



Programmer's Reference

**Zinc® Application Framework™
Version 5**

Zinc Software Incorporated
Pleasant Grove, Utah

NOTICE

This documentation is available in electronic and printed formats. If the electronic documentation is printable, a single copy may be printed for use by the Developer. Except for the foregoing, no part of this publication may be reproduced, translated, stored in a retrieval system, or transmitted, in any form or by any means, without the prior written permission of Zinc Software Incorporated (“Zinc”).

DISCLAIMER

While every precaution has been taken in the preparation of this manual, Zinc assumes no responsibility for errors or omissions. This publication and features described herein are subject to change without notice. ZINC MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE.

TRADEMARKS

Zinc is a registered trademark and Zinc Application Framework, Zinc Designer and Zinc DataConnect are trademarks of Zinc Software Incorporated. All other trademarks and tradenames used herein are owned by their respective holders.

LICENSE AGREEMENTS

Zinc Application Framework is licensed subject to the terms and conditions of one of two separate license agreements found in the “Getting Started” manual. The Personal Version license is provided to individuals developing non-commercial, non-distributable, personal-use-only applications. There is no license fee or royalty required for the Personal Version license. HOWEVER, TO EXERCISE RIGHTS BEYOND THE PERSONAL VERSION LICENSE, THE DEVELOPER MUST PURCHASE A PROFESSIONAL VERSION LICENSE FROM ZINC.

ACKNOWLEDGEMENTS

The ChartFolio framework used by ZafChart is licensed software ©1994-97 DPC Technology Corporation. The XPM library used by ZafImage on Motif is licensed software ©1989-95 GROUPE BULL. The ZAF installation program on Windows (INSTALIT) is licensed software ©1986-96 HPI. The MetaWINDOW graphics primitives used by ZafDisplay on DOS is licensed software ©1988-96 Metagraphics, Inc.

This manual was generated December 23, 1997.

Contacting Zinc

Worldwide

Sales: info@zinc.com, sales@zinc.com
Technical Support: support@zinc.com
Training and Consulting: services@zinc.com
Web: <http://www.zinc.com/>
Ftp: <ftp://ftp.zinc.com/>
CompuServe: GO ZINC

North America

Zinc Software Incorporated
405 South 100 East
Pleasant Grove, Utah 84062 USA
Tel: 1-801-785-8900
Sales: 1-800-638-8665
Support: 1-801-785-8998
Fax: 1-801-785-8996

Zinc Software Services, Inc.
42627 Garfield, Suite 214
Clinton Township, Michigan 48038 USA
Tel: 1-810-228-4900
Fax: 1-810-228-6633

Europe

Zinc Software (UK) Ltd.
106-108 Powis Street
London, SE18 6LU United Kingdom
Tel: +44 (0)181 855-9918
Fax: +44 (0)181 316-2211
BBS: +44 (0)181 317-2310
Email: europe@zinc.com

Table of Contents

Introduction.	11
----------------------	-----------

Class Reference	17
------------------------	-----------

Class Hierarchy	19
ZafApplication	22
ZafAttachment	27
ZafBignum	32
ZafBignumData	37
ZafBitmap	47
ZafBitmapData	50
ZafBitmapStruct	53
ZafBorder	55
ZafButton	58
ZafChart	72
ZafChartStub	81
ZafCodeSetData	82
ZafComboBox	89
ZafConstraint	94
ZafCoordinateType	97
ZafCursor	99
ZafData	102
ZafDataManager	108
ZafDataPersistence	110
ZafDataRecord	119
ZafDate	121
ZafDateData	126
ZafDevice	129
ZafDialogWindow	133
ZafDimensionConstraint	136
ZafDiskFile	140
ZafDiskFileSystem	142

ZafDisplay	147
ZafElement	170
ZafEraStruct.	180
ZafErrorStub	182
ZafErrorSystem	184
ZafEventManager	186
ZafEventMap	191
ZafEventStruct.	193
ZafFile	200
ZafFileDialog	207
ZafFileInfoStruct.	211
ZafFileSystem	213
ZafFormatData.	218
ZafFormattedString	221
ZafGdiDisplay	227
ZafGeometryManager	228
ZafGroup	232
ZafHelpStub.	237
ZafHelpSystem	238
ZafHelpTips.	242
ZafHzList.	247
ZafI18nData.	252
ZafIcon.	257
ZafIconData	262
ZafIconStruct	265
ZafImage	267
ZafImageData	272
ZafImageStruct	274
ZafInteger.	275
ZafIntegerData.	280
ZafKeyboard	285
ZafKeyStruct	288
ZafLanguageData.	289
ZafLanguageManager	293
ZafList	296
ZafListBlock	300
ZafLocaleData	301
ZafLocaleStruct	304

ZafMaximizeButton	310
ZafMDIWindow	313
ZafMessageData	317
ZafMessageStruct	319
ZafMessageWindow	320
ZafMinimizeButton	326
ZafMouse	329
ZafMouseData	333
ZafMouseStruct	337
ZafMSWindowsApp	340
ZafNotebook	346
ZafNotification	350
ZafObjectPersistence	358
ZafPaletteData	367
ZafPaletteMap.	370
ZafPaletteStruct	371
ZafPath	375
ZafPathElement	376
ZafPopUpItem	378
ZafPopUpMenu	385
ZafPositionStruct	388
ZafPrintDialog	390
ZafPrinter	392
ZafPrintJobStruct	399
ZafProgressBar	400
ZafPrompt	406
ZafPullDownItem	411
ZafPullDownMenu	417
ZafQueueBlock	420
ZafQueueElement	421
ZafReal	422
ZafRealData	426
ZafRegionStruct	430
ZafRelativeConstraint.	434
ZafScreenDisplay	439
ZafScrollBar	442
ZafScrollData	446
ZafScrollStruct	450

ZafScrolledWindow	452
ZafSpinControl	456
ZafSplitter	460
ZafStandardArg	465
ZafStatusBar	468
ZafStorage	471
ZafStorageFile	476
ZafString	478
ZafStringData	490
ZafSystemButton	501
ZafTable	507
ZafTableHeader	518
ZafTableRecord	522
ZafText.	525
ZafTime	532
ZafTimeData	536
ZafTimer	539
ZafTitle	543
ZafToolBar	546
ZafTreeItem	550
ZafTreeList	558
ZafUTime.	565
ZafUTimeData.	570
ZafVtList	579
ZafWindow	583
ZafWindowManager	603
ZafWindowObject	609

Function Reference 703

ZafAbs	705
ZafCrNIToCr	706
ZafCrNIToNI	707
ZafCrToCrNI	708
DynamicPtrCast	709
ZafMax.	710
ZafMin	711
ZafNIToCrNI	712
ZafRegisterMouse	714

ZafStrColl	715
ZafStrdup	716
streq.	717
ZafStricmp	718
ZafStrlwr.	719
strneq	720
strnicmp	721
strrepc	722
strstrip	723
ZafStrupr.	724
ZafStrXFrm.	725
WildStrcmp.	726
 Utility Reference727
Convert	729
Rep	730
Rep4to5	731
ZMake.	746
 Appendices753
Event Definitions	A-755
Property Matrices	B-773
ZAF 5 to 4 Class Comparisons	C-791
Character Maps	D-796
ISO Country Codes.	E-798
ISO Language Codes	F-806
X Resources	G-811
Zinc Coding Standards	H-813
 Index.823

Introduction

Zinc Application Framework Version 5 is a sophisticated product with many nuances. As you will soon discover, there are as many ways to accomplish a given task using ZAF as there are programmers. Zinc has created this reference with this principle in mind. Not only does this reference manual define and publish the ZAF API, it also offers examples and hints on its use.

The Programmer's Reference is available in two formats: hard copy documentation and electronic documentation using Adobe® Acrobat®. Electronic documentation is supplied with Zinc Application Framework. Hard copy documentation is available as a separate purchase.

Many programmers are already familiar with the advantages of electronic documentation and have become accustomed to using it. If you are not one of these people, you've probably had a bad experience with first-generation electronic documentation. We encourage you to try again! The ZAF 5 Programmer's Reference will be the best on-line documentation you've ever experienced—and may change your mind forever about electronic docs.

This introduction chapter will familiarize you with the organization and formatting conventions of the Programmer's Reference, and provide useful tips for effective use of the electronic documentation. Contact information for Zinc software is provided at the end of the chapter.

Organization

This manual is divided into five major sections as follows:

- **Table of Contents**

The table of contents is hypertext linked in the electronic documentation. This allows rapid navigation to each chapter.

- **Class Reference**

All ZAF 5 classes and structures are documented in this section—the majority of the manual. Each class is formatted similarly to provide complete yet comfortable information about each class. (Refer to the “[Conventions and Formats](#)” section below for more information.) For programmers using electronic documentation, these chapters are internally hypertext linked, plus linked to related and base classes for rapid reference.

- **Utility Reference**

External applications supplied with ZAF 5 are documented in this section, with one notable exception—Zinc Designer is documented separately in its own manual. If you are converting applications developed using a previous version of Zinc Application Framework to ZAF 5, this section contains useful information including a complete correlation of ZAF 4 symbols to their new ZAF 5 equivalents.

- **Appendices**

Appendices include all essential reference information for ZAF 5 that cannot be comfortably categorized as “Class Reference” or “Utility Reference.” Information found here includes a comprehensive reference to messages, Zinc internal coding conventions, and useful tables of object properties including defaults.

- **Index**

The Programmer’s Reference index includes every public member function in the ZAF 5 API, plus many additional references to information found only in this manual. This section is fully hypertext linked for programmers using electronic documentation.

Conventions and Formats

The majority of this manual is dedicated to the documentation of classes and structures. The following format is generally followed in each chapter:

ClassName

Member 1	Member 2	Operator 1
----------	----------	------------

A table at the beginning of each chapter lists all members documented in the chapter. Only those members that meet *all* of the following criteria are documented:

- The member is public
- The member is overloaded in this class (not simply inherited from a base class)
- The member is substantially changed from the base class implementation, if any
- The member is not a “Blocking function.” Blocking functions are those members overloaded simply to prevent modification of a class property in a derived class.

