 Delta
 ProgressBar 128
 Depressed
 Button 35
 Depth
 Button 35
 DepthCurrent
 TreeItem 208
 TreeList 216
 DepthFirst
 TreeItem 208
 TreeList 216
 DepthLast
 TreeItem 208
 TreeList 216
 DepthNext
 TreeItem 209
 DepthPrevious
 TreeItem 209
 Destroy

- PopUpItem 149
- PullDownItem 140
- Window 236
- Destroyable
 - Data 44
 - Window 236
- DestroyColor
 - ZafDisplay 57
- DestroyEvent
 - EventManager 85
- DestroyFont
 - ZafDisplay 58
- DestroyOSBitmap
 - Display 62
- DestroyOSIcon
 - Display 62
- DestroyOSMouse
 - Display 62
- DestroyZafBitmap
 - Display 63
- DestroyZafIcon
 - Display 63
- DestroyZafMouse
 - Display 63
- Device 51
- DeviceImage
 - EventManager 85
- DevicePosition
 - EventManager 85
- DeviceState
 - Device 52
 - EventManager 85
- DeviceType
 - Device 53
- Disabled
 - WindowObject 273

- Display 55
- display
 - Device 52
 - WindowObject 274
- DisplayContext
 - Display 63
- DisplayMode
 - Display 63
- DockType
 - ToolBar 203
- double
 - BignumData 18
 - RealData 160, 161
- DragDropEvent
 - WindowObject 274
- Draggable
 - WindowObject 276
- dragObject
 - WindowManager 248
- Draw
 - WindowObject 277
- DrawBackground
 - WindowObject 277
- DrawBorder
 - WindowObject 277
- DrawContext
 - Display 63
- DrawFocus
 - WindowObject 278
- DrawLines
 - TreeList 217
- DrawShadow
 - WindowObject 278
- Duplicate
 - Data 44
 - WindowObject 280

- DynamicOSText
 - StringData 183
- DynamicOSWText
 - StringData 183
- DynamicText
 - StringData 183

E

- Element 75
- Ellipse
 - Display 64
- EndDraw
 - WindowObject 263
- Error
 - Data 44
 - WindowObject 281
- EvaluateIsA
 - Element 78
- Event
 - Bignum 14
 - Button 35
 - Date 50
 - EventManager 86
 - HelpTips 93
 - Integer 107
 - Mouse 116
 - PopUpItem 149
 - PullDownItem 140
 - Real 158
 - String 174
 - Text 194
 - Time 200
 - UTime 222
 - Window 236
 - WindowManager 248
 - WindowObject 282
- EventManager 84
- eventManager

WindowObject 288

ExitFunction
WindowManager 249

Expandable
TreeItem 209

Expanded
TreeItem 209

F

FillPattern
Display 65

First
PopUpItem 150
PullDownItem 140

Focus
WindowObject 288

FocusObject
PopUpItem 150
PullDownItem 141
Window 237
WindowObject 290

Font
Display 65
WindowObject 291

fontTable
Display 65

Foreground
Display 65

FormatData 88

FormattedText
BignumData 17
FormatData 89
IntegerData 112
RealData 161

G

GeometryManager
Window 237

Get
EventManager 86
PopUpItem 150
PullDownItem 141

GetObject
Data 46
PopUpItem 150
PullDownItem 141
WindowObject 292

H

HelpContext
WindowObject 293

HelpObject
HelpTips 93

helpObject
WindowManager 249

HelpObjectTip
WindowObject 293

HelpTips 91

HelpTipsType
HelpTips 94

HorizontalScrollBar
Window 238

HotKeyChar
Button 36
Prompt 135

HotKeyIndex
Button 36
Prompt 135

HzJustify
Button 37
Prompt 135
String 175
Text 195

HzList 96

I

Icon
Display 66

Image 101

ImageType
Mouse 116

Increment
ProgressBar 129

Index
PopUpItem 150
PullDownItem 141

InitialDelay
HelpTips 94
WindowObject 295

InitializeOSBitmap
Display 66

InitializeOSIcon
Display 66

InitializeOSMouse
Display 66

InputFormatData
String 175

InputFormatText
String 175

Insert
StringData 184

Installed

Device 53
Integer 105
IntegerData 110
 Integer 107
Invalid
 String 176
 Text 195
IsA
 Element 78
 WindowObject 296
ItemType
 PopUpItem 150
IValue
 BignumData 18

L

Last
 PopUpItem 151
 PullDownItem 141
Length
 StringData 185
Line
 Display 66
lines
 Display 61
LineStyle
 Display 67
lineTable
 Display 67
LinkDraggable
 WindowObject 276
LinkMain
 Application 9
ListIndex