These are not normally documented. Blocking functions can be identified by entries in the “Member Initializations” table of the chapter’s “Constructors” section.

Inheritance

```
ZafDialogWindow : ZafWindow : ((ZafWindowObject :
    ZafElement), ZafList)
```

The inheritance list shows the complete class hierarchy for this class. It uses a terse notation (shown above) where a “:” (colon) separates a derived class from its base class, and a “,” is used to denote multiple inheritance. “()” may also be used for simple grouping.

The class hierarchy depicted above can be represented hierarchically as:

```
ZafDialogWindow
    ZafWindow
        ZafWindowObject
            ZafElement
            ZafList
```

In this hierarchy, “ZafDialogWindow” multiply inherits from “ZafWindow”. “ZafWindow” in turn inherits from “ZafWindowObject” (which inherits from “ZafElement”) and “ZafList”.

Base classes are hypertext linked to their reference chapters for rapid access by programmers using electronic documentation.

Declaration

```
#include <z_dlgwin.hpp>
```

The declaration directs the programmer to the ZAF header file (found in the “\include” directory) that contains the definition for the class. The programmer does not typically need to specifically include this header file since it is automatically included with “#include <zaf.hpp>.”

Description

This section describes the purpose of the class and its typical uses. Often the description will also include important information about class interactions, caveats, and specific application usage.

Constructor(s) The “constructors” section begins with a brief description of the purpose for the classes constructors and is followed by a table of “Member Initializations” as follows:

Member Initializations

ZafWindow

Destroyable()	false
...	

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	false [†]
...	

The member initializations table lists the referenced class and any relevant base classes along with their individual members that are specifically set in the constructors for this class. To emphasize proper API usage, the public accessor functions used to manipulate the protected and private data members are listed instead of the data members themselves.

When present, “†” indicates a “blocking function” that prevents changes to the attribute in this class. The value is set in the constructor and should not be altered.

Destructor Specific information important to the deallocation of this class is presented in this section. ZAF classes chain their destructor to the destructors of base classes.

Members
SetCurrentPage

virtual int **SetCurrentPage**(ZafWindowObject *currentPage)

Each member variable and function is documented with its complete declaration from the header file, and a full description of the purpose and usage of the member. Parameters are detailed and sample code snippets are often provided.

Tips for Electronic Docs

To maximize the utility of electronic documentation programmers may wish to consider the following tips:

- Obtain the latest version of Adobe Acrobat with full text search capabilities. Zinc's documentation relies on the capabilities of Adobe Acrobat version 3.0 with full text search. The Zinc ftp site (<ftp://ftp.zinc.com>) and web site (<http://www.zinc.com>) maintain copies of Acrobat, but Adobe's site (<http://www.adobe.com>) is the authoritative source.
- Maximize the Acrobat window. You're looking at a document formatted for printing. Pages are larger than the display and fonts are relatively small. The bigger the viewing area the better!
- For maximum performance, select "File | Preferences | General" and deselect the option to "Smooth text and monochrome images." This will greatly enhance the speed of scrolling and page redisplay.
- Set continuous page display mode by selection "View | Continuous" from the menu. This will allow you to more easily read sections that are split across two pages.
- If you know which class you are looking for, use the Acrobat's "bookmarks" (the left side of the display) to expand the "Class Reference" section and directly select the class you want.
- Once you're positioned on the correct class, click once in the document to automatically "zoom" the display to full page width. This is the best size for actually reading the content!
- Scan the members table at the top of the class. If you find the member you want, click it and you'll be taken directly to the reference for that member.
- If you don't find the member you want, chances are good that the member was inherited from a base class. Click on the first base class in the class' "Inheritance" diagram to jump directly to the base class. Repeat the process of scanning for the member you want.
- To jump back to a previous view (i.e. "undo" a hypertext jump) use the "<<" button on the Acrobat toolbar.
- If you don't know which class you're looking for, select the member in the "Index" portion of Acrobat's "bookmarks" (the left side of the display). You'll jump into the index where you can select which of the references to use for that member. Click on the page number of the reference to jump directly to the member.
- Copy and paste code snippets into your own projects using the native clipboard features of your OS. Note, however, that Acrobat is not a textual application and does not preserve indenting when copying to the clipboard.
- Use the "full text indexing" capability of Acrobat to search for words that cannot be found any other way. Select "Tools | Search" from the menu. This is a rapid, full-text search. If your version of Acrobat does not support full-text searching,

select “Tools | Find” from the menu for a slower, sequential search of the documentation.

**Contacting
Zinc**

We want to hear from you! Your feedback drives the progress of the products and services at Zinc. Watch our web site regularly for the latest information from Zinc, and participate in Zinc’s discussion groups.

To contact Zinc directly, please refer to the information at the beginning of this manual.

Class Reference

Class Hierarchy

The class hierarchy for ZAF 5 is presented below. “<-” indicates multiple inheritance and specifies additional base classes.

```

ZafApplication                                (z_app.hpp)
ZafDataPersistence                            (z_data.hpp)
    ZafObjectPersistence                      (z_win.hpp)
ZafDisplay                                    (z_dsp.hpp)
    ZafScreenDisplay                         (z_scrdsp.hpp)
    ZafPrinter                              (z_print.hpp)
ZafElement                                    (z_list.hpp)
    ZafConstraint                           (z_gmgr.hpp)
        ZafAttachment                      (z_gmgr.hpp)
        ZafDimensionConstraint             (z_gmgr.hpp)
        ZafRelativeConstraint              (z_gmgr.hpp)
    ZafData <- ZafNotification                (z_data.hpp)
        ZafDataRecord <- ZafList            (z_data.hpp)
            ZafDataManager                  (z_data.hpp)
            ZafLanguageData                 (z_lang.hpp)
            ZafLanguageManager              (z_lang.hpp)
        ZafFormatData                      (z_fdata.hpp)
            ZafBignumData                   (z_bnum.hpp)
            ZafIntegerData                  (z_int.hpp)
            ZafRealData                     (z_real.hpp)
            ZafStringData                   (z_str.hpp)
                ZafMessageData              (z_lang.hpp)
            ZafUTimeData                    (z_ftime.hpp)
                ZafDateData                 (z_date.hpp)
                ZafTimeData                 (z_time.hpp)
        ZafI18nData                        (z_idata.hpp)
            ZafCodeSetData                  (z_cset.hpp)
            ZafLocaleData <- ZafLocaleStruct (z_loc.hpp)
        ZafImageData                       (z_idata1.hpp)
            ZafBitmapData <- ZafBitmapStruct (z_bmap.hpp)
            ZafIconData <- ZafIconStruct     (z_icon.hpp)
            ZafMouseData <- ZafMouseStruct   (z_mouse1.hpp)
        ZafPaletteData                     (z_pall1.hpp)
        ZafScrollData <- ZafScrollStruct     (z_scrll1.hpp)
ZafDevice                                    (z_dev.hpp)
    ZafCursor                              (z_cursor.hpp)
    ZafHelpTips                            (z_h tips.hpp)
    ZafKeyboard                             (z_keybrd.hpp)
    ZafMouse                                (z_mouse2.hpp)
    ZafTimer                                (z_timer.hpp)
ZafPathElement                              (z_file.hpp)

```

ZafQueueElement	(z_evtmgr.hpp)
ZafWindowObject	(z_win.hpp)
ZafBitmap	(z_bmap1.hpp)
ZafBorder	(z_border.hpp)
ZafButton	(z_button.hpp)
ZafIcon	(z_icon1.hpp)
ZafMaximizeButton	(z_max.hpp)
ZafMinimizeButton	(z_min.hpp)
ZafPopUpItem	(z_popup.hpp)
ZafPullDownItem	(z_plldn.hpp)
ZafSystemButton	(z_sys.hpp)
ZafTitle	(z_title.hpp)
ZafChartStub	(z_chart.hpp)
ZafChart	(cf_api/bz_zcht.hpp)
ZafGeometryManager <- ZafList	(z_gmgr.hpp)
ZafImage	(z_image.hpp)
ZafProgressBar	(z_pgrss.hpp)
ZafPrompt	(z_prompt.hpp)
ZafSplitter	(z_split.hpp)
ZafString	(z_str1.hpp)
ZafBignum	(z_bnum1.hpp)
ZafDate	(z_date1.hpp)
ZafFormattedString	(z_fmtstr.hpp)
ZafInteger	(z_int1.hpp)
ZafReal	(z_real1.hpp)
ZafTime	(z_time1.hpp)
ZafUTime	(z_utime1.hpp)
ZafWindow <- ZafList	(z_win.hpp)
ZafComboBox	(z_combo.hpp)
ZafDialogWindow	(z_dlgwin.hpp)
ZafFileDialog	(z_fildlg.hpp)
ZafMessageWindow	(z_msgwin.hpp)
ZafGroup	(z_group.hpp)
ZafHelpSystem <- ZafHelpStub	(z_help.hpp)
ZafHzList	(z_hlist.hpp)
ZafMDIWindow	(z_mdwin.hpp)
ZafNotebook	(z_notebk.hpp)
ZafPopUpMenu	(z_popup.hpp)
ZafPullDownMenu	(z_plldn.hpp)
ZafScrollBar	(z_scrll2.hpp)
ZafScrolledWindow	(z_sclwin.hpp)
ZafSpinControl	(z_spin.hpp)
ZafStatusBar	(z_status.hpp)
ZafTable	(z_table.hpp)
ZafTableHeader	(z_table.hpp)
ZafTableRecord	(z_table.hpp)
ZafText	(z_text.hpp)
ZafToolBar	(z_tbar.hpp)
ZafTreeItem	(z_tree.hpp)

ZafTreeList	(z_tree.hpp)
ZafVtList	(z_vlist.hpp)
ZafWindowManager	(z_win.hpp)
ZafEraStruct	(z_loc.hpp)
ZafErrorStub	(z_error.hpp)
ZafErrorSystem	(z_error.hpp)
ZafEventMap	(z_win.hpp)
ZafEventStruct	(z_event.hpp)
ZafFile	(z_file.hpp)
ZafDiskFile	(z_dskfil.hpp)
ZafStorageFile	(z_store.hpp)
ZafFileInfoStruct	(z_file.hpp)
ZafFileSystem	(z_file.hpp)
ZafDiskFileSystem	(z_dskfil.hpp)
ZafStorage	(z_store.hpp)
ZafHelpStub	(z_help.hpp)
ZafHelpSystem <- ZafWindow	(z_help.hpp)
ZafImageStruct	(z_dsp.hpp)
ZafBitmapStruct	(z_dsp.hpp)
ZafBitmapData <- ZafImageData	(z_bmap.hpp)
ZafIconStruct	(z_dsp.hpp)
ZafIconData <- ZafImageData	(z_icon.hpp)
ZafMouseStruct	(z_dsp.hpp)
ZafMouseData <- ZafImageData	(z_mouse1.hpp)
ZafKeyStruct	(z_key.hpp)
ZafList	(z_list.hpp)
ZafDataRecord <- ZafData	(z_data.hpp)
ZafDataManager	(z_data.hpp)
ZafEventManager	(z_evtmgr.hpp)
ZafGeometryManager <- ZafWindowObject	(z_gmgr.hpp)
ZafListBlock	(z_list.hpp)
ZafQueueBlock	(z_evtmgr.hpp)
ZafPath	(z_file.hpp)
ZafWindow <- ZafWindowObject	(z_win.hpp)
ZafLocaleStruct	(z_loc.hpp)
ZafLocaleData <- ZafI18nData	(z_loc.hpp)
ZafMessageStruct	(z_lang.hpp)
ZafMSWindowsApp	(w_app.hpp)
ZafNotification	(z_notify.hpp)
ZafData <- ZafElement	(z_data.hpp)
ZafPaletteMap	(z_pal1.hpp)
ZafPaletteStruct	(z_pal.hpp)
ZafPositionStruct	(z_pos.hpp)
ZafRegionStruct	(z_region.hpp)
ZafScrollStruct	(z_scrll.hpp)
ZafScrollData <- ZafData	(z_scrll1.hpp)
ZafStandardArg	(z_stdarg.hpp)

ZafApplication

argc	Control	LinkMain
argv	Error	Main

Inheritance	Root Class
Declaration	<code>#include <z_app.hpp></code>
Description	<p>ZafApplication provides a platform independent entry point for ZAF applications called <code>ZafApplication::Main()</code> and a mechanism for managing global objects used by these applications.</p> <p>Execution of a C/C++ application begins with a call to <code>main()</code> or <code>WinMain()</code> (for Microsoft Windows applications). ZAF includes <code>main()</code> or <code>WinMain()</code> in its interface library so that it need not be provided by the ZAF developer. Instead, the ZAF developer provides <code>ZafApplication::Main()</code>.</p> <p>A <code>ZafApplication</code> class instance is created and destroyed within the <code>main()</code> or <code>WinMain()</code> function provided in the Zinc Application Framework library. This allows <code>ZafApplication</code> to perform initialization of global objects before <code>ZafApplication::Main()</code> is called, and to perform clean-up after. The following global pointers are initialized by the <code>ZafApplication</code> constructor:</p> <ul style="list-style-type: none">• <code>zafApplication</code>• <code>zafDisplay</code>• <code>zafEventManager</code>• <code>zafWindowManager</code>• <code>zafLanguageManager</code>• <code>zafLocale</code>• <code>zafCodeSet</code>• <code>zafI18nStorage</code>• <code>zafErrorSystem</code>• <code>zafSearchPath</code> <p><code>ZafApplication</code> also provides an event processing loop, <code>ZafApplication::Control()</code>. 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.</p>
Constructor	The <code>ZafApplication</code> constructor initializes global objects for use by ZAF applications. The constructor is called when a <code>ZafApplication</code> class instance is cre-

ated inside of the `main()` or `WinMain()` function found in the library. The `ZafApplication` constructor should not be called by the ZAF developer.

The `ZafApplication` constructor also initializes the member variables associated with an instantiated `ZafApplication` object. The default values set by the `ZafApplication` follow.

Member Initializations

ZafApplication

<code>Error()</code>	<code>ZAF_ERROR_NONE</code>
----------------------	-----------------------------

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. If it exists, *argv*[0] contains the full path name of the program being executed. Under Microsoft Windows, *argc* and *argv* are synthesized from the parameters passed to `WinMain()`.

Several global pointers are initialized by the constructor:

- `zafApplication`—a pointer to the application object
- `zafDisplay`—a pointer to the display
- `zafEventManager`—a pointer to the event manager
- `zafWindowManager`—a pointer to the window manager
- `zafLanguageManager`—a pointer to a `ZafLanguageManager` object containing information about the 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
- `zafI18nStorage`—a pointer to a `ZafStorage` object associated with `I18N.ZNC`, if it was found
- `zafErrorSystem`—a pointer to a `ZafErrorSystem` object for reporting errors
- `zafSearchPath`—a pointer to a `ZafPath` object associated with the application startup directory

In addition, `ZafKeyboard`, `ZafMouse`, `ZafCursor`, and `ZafHelpTips` devices are added to the event manager. See [ZafEventManager](#) for additional information.

Destructor `virtual ~ZafApplication(void);`

The ZafApplication destructor destroys all objects allocated by the constructor.

Members

argc `int argc;`

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

argv `ZafIChar **argv;`

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

Control `ZafEventType Control(ZafQFlags flags = Q_NORMAL);`

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(ZafQFlags flags)
{
    // Make sure there is a window attached to the window manager.
    if (!zafWindowManager->First())
        return (S_NO_OBJECT);
    // Wait for user response.
    ZafEventStruct event;
    ZafEventType ccode;
    do
    {
        if (zafEventManager->Get(event, flags) == 0)
            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.

Error

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

These functions get/set the error state of the ZafApplication. 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 ZafApplication:

Error Value	Description
ZAF_ERROR_NONE	No error exists.
ZAF_ERROR_CONSTRUCTOR	The ZafApplication was not properly constructed, and will exit automatically.

LinkMain

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

Main

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

Main() is the entry point for a ZAF application and must be provided by the ZAF programmer. 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](#)).

`Main()` is typically where the user would reinitialize internationalization for the program. For more information on adding internationalization, see the [ZafI18nData](#) chapter.

ZafAttachment

Event	Reference	Type
Offset	Reference-NumberID	
OppositeSide	Stretch	

Inheritance ZafAttachment : ZafConstraint : ZafElement

Declaration #include <z_gmgr.hpp>

Description ZafAttachment geometry management constraints allow window objects to be attached to the edges of sibling or parent window objects. ZafAttachment constraints must be added to the ZafGeometryManager object that has been added to the managed object's parent. (See [ZafGeometryManager](#) for more information.)

Constructors All ZafAttachment constructors initialize the member variables associated with an instantiated ZafAttachment object. The default values set by the ZafAttachment and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafAttachment

Offset()	-1
OppositeSide()	false
Reference()	null
ReferenceNumberID()	-1
Stretch()	false
Type()	user-supplied parameter

ZafElement

ClassID()	ID_ZAF_ATTACHMENT
ClassName()	"ZafAttachment"

```
ZafAttachment(ZafWindowObject *object, ZafAttachmentType
               type);
```

This constructor is useful in straight-code situations to create a ZafAttachment object. *object* specifies the window object the constraint applies to, and *type* specifies the type of attachment constraint. See [ZafConstraint::Object\(\)](#) and [ZafAttachment::Type\(\)](#) for more information.

```
ZafAttachment(const ZafAttachment &copy);
```

The copy constructor creates a new `ZafAttachment` object and initializes its data from *copy*.

```
ZafAttachment(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 an attachment constraint follows:

```
// Create a status bar with geometry-managed children.  
ZafStatusBar *stat = new ZafStatusBar(0, 0, 0, 1);  
ZafString *string = new ZafString(0, 0, 15, new  
    ZafStringData("String"));  
stat->Add(string);  
ZafTime *time = new ZafTime(15, 0, 15, new ZafTimeData);  
stat->Add(time);  
  
// Time field will remain at the right side.  
ZafAttachment *attach = new ZafAttachment(time, ZAF_ATCF_RIGHT);  
attach->SetOffset(0);  
  
// Create the geometry manager and add the first constraint.  
ZafGeometryManager *geo = new ZafGeometryManager;  
geo->Add(attach);  
  
// String field will occupy the remaining space.  
attach = new ZafAttachment(string, ZAF_ATCF_RIGHT);  
attach->SetStretch(true);  
attach->SetReference(time);  
attach->SetOppositeSide(true);  
attach->SetOffset(1);  
  
// Add the second constraint to the geometry manager.  
geo->Add(attach);  
  
// Add the geometry manager to the status bar.  
stat->Add(geo);
```

Destructor

```
virtual ~ZafAttachment(void);
```

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

Generally, the programmer will not directly destroy a ZafAttachment object, since it is automatically destroyed when its parent geometry manager is destroyed. For more information on child object deletion, see the [ZafWindow](#) destructor.

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.

Event

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

This overloaded function handles all events sent to the ZafAttachment object. The events handled by ZafAttachment are as follows:

ZafEventType	Description
S_COMPUTE_SIZE	causes the constraint to compute and modify the size or position of its window object
S_INITIALIZE	causes the constraint to initialize its numberID, stringID, Object(), and Reference()

Offset

```
int Offset(void) const;
int SetOffset(int offset);
```

Offset() is the distance from the edge of the Reference() object that the edge of Object() is positioned. Distances depend on Type() and OppositeSide(), and are specified in the same coordinate type as Object()->Region().

For example, if Type() is ZAF_ATCF_LEFT, Offset() is 0, and OppositeSide() is false, then the left edge of Object() will be set to the same value as the left edge of Reference(). This attribute defaults to -1, requiring the programmer to change it with SetOffset().

OppositeSide

```
bool OppositeSide(void) const;
bool SetOppositeSide(bool oppositeSide);
```

If OppositeSide() is true, then the opposite edge of the Reference() object is used when computing the position of Object().

For example, if `Type()` is `ZAF_ATCF_LEFT`, `Offset()` is 10, and `OppositeSide()` is true, then the left edge of `Object()` will be set to 10 pixels to the right of the right edge of `Reference()`. If `OppositeSide()` is false, then the left edge of `Object()` would be set to 10 pixels to the right of the left edge of `Reference()`. This attribute defaults to false, but the programmer may change it with `SetOppositeSide()`.

Reference