Element 80
Locked
 Window 238
LogicalEvent
 WindowObject 297
LogicalPalette
 WindowObject 300
long
 BignumData 18
 IntegerData 112
LowerCase
 String 176

M

Main
 Application 10
MaximizeButton
 Window 238
Maximized
 Window 239
Maximum
 ProgressBar 129
MaxLength
 StringData 185
MemberUserFunction
 WindowObject 304
memberUserFunction
 WindowObject 304
menu
 PopUpItem 152
 PullDownItem 141
MinimizeButton
 Window 239
Minimized

Window 239
MinimizeIcon
 Window 240
Minimum
 ProgressBar 129
Mode
 Display 67
modeTable
 Display 67
MonoBackground
 Display 67
MonoForeground
 Display 68
monoTable
 Display 68
Mouse 115
 Display 68
mouseObject
 WindowManager 250
Moveable
 Window 240
MoveDraggable
 WindowObject 276
MoveEvent
 WindowObject 306

N

NewHelpTipDelay
 HelpTips 94
Next
 Device 53
 Element 80
 WindowObject 307
Noncurrent

WindowObject 308

NormalBitmap
TreeItem 209

Notification 118

NotifyCount
Notification 122

NotifyFocus
WindowObject 309

NotifySelection
WindowObject 312

NumberID
Element 81

O

operator -
BignumData 19
PopUpItem 152
PullDownItem 142
StringData 187
Window 241

operator --
BignumData 20
IntegerData 113
RealData 162

operator !=
BignumData 23
StringData 188

operator %
BignumData 21

operator %=
BignumData 22
IntegerData 114
RealData 162

operator *
BignumData 20

operator *=
BignumData 22
IntegerData 113
RealData 162

operator +
BignumData 20
PopUpItem 148
PullDownItem 139
StringData 188
Window 230