```
ZafWindowObject *Reference(void) const;  
ZafError SetReference(ZafWindowObject *reference);
```

`Reference()` is the window object whose `Region()` is used when computing the position of `Object()` (or the object to which this object is attached). If `Reference()` is null, the `Region()` of the parent of `Object()` is used as a reference. This attribute defaults to null, but the programmer may change it with `SetReference()`.

Reference- NumberID

```
ZafNumberID *ReferenceNumberID(void) const;  
void SetReferenceNumberID(ZafNumberID referenceNumberID);
```

`ReferenceNumberID()` specifies the numberID of the `Reference()` object. When reading a constraint from a persistent data file, the constraint is tied to its `Reference()` object via the numberID. This attribute defaults to , but the programmer may change it with `SetReferenceNumberID()`.

Stretch

```
bool Stretch(void) const;  
bool SetStretch(bool stretch);
```

If `Stretch()` is true, then the opposite side of `Object()` than that specified by `Type()` is not modified. For example, if `Type()` is `ZAF_ATCF_RIGHT`, `Offset()` is 0, and `Stretch()` is true, then the right edge of `Object()` will be set to the same value as the right edge of `Reference()` and the left edge of `Object()` will remain where it is, in effect making it possible to stretch `Object()`. If `Stretch()` is false, then the left edge of `Object()` would be modified so that the width of `Object()` would remain the same. This attribute defaults to false, but the programmer may change it with `SetStretch()`.

Type

```
ZafAttachmentType Type(void) const;  
ZafAttachmentType SetType(ZafAttachmentType type);
```

`Type()` is the attachment constraint's type, which specifies the side of `Object()` used when calculating its position. If `OppositeSide()` is false, then `Type()` also specifies the side of the `Reference()` object used when calculating the position

of `Object()`. If `OppositeSide()` is true, then the opposite side of the `Reference()` object is used when calculating the position of `Object()`. The programmer may use `SetType()` to change this attribute. The possible values for `Type()` follow:

Type()	Description
ZAF_ATCF_BOTTOM	causes the constraint to affect the bottom side of <code>Object()</code>
ZAF_ATCF_LEFT	causes the constraint to affect the left side of <code>Object()</code>
ZAF_ATCF_RIGHT	causes the constraint to affect the right side of <code>Object()</code>
ZAF_ATCF_TOP	causes the constraint to affect the top side of <code>Object()</code>

ZafBignum

BignumData	IValue	SetBignum
Event	RValue	

Inheritance	<code>ZafBignum : ZafString : ZafWindowObject : ZafElement</code>
Declaration	<code>#include <z_bnum1.hpp></code>
Description	<p>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.</p> <p>All ZafBignum objects refer to data contained in a ZafBignumData object (refer to this class for additional essential information).</p>

Formats	<p>ZafString and derived classes parse input and display output based on default or programmer-defined input and output formats. Programmers may use the SetInputFormat() and SetOutputFormat() families of functions to change the default format. These functions are documented in the ZafString reference.</p> <p>Each class derived from ZafFormatData handles a different set of formatting arguments, in addition to those supported by its base classes. ZafBignumData (and therefore ZafBignum) handles the following arguments in addition to those used by ZafReal, ZafInteger, and ZafString:</p>
----------------	---

Format Argument	Substitution
<code>%[\$ @+-,]B</code>	Bignum, with any combination of the following options:
<code>\$</code>	adds the currency symbol
<code>' '</code> (space character)	inserts a space between the currency symbol and the number
<code>@</code>	adds credit symbols such as parens for a negative number
<code>+</code>	adds a positive sign for positive numbers
<code>-</code>	causes the bignum to be left-justified
<code>,</code>	adds thousands separators

Constructors	<p>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</p>
---------------------	---

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

BignumData()	null
--------------	------

ZafString

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

ZafElement

ClassID()	ID_ZAF_BIGNUM
ClassName()	"ZafBignum"

```
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 the ZafBignum object is 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 that initially appears 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 ZafBignumData 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

BignumData

```
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](#). SetBignumData() will delete the previous BignumData() object if it is Destroyable() and no other object uses it.

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.

Event

```
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
N_RESET_I18N	causes the object to redisplay its data according to the new internationalization values
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

IValue

```
long IValue(void);
```

IValue() returns the value of the ZafBignumData associated with this ZafBignum as a long.

RValue

```
double RValue(void);
```

RValue() returns the value of the ZafBignumData associated with this ZafBignum as a double.

SetBignum

```
virtual ZafError SetBignum(long value);  
virtual ZafError SetBignum(double value);
```

SetBignum() sets the value of the **ZafBignumData** associated with this **ZafBignum** from *value*.

ZafBignumData

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

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

Declaration #include <z_bnum.hpp>

Description 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.

Constructors

ZafBignumData constructors initialize the member variables associated with a new ZafBignumData object and allocate space to hold the bignum data.

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

Member Initializations

ZafBignumData	
IValue()	(varies by constructor)
RValue()	(varies by constructor)
ZafElement	
ClassID()	ID_ZAF_BIGNUM_DATA
ClassName()	"ZafBignumData"

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*. Since a compiler error may result in passing a value of type int, such a value must be explicitly cast by the programmer. The data is automati-

cally 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 &copy);
```

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 stringBignum("123.45");
ZafBignumData zeroBignum;
```

Destructor

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

The destructor is used to free the memory associated with an instantiated ZafBignumData object. A ZafBignumData object is usually destroyed automatically when all ZafBignum objects that refer to it are destroyed (unless the last ZafBignum object referring to the ZafBignumData object has its StaticData() set to true).

Members

Clear

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

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

double operator **double**();
See [RValue\(\)](#).

FormattedText 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* characters. **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, 256);
printf("bignum - %s\n", buffer);

bignum.FormattedText(buffer, 256, "%.0B");
printf("integer - %s\n", buffer);

bignum.FormattedText(buffer, 256, "%5.2B");
printf("real - %s\n", buffer);

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

IValue long **IValue**(void) const;
long 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
```

```
operator long();
```

See [IValue\(\)](#).

RValue

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

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

SetBignum

```
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().

operator -

```
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);
```

operator --

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

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

operator +

```
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.

operator ++

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

These pre- and post-operators increment the ZafBignumData object's value by 1. See [operator -](#) for sample code.

*operator **

```
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.

operator /

```
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.

operator %

```
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.

```
operator =      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.

```
operator +=     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.

```
operator -=     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.

```
operator *=     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.

operator /=

```
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.

operator %=

```
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.

operator ==

```
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)
```

operator !=

```
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.

operator <

```
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)
```

```
operator <=      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)
operator >       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)
operator >=     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.

ZafBitmap

	AutoSize	BitmapData
Inheritance	ZafBitmap : ZafWindowObject : ZafElement	
Declaration	#include <z_bmap1.hpp>	
Description	ZafBitmap displays a portable bitmap after automatically converting it to the environment's native format. ZafBitmap is useful for displaying bitmap information portably where user interaction is not required.	
Constructors	All ZafBitmap constructors initialize the member variables associated with an instantiated ZafBitmap object. The default values set by the ZafBitmap 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 ZafBitmap. “†” Indicates a blocking function that prevents changes to the attribute in this class.	

Member Initializations

ZafBitmap

AutoSize()	true
BitmapData()	null

ZafWindowObject

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

ZafElement

ClassID()	ID_ZAF_BITMAP
ClassName()	"ZafBitmap"

```
ZafBitmap(int left, int top, int width, int height,  
           ZafBitmapData *bitmapData = ZAF_NULLP(ZafBitmapData));
```

This constructor is useful in straight-code situations. *left*, *top*, *width*, and *height* specify the position and size of the bitmap on its parent. 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 **ZafBitmap** object. *bitmapData* may be shared by many **ZafBitmaps** to conserve system resources or ensure consistency. See [ZafDataManager](#) for information on sharing data objects.

```
ZafBitmap(const ZafBitmap &copy);
```

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

```
ZafBitmap(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 is a code snippet that shows how to create a **ZafBitmap** object.

```
// Create a sample window with some bitmap objects.  
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);  
  
// Add the bitmaps to the window.  
extern ZafBitmapData *bitmapData1, *bitmapData2;  
window1->Add(new ZafBitmap(1, 1, 2, 2, bitmapData1));  
window1->Add(new ZafBitmap(4, 1, 2, 2, bitmapData2));
```

Destructor

```
virtual ~ZafBitmap(void);
```

The destructor is used to free the memory associated with a **ZafBitmap** 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 **ZafBitmap** 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.

AutoSize

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

If AutoSize() is true, the ZafBitmap object will automatically grow to the size it needs to be to display all of its data. The default value of this attribute is true, but the user may call SetAutoSize() to change it.

BitmapData

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

The BitmapData() object is where the actual image data is stored. The BitmapData() may be shared among several ZafBitmap or ZafButton objects (perhaps to save memory by allowing many objects to use the same image data), or it may belong to a single ZafBitmap object. If shared among several ZafBitmap objects, all the associated ZafBitmap objects will be updated when the BitmapData() changes. SetBitmapData() may be used to associate a BitmapData() object with a ZafBitmap object. For more information on data sharing in ZAF, see [ZafDataManager](#). SetBitmapData() will delete the previous BitmapData() object if it is Destroyable() and no other object uses it.

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

ZafBitmapData

Array	Height	Width
Clear	SetBitmap	operator =

Inheritance	<code>ZafBitmapData : ((ZafImageData : ZafData : (ZafNotification, ZafElement)), ZafBitmapStruct)</code>
Declaration	<code>#include <z_bitmap.hpp></code>
Description	ZafBitmapData objects can be used to store and manipulate bitmap information. ZafBitmapData is used in conjunction with ZAF classes that utilize bitmap information, such as ZafButton.
Constructors	All ZafBitmapData constructors initialize the member variables associated with an instantiated ZafBitmapData object. The default values set by the ZafBitmapData and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafBitmapData

<code>Array()</code>	<code>null</code>
----------------------	-------------------

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_BITMAP_DATA</code>
<code>ClassName()</code>	<code>"ZafBitmapData"</code>

```
ZafBitmapData(const ZafImageStruct &data);  
ZafBitmapData(const ZafBitmapStruct &data);
```

These constructors allocate a ZafBitmapData instance and initialize its data to the values in *data*.

```
ZafBitmapData(const ZafBitmapData &copy);
```

The copy constructor creates a new ZafBitmapData object and initializes its data from *copy*.

```
ZafBitmapData(const ZafIChar *name, ZafDataPersistence
    &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 code snippet shows how to create a ZafBitmapData object:

```
// Create a bitmap array.
#define gnd ZAF_CLR_BACKGROUND
#define blk ZAF_CLR_BLACK
static ZafLogicalColor ZAF_FARDATA upArrowArray[28] =
{
    gnd,gnd,gnd,blk,gnd,gnd,gnd,
    gnd,gnd,blk,blk,blk,gnd,gnd,
    gnd,blk,blk,blk,blk,blk,gnd,
    blk,blk,blk,blk,blk,blk,blk
};
static ZafBitmapStruct upArrowStruct(7, 4, upArrowArray, true);

// Create ZafBitmapData objects based on the bitmap array.
ZafBitmapData bitmap1(upArrowStruct);
ZafBitmapData bitmap2(bitmap1);
```

Destructor

```
virtual ~ZafBitmapData(void);
```

This virtual destructor is used to free the memory associated with an instantiated ZafBitmapData object.

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.

Array

```
ZafLogicalColor *Array(void) const;
```

Array() returns the portable logical color array that the bitmap data is based on. This array is converted to a native environment bitmap handle. Each element of the array is a ZAF logical color. See [ZafDisplay](#) for more information.

Clear

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

Clear() destroys the portable Array() if StaticArray() is false, and it destroys the environment handle if StaticHandle() is false. Regardless of StaticArray()

and `StaticHandle()`, the portable `Array()` and the environment handle are both set to null, effectively clearing the bitmap data.

Height

```
int Height(void) const;
```

`Height()` returns the height of the bitmap data.

SetBitmap

```
virtual ZafError SetBitmap(int width, int height,  
    ZafLogicalColor *array);  
virtual ZafError SetBitmap(const ZafBitmapStruct  
    &bitmap);
```

These functions copy the data passed in to the `ZafBitmapData` object. *width* and *height* are the width and height of *array*, respectively. If `StaticArray()` is true, *array* becomes `Array()`; otherwise, a new array is created for `Array()`. In the second function, the data is copied from *bitmap*. These functions always return `ZAF_ERROR_NONE`.

Width

```
int Width(void) const;
```

`Width()` returns the width of the bitmap data.

operator =

```
ZafBitmapData &operator=(const ZafBitmapData &bitmap);
```

This operator copies the data from *bitmap* into this `ZafBitmapData` object.

ZafBitmapStruct

[StaticHandle](#)

Inheritance	ZafBitmapStruct : ZafImageStruct
Declaration	<code>#include <z_dsp.hpp></code>
Description	ZafBitmapStruct is used to store bitmap information. The base ZafImageStruct stores the portable image array using elements of ZafLogicalColors, and ZafBitmapStruct defines additional members that store environment-specific structures for the bitmap filled by the ZafDisplay conversion function ZafDisplay::ConvertToOSBitmap().
Constructors	All ZafBitmapStruct constructors initialize the member variables associated with an instantiated ZafBitmapStruct object. The default values set by the ZafBitmapStruct constructors follow.

Member Initializations

ZafBitmapStruct

StaticHandle()	false
----------------	-------

ZafImageStruct

array	null
height	0
StaticArray()	false
width	0

ZafBitmapStruct(void);

This constructor allocates a ZafBitmapStruct instance and initializes its data to indicate that no bitmap information has been set.

ZafBitmapStruct(const ZafImageStruct &data);

This constructor allocates a ZafBitmapStruct instance and initializes its data to the values in *data*. The environment-specific bitmap information is initialized to indicate that the bitmap has not yet been converted.

```
ZafBitmapStruct(int width, int height, ZafLogicalColor  
                  *array, bool staticArray);
```

This constructor allocates a `ZafBitmapStruct` instance and initializes its data to the values passed in. *width* and *height* indicate the size of the image, *array* specifies a pointer to the portable array of `ZafLogicalColors`, and *staticArray* indicates if *array* is declared static.

The following code snippet shows how to create a `ZafBitmapStruct` object:

```
// Create a bitmap structure.  
#define gnd ZAF_CLR_BACKGROUND  
#define blk ZAF_CLR_BLACK  
static ZafLogicalColor ZAF_FARDATA upArrowArray[28] =  
{  
    gnd,gnd,gnd,blk,gnd,gnd,gnd,  
    gnd,gnd,blk,blk,blk,gnd,gnd,  
    gnd,blk,blk,blk,blk,blk,gnd,  
    blk,blk,blk,blk,blk,blk,blk  
};  
static ZafBitmapStruct upArrowStruct(7, 4, upArrowArray, true);
```

Members

StaticHandle

```
bool StaticHandle(void) const;  
bool SetStaticHandle(bool staticHandle);
```

If `StaticHandle()` is true, the environment-specific information (generally known as a handle) of the `ZafBitmapStruct` is recognized as static, so that it will not be deleted by ZAF, and may be used by several `ZafBitmapStruct` objects. This attribute is initialized to false, but it may be changed with `SetStaticHandle()`. Both functions return the current value of the `StaticHandle()` attribute.

ZafBorder

Width

Inheritance ZafBorder : ZafWindowObject : ZafElement

Declaration #include <z_border.hpp>

Description 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. The ZafBorder is used to size the parent window with the ZafMouse device.

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 [†]
RegionType()	ZAF_OUTSIDE_REGION [†]
Selected()	false [†]

Member Initializations

<code>SupportObject()</code>	<code>true</code> [†]
<code>UserFunction()</code>	<code>null</code> [†]

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_BORDER</code>
<code>ClassName()</code>	<code>"ZafBorder"</code>
<code>NumberID()</code>	<code>ZAF_NUMID_BORDER</code>
<code>StringID()</code>	<code>"ZAF_NUMID_BORDER"</code>

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*Width*

```
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	Depressed	SelectOnDownClick
AllowToggling	Depth	SendMessage
AutoRepeatSelection	Event	SendMessageText
AutoSize	HotKeyChar	SendMessageWhenSelected
BitmapData	HotKeyIndex	StringData
ButtonType	HzJustify	Text
ClearImage	Region	Value
ClearText	SelectOnDoubleClick	VtJustify

Inheritance ZafButton : ZafWindowObject : ZafElement

Declaration #include <z_button.hpp>

Description 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. Note: AutoSize() buttons (which is the default) grow up rather than down as other objects. This behavior will become optional in a future version.

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

<code>HzJustify()</code>	<code>ZAF_HZ_CENTER</code>
<code>SelectOnDoubleClick()</code>	<code>false</code>
<code>SelectOnDownClick()</code>	<code>false</code>
<code>SendMessageText()</code>	<code>null</code>
<code>SendMessageWhen-Selected()</code>	<code>false</code>
<code>StringData()</code>	<code>ZAF_NULLP(ZafStringData)</code>
<code>Value()</code>	<code>0</code>
<code>VtJustify()</code>	<code>ZAF_VT_CENTER</code>

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_BUTTON</code>
<code>ClassName()</code>	<code>"ZafButton"</code>

```
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 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. *buttonType* 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 manually, 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 some `ZafStringData` objects and cre-

ate `ZafButton` objects using the `ZafStringData` objects. For more information on using `ZafStringData` objects, see [ZafStringData](#). *stringData* specifies the string data object to be associated with the button. Either *bitmapData* or *stringData* 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. If the data objects 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,  
          ZAF_NULLP(ZafStringData), bitmapData));  
window1->Add(new ZafButton(20, 4, 15, 1, "Button2",  
          ZAF_NULLP(ZafBitmapData)));  
...  
// 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, "Choice 1",  
          ZAF_NULLP(ZafBitmapData), ZAF_RADIO_BUTTON));  
group->Add(new ZafButton(1, 1, 25, 1, "Choice 2",  
          ZAF_NULLP(ZafBitmapData), ZAF_RADIO_BUTTON));
```

```
group->Add(new ZafButton(1, 2, 25, 1, "Choice 3",
    ZAF_NULLP(ZafBitmapData), ZAF_RADIO_BUTTON));
group->Add(new ZafButton(1, 3, 25, 1, "Choice 4",
    ZAF_NULLP(ZafBitmapData), 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.

AllowDefault

```
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, "OK",
    ZAF_NULLP(ZafBitmapData));
ZafButton *button2 = new ZafButton(15, 4, 12, 1, "Cancel",
    ZAF_NULLP(ZafBitmapData));
button1->SetAllowDefault(true);
button2->SetAllowDefault(true);
window->Add(button1);
window->Add(button2);
window->SetDefaultButton(button1);
```

AllowToggling

```
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, "Toggle Button",
    ZAF_NULLP(ZafBitmapData));
button->SetAllowToggling(true);
```

AutoRepeat- Selection

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

If `AutoRepeatSelection()` is true, the button performs its action repeatedly at a delay rate specified by the environment as long as the button is depressed by

the user. 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, "Repeat Button",
    ZAF_NULLP(ZafBitmapData));
button->SetSendMessageWhenSelected(true);
button->SetValue(USER_ACTION_EVENT);
button->SetAutoRepeatSelection(true);
```

AutoSize

```
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.

BitmapData

```
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](#). `SetBitmapData()` will delete the previous `BitmapData()` object if it is `Destroyable()` and no other object uses it.

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);
```

ButtonType

```
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:

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

The parent's `SelectionType()` directly affects the behavior of buttons on a window. For example, a group with radio buttons in it must be `SelectionType() == ZAF_SINGLE_SELECTION` to behave as expected, and a group with check boxes in it must be `SelectionType() == ZAF_MULTIPLE_SELECTION` to behave as expected. Only one radio button should be selected at a time, whereas multiple check boxes may be selected at a time. This way, the parent correctly acts as the container of the button objects and the button objects need not know of their siblings' states. See `ZafWindow::SelectionType()` for more information.

Check boxes and radio buttons should also set `AllowToggling()` to true, since they generally toggle selection state when selected (or when another radio button is selected). See [AllowToggling\(\)](#) for more information.

ClearImage

```
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.