operator ++
BignumData 20
IntegerData 113
RealData 162

operator +=
BignumData 22
IntegerData 113
RealData 162
StringData 188

operator /
BignumData 21

operator /=
BignumData 22
IntegerData 113
RealData 162

operator <
BignumData 23
StringData 188

operator <=
BignumData 23
StringData 189

operator -=
BignumData 22
IntegerData 113
RealData 162
StringData 187

operator =
BignumData 21
IntegerData 113
RealData 162
StringData 189

operator ==
BignumData 23
StringData 189

operator >
BignumData 23
StringData 189

operator >=
BignumData 24
StringData 190

operator []
StringData 190

OSDraw
WindowObject 314

OSScreenID
WindowObject 325

OSUpdatePalettes
Button 37

OutputFormatData
String 177

OutputFormatText
String 177

P

Palette
Display 68

PaletteState
WindowObject 315

Parent
WindowObject 316

ParentDrawBorder
WindowObject 317

ParentDrawFocus
WindowObject 317

Password

String 177

PathID
Image 103

PathName
Image 103

patternTable
Display 69

Pixel
Display 69

Poll
Device 54
HelpTips 94
Mouse 117

PollDevices
EventManager 87

Polygon
Display 69

PopUpItem 146

PopUpMenu 153

postSpace
Display 70

preSpace
Display 70

Previous
Device 53
Element 80
WindowObject 318

ProgressBar 126

ProgressData
ProgressBar 129

ProgressStyle
ProgressBar 130

ProgressType
ProgressBar 130

Prompt 132

PullDownItem 137

PullDownMenu 143
Window 240

Put
EventManager 87

Q

QuickTip
WindowObject 319

R

RangeData
String 177

RangeText
String 177

ReadFromBeginning
EventManager 87

ReadFromEnd
EventManager 87

Real 156

RealData 159
Real 158

Rectangle
Display 70

RectangleXORDiff
Display 71

Redisplay
WindowObject 320

RedisplayData
WindowObject 320

Region
Button 37

WindowObject 320

RegionCopy
Display 72

RegionType
WindowObject 321

Remove
StringData 185

RepeatDelay
WindowObject 295

ResetOSBitmap
Display 72

ResetOSIcon
Display 72

ResetOSMouse
Display 72

RestoreDisplayContext
Display 63

RestoreDrawContext
Display 63

RootObject
WindowObject 325

RValue
BignumData 18

S

Scaled
Image 104

screenID
WindowObject 325

ScrollBar 164

ScrollData
ScrollBar 166

ScrollEvent

-
- WindowObject 327
 - ScrollType
 - ScrollBar 167
 - Selected
 - WindowObject 328
 - SelectedBitmap
 - TreeItem 209
 - SelectionType
 - HList 99
 - TreeList 217
 - VtList 226
 - Window 241
 - SelectOnDoubleClick
 - Button 38
 - SelectOnDownClick
 - Button 38
 - SendMessage
 - Button 38
 - SendMessageWhenSelected
 - Button 38
 - SetAcceptDrop
 - WindowObject 256
 - SetAllowDefault
 - Button 31
 - SetAllowToggling
 - Button 32
 - SetAutoClear
 - String 174
 - Text 193
 - SetAutomaticUpdate
 - Window 232
 - WindowObject 260
 - SetAutoRepeatSelection
 - Button 32
 - SetAutoSize
 - Button 33
 - Image 103
 - Prompt 134
 - ScrollBar 166
 - SetAutoSortData
 - HList 98
 - TreeItem 208
 - TreeList 216
 - VtList 225
 - SetBackground
 - Display 59
 - SetBackgroundColor
 - HList 99
 - TreeList 217
 - VtList 225
 - WindowObject 261
 - SetBignum
 - BignumData 19
 - SetBignumData
 - Bignum 14
 - SetBitmapData
 - Button 33
 - SetBordered
 - WindowObject 263
 - SetButtonType
 - Button 34
 - SetCellHeight
 - HList 99
 - SetCellWidth
 - HList 99
 - SetChanged
 - Window 234
 - WindowObject 264
 - SetChar
 - StringData 183
 - SetClipRegion
 - Display 60
 - SetCoordinateType
 - Button 35
 - SetCopyDraggable
 - WindowObject 276
 - SetCurrent
 - ProgressBar 128
 - SetCursorOffset
 - String 174
 - Text 194
 - SetCursorPosition
 - Text 194
 - SetDateData
 - Date 50
 - SetDefaultButton
 - Window 236
 - SetDelta
 - ProgressBar 128
 - SetDepressed
 - Button 35
 - SetDepth
 - Button 35
 - SetDestroyable
 - Data 44
 - Window 236
 - SetDeviceState
 - Device 52
 - SetDisabled
 - WindowObject 273
 - SetDisplayContext
 - Display 63
 - SetDockType
 - ToolBar 203
 - SetDrawContext
 - Display 63
 - SetDrawLines
 - TreeList 217

SetError	Mouse 116	SetMode
Data 44		Display 67
WindowObject 281	SetInitialDelay	
	HelpTips 94	SetMonoBackground
SetExitFunction	WindowObject 295	Display 67
WindowManager 249	SetInputFormat	
	String 175	SetMonoForeground
SetExpandable		Display 68
TreeItem 209	SetInputFormatData	
	String 175	SetMoveable
SetExpanded		Window 240
TreeItem 209	SetInteger	
	IntegerData 112	SetMoveDraggable
SetFillPattern		WindowObject 276
Display 65	SetIntegerData	
	Integer 107	SetNewHelpTipDelay
SetFocus		HelpTips 94
WindowObject 289	SetInvalid	
	String 176	SetNoncurrent
SetFont	Text 195	WindowObject 308
Display 65		
HzList 100	SetItemType	SetNormalBitmap
VtList 226	PopUpItem 150	TreeItem 209
WindowObject 291		
SetForeground	SetLineStyle	SetNumberID
Display 65	Display 67	Element 81
SetHelpContext	SetLinkDraggable	SetOSDraw
WindowObject 293	WindowObject 276	WindowObject 314
SetHelpObject	SetLocked	SetOSText
HelpTips 93	Window 238	StringData 186
SetHelpObjectTip	SetLowerCase	SetOSWText
WindowObject 294	String 176	StringData 186
SetHelpTipsType	SetMaximized	SetOutputFormat
HelpTips 94	Window 239	String 177
SetHotKey	SetMaximum	SetOutputFormatData
Button 36	ProgressBar 129	String 177
Prompt 135		
	SetMaxLength	SetPalette
SetHzJustify	StringData 185	Display 68
Button 37		
Prompt 135	SetMinimized	SetParent
String 175	Window 239	WindowObject 316
Text 195		
	SetMinimum	SetParentDrawBorder
SetImageType	ProgressBar 129	WindowObject 317
		SetParentDrawFocus