ClearText

```
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.

Depressed

```
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, since each environment handles button depression differently. `Selected()` and `SetSelected()` provide correct support for selecting and de-selecting a button.

Depth

```
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, since each environment dictates its own button depth.

Event

```
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
<code>S_HOT_KEY</code>	Causes the object to perform its action, if <code>event.key.value</code> is the object's hot key
<code>S_REDISPLAY_DEFAULT</code>	Causes the object to display or erase its default border, as appropriate

HotKeyChar
HotKeyIndex

```
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.

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. *hotKeyChar* specifies the hot key character, and *index* 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, "Quit",  
    ZAF_NULLP(ZafBitmapData));  
button->SetHotKey('Q', 0);
```

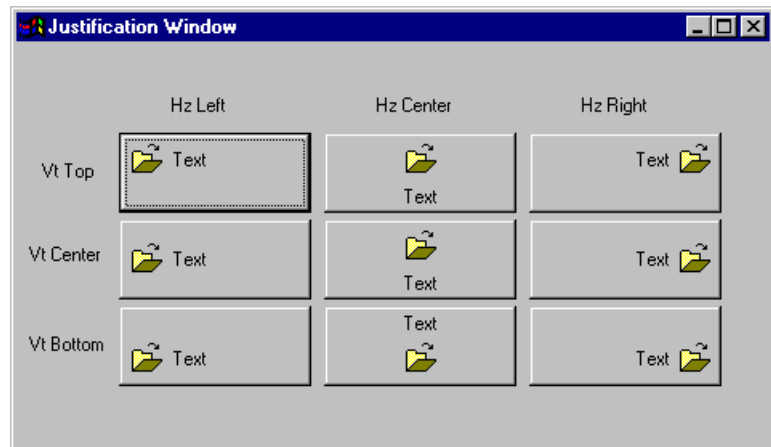
HZJustify
VtJustify

```
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. The following picture illustrates how justification effects a button object.



Region

```
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.

SelectOnDoubleClick

```
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.

SelectOnDownClick

```
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.

SendMessage

```
ZafEventType SendMessage(const ZafEventStruct &event,  
    ZafEventType ccode);
```

SendMessageWhenSelected

```
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 programmer 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, "Action",
    ZAF_NULLP(ZafBitmapData));
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, "Quit",
    ZAF_NULLP(ZafBitmapData));
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. If `SendMessageText()` is non-null, then `SendMessageText()` is copied into event.text, and the object handling the event must delete event.text. The programmer should never need to call `SendMessage()`, since it is called internally by the `ZafButton` class.

Either `SendMessageWhenSelected()` or `UserFunction()` may be true, but not both.

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.

SendMessageText

```

const ZafIChar *SendMessageText(void) const;
virtual const ZafIChar *SetSendMessageText(const ZafIChar
    *text);

```

`SendMessageText()` stores the text that is copied to event.text when a message is posted to the event manager's queue for `SendMessageWhenSelected()` buttons. See [SendMessageWhenSelected\(\)](#) for more information. `SendMessageText()` is null by default, but the user may call `SetSendMessageText()` to change it.

The object handling the event where event.text contains a duplicate of SendMessageText() must delete event.text. Otherwise, a memory leak will occur. If SendMessageText() is null, there is no need to delete event.text. The following code snippet shows how to create a button that utilizes SendMessageText().

```
// Create a button that closes an extra window.
ZafButton *button = new ZafButton(1, 4, 12, 1, "Clean Up",
    ZAF_NULLP(ZafBitmapData));
button->SetSendMessageWhenSelected(true);
button->SetValue(A_CLOSE_WINDOW);

// The window to be closed has a stringID of "EXTRA_WINDOW".
button->SetSendMessageText("EXTRA_WINDOW");
```

StringData

```
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](#). SetStringData() will delete the previous StringData() object if it is Destroyable() and no other object uses it.

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.

Text

```
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.

Value

```
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.

VtJustify

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

See [HzJustify\(\)](#).

ZafChart

ActiveModel	DataGroup	IndexMethod
AddDataGroup	DataGroupBackgroundColor	MarksXAxis
AddDataPoint	DataGroupFillPattern	MarksYAxis
AxisColor	DataGroupFont	TextTitle
ChartType	DataGroupForegroundColor	TextXAxis
ClearDataAll	DataGroupLineStyle	TextYAxis
ClearDataGroup	DataPoint	

Inheritance	ZafChart : ZafChartStub : ZafWindowObject : ZafElement
Declaration	#include <cf_api/bz_zcht.hpp>
Description	<p>ZafChart supports graphical charting of data groups. Each data group is represented by points, lines, bars, pie slices, or another representation.</p> <p>ZafChart is a unique ZAF class. It is a “wrapper” class serving only as a ZAF-like interface to a third-party library: DPC Technology’s ChartFolio™. Basic use of the ZafChart class is supported directly by Zinc Software. Technical support for specific functionality and underlying ChartFolio operation is provided directly by DPC Technology.</p> <p>The “cf_zaf” directory of the ZAF 5 distribution contains the source code for a limited version of DPC’s ChartFolio library. Before utilizing the ZafChart class, the ChartFolio library must be built using the supplied make files. The cf_zaf/include directory must be part of the compiler’s include search path in order to utilize the ChartFolio header files.</p> <p>The limited ChartFolio library supplied with ZAF provides two-dimensional chart types including pie, bar, column, stack, point-series, line, and integral/area. Data-domain indexing modes supplied include natural number (n=1, 2, 3, .. N) and integral number (i=0, 1, 2, .. I).</p> <p>DPC Technology also publishes advanced, full-featured versions of ChartFolio that provide complete 3-D charting support, additional chart types, real-time rotation and scaling, real number value indexing, user dialog and chart control objects, and more. These add-on products are available from Zinc Software, or directly from DPC Technology.</p>

DPC Technology Corporation
9586 Easter Way
San Diego, CA
92121 USA

Tel: 619 / 622-1887

Fax: 619 / 622-1833

<http://www.dpc-tech.com>

Email: info@dpc-tech.com

Constructors

All ZafChart constructors initialize the member variables associated with an instantiated ZafChart object. The default values set by the ZafChart 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 ZafChart. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafChart

AxisColor()	ZAF_CLR_BLACK
ChartType()	user-supplied parameter
IndexMethod()	user-supplied parameter
MarksXAxis()	true
MarksYAxis()	true
TextTitle()	null
TextXAxis()	null
TextYAxis()	null

ZafWindowObject

AcceptDrop()	false [†]
Changed()	false [†]
CopyDraggable()	false [†]
Disabled()	false [†]
Focus()	false [†]
HelpContext()	null [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
Noncurrent()	true [†]
OSDraw()	false

ZafElement

ClassID()	ID_ZAF_CHART
ClassName()	"ZafChart"

```
ZafChart(int left, int top, int width, int height, int  
         iChartTypeID = ZAF_POINTS_2D, int iIndexMethodID =  
         ZAF_RANGE_NATURAL);
```

This constructor is useful in straight-code situations. *left*, *top*, *width*, and *height* specify the position and size of the object on its parent. These values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. *iChartTypeID* specifies the chart type (see [Chart-Type\(\)](#) for more information). *iIndexMethodID* specifies the chart index method (see [IndexMethod\(\)](#) for more information).

```
ZafChart(const ZafChart &copy);
```

The copy constructor creates a new `ZafChart` object and initializes its data from *copy*. Data groups and data points are part of the `ChartFolio` data as opposed to the user interface `ZafChart` object, and are not copied. The programmer must create new data groups and new data points for the copy.

```
ZafChart(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. Data groups and data points are part of the `ChartFolio` data as opposed to the user interface `ZafChart` object, and are not stored. The programmer must create new data groups and new data points for the new `ZafChart` object.

An example of how to create a chart object follows:

```
// Create the sample window.  
ZafWindow *window = new ZafWindow(0, 0, 35, 10);  
window->AddGenericObjects(new ZafStringData("Sample Chart  
Window"));  
  
// Create a line chart.  
ZafChart *chart = new ZafChart(1, 0, 30, 7, CF_ID_LINE_2D,  
                               CF_ID_YVALS_ON_NAT_NS);  
  
// Define 5 data groups of 10 values each and add them to the  
// chart.  
double chartData[5][10] =  
{  
    { 20.0, 10.0, 20.0, 10.0, 20.0, 10.0, 20.0, 10.0, 20.0, 10.0 },  
    { 20.0, 18.0, 14.0, 2.0, 14.0, 37.0, 35.0, 30.0, 24.0, 23.0 },  
    { 33.0, 30.0, 27.0, 24.0, 21.0, 18.0, 15.0, 12.0, 9.0, 6.0 },  
    { 20.0, 15.0, 10.0, 12.0, 14.0, 20.0, 24.0, 28.0, 30.0, 32.0 },  
    { 16.0, 15.0, 23.0, 17.0, 24.0, 20.0, 18.0, 13.0, 28.0, 16.0 }  
}
```

```

};
for (int i = 0; i < 5; i++)
{
    // Add each data group (data groups are one-based).
    chart->AddDataGroup(i + 1);

    // Add each data point in the data group.
    for (int j = 0; j < 10; j++)
        chart->AddDataPoint(i + 1, chartData[i][j]);
}

// Add the chart to the window.
window->Add(chart);

```

Destructor

```
virtual ~ZafChart(void);
```

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

Generally, the programmer will not directly destroy a ZafChart 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.

ActiveModel

```
cf_ChartModel *ActiveModel(void);
```

ActiveModel() returns a pointer to the active ChartFolio chart model object associated with this ZafChart. ActiveModel() is an advanced routine and should normally not be called by the programmer. Consult ChartFolio documentation for detailed information on the use of a cf_ChartModel.

AddDataGroup