-
- WindowObject 317
 - SetParentPalette
 - WindowObject 317
 - SetPassword
 - String 177
 - SetPathID
 - Image 103
 - SetPathName
 - Image 103
 - SetProgressData
 - ProgressBar 129
 - SetProgressStyle
 - ProgressBar 130
 - SetProgressType
 - ProgressBar 130
 - SetQuickTip
 - WindowObject 319
 - SetRange
 - String 177
 - SetRangeData
 - String 177
 - SetReal
 - RealData 161
 - SetRealData
 - Real 158
 - SetRegion
 - Button 37
 - WindowObject 320
 - SetRegionType
 - WindowObject 321
 - SetRepeatDelay
 - WindowObject 295
 - SetScaled
 - Image 104
 - SetScrollData
 - ScrollBar 166
 - SetScrollType
 - ScrollBar 167
 - SetSelected
 - Button 38
 - String 178
 - WindowObject 328
 - SetSelectedBitmap
 - TreeItem 209
 - SetSelectionType
 - HzList 99
 - TreeItem 210
 - TreeList 217
 - VtList 226
 - Window 241
 - SetSelectOnDouble-Click
 - Button 38
 - SetSelectOnDown-Click
 - Button 38
 - SetSendMessage-WhenSelected
 - Button 38
 - SetSizeable
 - Window 241
 - SetStaticData
 - StringData 186
 - SetStep
 - ProgressBar 130
 - SetStringData
 - Button 39
 - Prompt 136
 - String 178
 - Text 195
 - TreeItem 210
 - SetStringID
 - Element 82
 - SetTemporary
 - Window 244
 - SetText
 - Button 40
 - String 178
 - StringData 186
 - Text 196
 - TreeItem 210
 - Window 244
 - WindowObject 329
 - SetTextColor
 - HzList 100
 - TreeList 218
 - VtList 226
 - WindowObject 331
 - SetTextStyle
 - ProgressBar 131
 - SetTiled
 - Image 104
 - SetTimeData
 - Time 200
 - SetTransparentBack-
 - ground
 - Prompt 136
 - SetUnanswered
 - String 179
 - Text 196
 - SetUpdate
 - Notification 122
 - SetUpperCase
 - String 179
 - SetUserFunction
 - WindowObject 332
 - SetUserPalette
 - HelpTips 95
 - SetUserPaletteData
 - WindowObject 335
 - SetUTimeData

- UTime 222
- SetValue
 - Button 40
- SetVariableName
 - String 179
- SetViewCurrent
 - TreeList 218
- SetViewOnly
 - String 180
 - Text 196
- SetVisible
 - Button 40
 - WindowObject 340
- SetVtJustify
 - Button 37
 - Prompt 136
- SetWallpaper
 - Image 104
- SetWidth
 - Border 27
- SetWordWrap
 - Text 197
- SetWrapChildren
 - ToolBar 203
- Sizeable
 - Window 241
- Sort
 - PopUpItem 152
 - PullDownItem 141
- StaticData
 - StringData 186
- StatusBar 168
- Step
 - ProgressBar 130
- String 171

- StringData 181
 - Button 39
 - Prompt 136
 - String 178
 - Text 195
 - TreeItem 210
- StringID
 - Element 82
- Subtract
 - PopUpItem 152
 - PullDownItem 142
 - Window 241
- SubtractNotification
 - Notification 120
- support
 - Window 242
- SupportCurrent
 - Window 242
- SupportDestroy
 - Window 242
- SupportFirst
 - Window 242
- SupportLast
 - Window 242
- SystemButton
 - Window 243
- SystemButtonMenu
 - Window 243

T

- Temporary
 - Window 244
- Text 191
 - Button 40
 - Display 73
 - String 178

- StringData 186
 - Text 196
 - TreeItem 210
 - Window 244
 - WindowObject 329
- TextColor
 - WindowObject 331
- TextSize
 - Display 73
- TextStyle
 - ProgressBar 130
- Tiled
 - Image 104
- Time 198
- TimeData
 - Time 200
- Title
 - Window 244
- ToggleExpanded
 - TreeItem 209
- ToggleSelected
 - WindowObject 328
- ToolBar 201
- TransparentBackground
 - Prompt 136
- TreeItem 205
- TreeList 213
 - TreeItem 211

U

- Unanswered
 - String 179
 - Text 196
- Update

Notification 122
 UpdateData
 Notification 124
 UpdateObjects
 Notification 124
 UpperCase
 String 179
 UserFunction
 WindowObject 332
 userFunction
 WindowObject 332
 userObject
 WindowObject 334
 UserPaletteData
 WindowObject 335
 userPaletteData
 WindowObject 335
 UTime 220
 UTimeData
 UTime 222

V

Value
 Button 40
 IntegerData 112
 RealData 161

VariableName
 String 179
 VerticalScrollBar
 Window 245
 ViewCount
 TreeList 218
 ViewCurrent
 TreeItem 211
 TreeList 218
 ViewFirst
 TreeItem 211
 TreeList 218
 ViewLast
 TreeItem 211
 TreeList 218
 ViewLevel
 TreeItem 212
 ViewNext
 TreeItem 212
 ViewOnly
 String 180
 Text 196

ViewPrevious
 TreeItem 212

Visible
 WindowObject 340

VtJustify
 Button 37
 Prompt 136

VtList 223

W

Wallpaper
 Image 104

Width
 Border 27

Window 227

WindowManager 246

windowManager
 WindowObject 341

WindowObject 251

WordWrap
 Text 197

WrapChildren
 ToolBar 203

Write
 WindowObject 342

Z

ZafIChar
 StringData 187

zafRegion
 WindowObject 320