```
ZafError AddDataGroup(int groupIndex);
```

AddDataGroup() creates a new data group, referenced by one-based *groupIndex*. The first data group that should be allocated is 1, then 2, and so on. Failure to create new data groups sequentially will cause undefined results by the ChartFolio library. AddDataGroup() returns ZAF_ERROR_INVALID if an error occurred; otherwise ZAF_ERROR_NONE is returned.

AddDataPoint

```
ZafError AddDataPoint(int groupIndex, double dDataValue);
```

AddDataPoint() creates a new data point in the previously-created data group referenced by one-based *groupIndex*. *dDataValue* is the value used by the new data point. AddDataPoint() returns ZAF_ERROR_INVALID if an error occurred; otherwise ZAF_ERROR_NONE is returned.

AxisColor

```
ZafLogicalColor AxisColor(ZafLogicalColor *mono =
    ZAF_NULLP( ZafLogicalColor ) );
ZafLogicalColor SetAxisColor(ZafLogicalColor color,
    ZafLogicalColor mono = ZAF_MONO_NULL);
```

AxisColor() specifies the logical color used when drawing the axes and the tick marks of the chart. AxisColor() defaults to ZAF_CLR_BLACK, but may be changed by calling SetAxisColor(). *color* specifies the logical color to be used when drawing the axes and the tick marks of the chart, but both routines ignore *mono* since black and white charts are not supported by the ChartFolio library.

ChartType

```
int ChartType(void) const;
int SetChartType(int chartTypeID);
```

ChartType() specifies the chart display type. SetChartType() may be used to change the chart display type. Any of the following values defined by the ChartFolio library may be used for the ChartType(). Note that 3-D charts are not supported by the limited version of ChartFolio supplied with ZAF 5, but are available in the add-on product:

ChartType()	Description
CF_ID_AREA_2D	Displays the data in a 2-D area chart
CF_ID_BAR_2D	Displays the data in a 2-D horizontal bar chart
CF_ID_COL_2D	Displays the data in a 2-D vertical (column) bar chart
CF_ID_LINE_2D	Displays the data in a 2-D line chart
CF_ID_PIE_2D	Displays the data in a 2-D pie chart
CF_ID_POINTS_2D	Displays the data in a 2-D point series chart
CF_ID_STACK_2D	Displays the data in a 2-D vertical stack chart

ClearDataAll

```
ZafError ClearDataAll(void);
```

ClearDataAll() deletes all the data points in all data groups currently used by the chart. The data groups themselves are not deleted. ClearDataAll() returns ZAF_ERROR_INVALID if an error occurred; otherwise ZAF_ERROR_NONE is returned.

ClearDataGroup

```
ZafError ClearDataGroup(int groupIndex);
```

ClearDataGroup() deletes all the data points in the data group referred to by one-based index *groupIndex*. The data group itself is not deleted. ClearDataGroup() returns ZAF_ERROR_INVALID if an error occurred; otherwise ZAF_ERROR_NONE is returned.

DataGroup

```
cf_DataGroup *DataGroup(int groupIndex);
```

DataGroup() returns a pointer to the ChartFolio data group object referred to by one-based index *groupIndex*. DataGroup() is an advanced routine and should normally not be called by the programmer. Consult ChartFolio documentation for detailed information on the use of a cf_DataGroup.

**DataGroupBack-
groundColor**

```
ZafLogicalColor DataGroupBackgroundColor(int groupIndex,  
      ZafLogicalColor *mono = ZAF_NULLP(ZafLogicalColor));  
ZafLogicalColor SetDataGroupBackgroundColor(int  
      groupIndex, ZafLogicalColor color, ZafLogicalColor  
      mono = ZAF_MONO_NULL);
```

DataGroupBackgroundColor() specifies the logical color used when drawing the background pieces of the data group referred to by one-based index *groupIndex*. Each data group's DataGroupBackgroundColor() is initially set by the ChartFolio library, but may be changed by calling SetDataGroupBackgroundColor(). *color* specifies the logical color to be used when drawing the background pieces of the data group, but both routines ignore *mono* since black and white charts are not supported by the ChartFolio library.

**DataGroupFill-
Pattern**

```
ZafLogicalFillPattern DataGroupFillPattern(int  
      groupIndex) const;  
ZafLogicalFillPattern SetDataGroupFillPattern(int  
      groupIndex, ZafLogicalFillPattern fillPattern);
```

DataGroupFillPattern() specifies the logical fill pattern used when drawing the data group referred to by one-based index *groupIndex*. Each data group's DataGroupFillPattern() is initially set by the ChartFolio library, but may be changed by calling SetDataGroupFillPattern(). *fillPattern* specifies the logical fill pattern to be used when drawing the data group.

DataGroupFont

```
ZafLogicalFont DataGroupFont(int groupIndex) const;
```

```
ZafLogicalFont SetDataGroupFont(int groupIndex,  
    ZafLogicalFont font);
```

`DataGroupFont()` specifies the logical font used when drawing the textual information of the data group referred to by one-based index *groupIndex*. Each data group's `DataGroupFont()` is initially set by the ChartFolio library, but may be changed by calling `SetDataGroupFont()`. *font* specifies the logical font to be used when drawing the textual information of the data group.

*DataGroup-
ForegroundColor*

```
ZafLogicalColor DataGroupForegroundColor(int groupIndex,  
    ZafLogicalColor *mono = ZAF_NULLP(ZafLogicalColor));  
ZafLogicalColor SetDataGroupForegroundColor(int  
    groupIndex, ZafLogicalColor color, ZafLogicalColor  
    mono = ZAF_MONO_NULL);
```

`DataGroupForegroundColor()` specifies the logical color used when drawing the foreground pieces of the data group referred to by one-based index *groupIndex*. Each data group's `DataGroupForegroundColor()` is initially set by the ChartFolio library, but may be changed by calling `SetDataGroupForegroundColor()`. *color* specifies the logical color to be used when drawing the foreground pieces of the data group, but both routines ignore *mono* since black and white charts are not supported by the ChartFolio library.

DataGroupLineStyle

```
ZafLogicalLineStyle DataGroupLineStyle(int groupIndex)  
    const;  
ZafLogicalLineStyle SetDataGroupLineStyle(int groupIndex,  
    ZafLogicalLineStyle lineStyle);
```

`DataGroupLineStyle()` specifies the logical line style used when drawing the data group referred to by one-based index *groupIndex*. Each data group's `DataGroupLineStyle()` is initially set by the ChartFolio library, but may be changed by calling `SetDataGroupLineStyle()`. *lineStyle* specifies the logical line style to be used when drawing the data group.

DataPoint

```
cf_XYZ *DataPoint(int groupIndex, int dataIndex);
```

`DataPoint()` returns a pointer to the ChartFolio data point object referred to by one-based data group index *groupIndex* and one-based data point index *dataIndex*. `DataPoint()` is an advanced routine and should normally not be called by the programmer. Consult ChartFolio documentation for detailed information on the use of a `cf_XYZ`.

IndexMethod

```
int IndexMethod(void) const;
```

```
int SetIndexMethod(int indexMethodID);
```

IndexMethod() specifies the index method used by the domain axis of the chart. SetIndexMethod() may be used to change the domain index method. Any of the following values defined by the ChartFolio library may be used for the IndexMethod(). Note that extended indexing modes, including real number value indexing, are available in the add-on product:

ChartType()	Description
CF_ID_YVALS_ON_INT_IS	Uses non-negative integral values for the domain
CF_ID_YVALS_ON_NAT_NS	Uses natural (counting) values for the domain

MarksXAxis

```
bool MarksXAxis(void) const;  
bool SetMarksXAxis(bool xAxisMarks);
```

If MarksXAxis() is true, the x-axis of the chart is drawn with tick marks and incremental values (if applicable according to the ChartType()). MarksXAxis() defaults to true, but may be changed by calling SetMarksXAxis().

MarksYAxis

```
bool MarksYAxis(void) const;  
bool SetMarksYAxis(bool yAxisMarks);
```

If MarksYAxis() is true, the y-axis of the chart is drawn with tick marks and incremental values (if applicable according to the ChartType()). MarksYAxis() defaults to true, but may be changed by calling SetMarksYAxis().

TextTitle

```
const ZafIChar *TextTitle(void);  
const ZafIChar *SetTextTitle(const ZafIChar *text);
```

TextTitle() specifies the text used when drawing the title of the chart. TextTitle() defaults to null (in which case no title is drawn with the chart), but may be changed by calling SetTextTitle().

TextXAxis

```
const ZafIChar *TextXAxis(void);  
const ZafIChar *SetTextXAxis(const ZafIChar *text);
```

TextXAxis() specifies the text used when drawing the x-axis label for the chart (if applicable according to the ChartType()). TextXAxis() defaults to null (in which case no label is drawn with the x-axis), but may be changed by calling SetTextXAxis().

TextYAxis

```
const ZafIChar *TextYAxis(void);  
const ZafIChar *SetTextYAxis(const ZafIChar *text);
```

TextYAxis() specifies the text used when drawing the y-axis label for the chart (if applicable according to the ChartType()). TextYAxis() defaults to null (in which case no label is drawn with the y-axis), but may be changed by calling SetTextYAxis().

ZafChartStub

Inheritance ZafChartStub : [ZafWindowObject](#) : [ZafElement](#)

Declaration `#include <z_chart.hpp>`

Description ZafChartStub serves solely as a base class for ZafChart, so that ZafChart—and the third-party ChartFolio library it relies on—are not linked into a program that does not make use of it. Zinc Designer uses this class to allow editing of a ZafChart persistent object without the need to link in the overhead. The programmer should normally not derive from this class. See [ZafChart](#) for more information.

ZafCodeSetData

Clear	ConvertToOSWChar	IsoToUnicode
CodeSetAllocate	ConvertToOSWString	mblen
CodeSetFree	ConvertToZafChar	mbstowcs
CodeSetName	ConvertToZafString	NativeMapping
ConvertToOSChar	DefaultLeadByte	UnicodeToISO
ConvertToOSStr	DirSepStr	wcstombs

Inheritance ZafCodeSetData : ZafI18nData : ZafData :
 ZafNotification, ZafElement

Declaration #include <z_cset.hpp>

Description Different environments use different character sets to represent characters and strings internally. For example, some environments use Unicode as the native character set, and others use multi-byte strings. ZAF provides support for ISO 8859-1 and Unicode character mapping. By default, the ZAF libraries support ISO 8859-1, but the programmer may uncomment the ZAF_UNICODE macro definition in the header file z_env.hpp and recompile the libraries to support Unicode character mapping.

Though ISO 8859-1 characters are all single-byte characters and Unicode characters are all double-byte characters, some systems may use multi-byte characters, which may use any number of bytes for a single character. ZAF provides transparent support for all these systems by using the ZafCodeSetData methods when passing information to and from the native environment. The ZAF programmer simply has to use ZafIChar characters and strings, which always use ISO 8859-1 or Unicode character mapping. By commenting out both ZAF_ISO8859_1 and ZAF_UNICODE macros in the header file z_env.hpp, no mapping will occur, and the programmer may use native characters and strings, albeit unportably.

ZafCodeSetData provides support for mapping characters and strings between ISO 8859-1 or Unicode character sets (depending on the mode of the application) and the native environment's character set. For example, before calling a native API, a ZafIChar character or string must at times be converted to the native environment's format so the API will be able to utilize it correctly. Internally to the ZAF library, ZafCodeSetData is utilized when appropriate, since all ZAF interfaces use ZafIChar characters and strings. The mapping tables that enable this functionality are included in the file i18n.znc, which must be available in the same directory as the ZAF application that requires these tables. The i18n.znc file is shipped in the zaf/bin directory.

The programmer should never construct a ZafCodeSetData object, since one is created by ZafApplication which may be accessed via the global zafCodeSet. And normally, the programmer will not need to call these routines. However, advanced programmers may call a native API, and may be required to call the routines in ZafCodeSetData to convert values passed to the native API.

Constructors

All ZafCodeSetData constructors initialize the member variables associated with an instantiated ZafCodeSetData object. The default values set by the ZafCodeSetData and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafCodeSetData

CodeSetName()	environment-specific
DefaultLeadByte()	0
dirSepStr[2]	environment-specific
NativeMapping()	false (Unicode mode only)

ZafElement

ClassID()	ID_ZAF_CODE_SET_DATA
ClassName()	"ZafCodeSetData"

```
ZafCodeSetData(ZafIChar *isoToLocal, ZafIChar
    *localToIso);
```

These constructors allocate a ZafCodeSetData instance. *isoToLocal* specifies the table used to map ISO 8859-1 characters to native characters, and *localToIso* specifies the table used to map native characters to ISO 8859-1 characters.

```
ZafCodeSetData(const ZafCodeSetData &copy);
```

The copy constructor creates a new ZafCodeSetData object and initializes its data from *copy*.

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

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

Destructor `virtual ~ZafCodeSetData(void);`

This virtual destructor is used to free the memory associated with an instantiated ZafCodeSetData object.

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.

Clear `virtual void Clear(void);`

Clear() does nothing, but provides a definition for the base class ZafData pure virtual function.

CodeSetAllocate `static void CodeSetAllocate(const ZafIChar *name = ZAF_NULLP(ZafIChar));`

CodeSetAllocate() is called at application startup, and creates the global ZafCodeSetData instance zafCodeSet. CodeSetAllocate() allocates the lookup tables used to convert between ISO 8859-1 or Unicode characters and native characters. If *name* is null, the default lookup tables are used for each environment; otherwise, *name* specifies the name of the code set data object to be loaded by the data manager.

CodeSetFree `static void CodeSetFree(bool globalRequest = false);`

If *globalRequest* is true, CodeSetFree() deletes the global ZafCodeSetData instance zafCodeSet; otherwise it does nothing. CodeSetFree() is called as an application shuts down, and should normally not be called by the programmer.

CodeSetName `static const ZafIChar *CodeSetName(void);`
`static ZafError SetCodeSetName(const ZafIChar *codeSetName);`

CodeSetName() specifies the name of the code set being used to map characters. For example, a straight mapping for ISO 8859-1 would specify a code set name of "ISO8859-1", a straight mapping for Unicode would specify "UNICODE", a MS-DOS code page 437 would specify "437", and the Macintosh character set would specify "MACINTOSH". The default for this attribute is different for each environment, and may be changed by calling SetCodeSet-

Name(). The programmer should not call SetCodeSetName(), since this attribute is set internally by the ZAF libraries.

ConvertToOSChar `int ConvertToOSChar(ZafIChar zafChar, char *osChar);`

ConvertToOSChar() converts the ISO 8859-1 or Unicode character *zafChar* to the native equivalent. Systems that require multi-byte strings may return a character that is more than a single byte (char), so the return buffer *osChar* must be large enough to hold the resulting character. The number of bytes (chars) required for *osChar* is returned.

ConvertToOSString `char *ConvertToOSString(const ZafIChar *zafString, char
 *osString = ZAF_NULLP(char), bool allocate = true);`

ConvertToOSString() converts the ISO 8859-1 or Unicode string *zafString* to the native equivalent and returns the result. Systems that require multi-byte strings may return a string that requires a different size buffer than the source string, so the return buffer *osString* may be a different size than *zafString*. If *allocate* is true, ConvertToOSString() will allocate a buffer in which the resulting string will be placed, and the programmer must delete the buffer when appropriate. If *allocate* is false and *osString* is null, a static buffer will be used in which the resulting string will be placed, and the programmer should not delete the buffer.

ConvertToOSW-Char `int ConvertToOSWChar(ZafIChar zafChar, wchar_t *osChar);`

Unicode only. ConvertToOSWChar() converts the Unicode character *zafChar* to the native wide character equivalent, if supported by the environment. The return buffer *osChar* must be allocated by the programmer before calling ConvertToOSWChar(). The number of wchar_ts required for *osChar* is returned, which is always 1 according to the portable definition of wchar_t.

ConvertToOSW-String `wchar_t *ConvertToOSWString(const ZafIChar *zafString,
 wchar_t *osString = ZAF_NULLP(wchar_t), bool allocate
 = true);`

Unicode only. ConvertToOSWString() converts the Unicode string *zafString* to the native wide character string equivalent and returns the result both as the return value for the function and in *osString*. If *allocate* is true, ConvertToOSWString() will allocate a buffer in which the resulting string will be placed, and the programmer must delete the buffer when appropriate. If *allocate* is false and *osString* is null, a static buffer will be used in which the resulting string will be placed, and the programmer should not delete the buffer.

ConvertToZafChar `int ConvertToZafChar(const char *osChar, ZafIChar
 &zafChar);`

ISO only. `ConvertToZafChar()` converts the native character *osChar* to the ISO 8859-1 or Unicode equivalent, which is returned in *zafChar*. The number of bytes (chars) required for *osChar* is returned.

`int ConvertToZafChar(const wchar_t *osChar, ZafIChar
 &zafChar);`

Unicode only. This overloaded `ConvertToZafChar()` converts the native wide character *osChar* to the Unicode equivalent, which is returned in *zafChar*. The number of ZafIChars required for *zafChar* is returned, which is always 1, since ZafIChar is ZAF's portable character type no matter what mode the application is running in.

ConvertToZafString `ZafIChar *ConvertToZafString(const char *osString,
 ZafIChar *zafString = ZAF_NULLP(ZafIChar), bool
 allocate = true);`
`ZafIChar *ConvertToZafString(const wchar_t *osString,
 ZafIChar *zafString = ZAF_NULLP(ZafIChar), bool
 allocate = true);`

The first method is for ISO only. `ConvertToZafString()` converts the native string *osString* to the ISO 8859-1 or Unicode equivalent and returns the result. Systems that require multi-byte strings may return a string that requires a different size buffer than the source string, so the return buffer *zafString* may be a different size than *osString*. If *allocate* is true, `ConvertToZafString()` will allocate a buffer in which the resulting string will be placed, and the programmer must delete the buffer when appropriate. If *allocate* is false and *zafString* is null, a static buffer will be used in which the resulting string will be placed, and the programmer should not delete the buffer.

The second method is for Unicode only. This overloaded `ConvertToZafString()` converts the native wide character string *osString* to the Unicode equivalent and returns the result both as the return value for the function and in *zafString*. If *allocate* is true, `ConvertToZafString()` will allocate a buffer in which the resulting string will be placed, and the programmer must delete the buffer when appropriate. If *allocate* is false and *zafString* is null, a static buffer will be used in which the resulting string will be placed, and the programmer should not delete the buffer.

DefaultLeadByte

```
ZafUInt8 DefaultLeadByte(void) const;  
ZafUInt8 SetDefaultLeadByte(ZafUInt8 defaultLeadByte);
```

When mapping a character from ISO 8859-1 to Unicode, `DefaultLeadByte()` is used as the high byte. `DefaultLeadByte()` is analogous to the Unicode page that corresponds to a language. For example, the Unicode page for English and other similar languages is 0. `DefaultLeadByte()` defaults to 0, but the programmer may call `SetDefaultLeadByte()` to change it.

DirSepStr

```
ZafIChar dirSepStr[2];
```

`dirSepStr` is a null-terminated string that includes the character used to separate directory names in a pathname for each environment. It is initialized at application startup.

IsoToUnicode

```
ZafIChar *IsoToUnicode(const char *isoText, ZafIChar  
    *unicodeText = ZAF_NULLP(ZafIChar));
```

Unicode only. `IsoToUnicode()` converts the single-byte ISO 8859-1 string *isoText* to a double-byte Unicode string and returns the Unicode string both as the return value for the function and in *unicodeText*. Since the first 256 Unicode characters are also ISO 8859-1 characters, `IsoToUnicode()` simply pads the high byte of each resulting character with `DefaultLeadByte()`, which corresponds to the Unicode page that a language uses. If *unicodeText* is null, `IsoToUnicode()` allocates the return buffer, and the programmer must delete the buffer when appropriate.

mblen

```
int mblen(const char *hardware);
```

Unicode only. `mblen()` returns the number of bytes (chars) of the multi-byte character that *hardware* points to.

mbstowcs

```
int mbstowcs(ZafIChar *pwcs, const char *s, int n = -1);
```

Unicode only. `mbstowcs()` converts the multi-byte string *s* to the native wide character string equivalent and returns the result in *pwcs*. *pwcs* must point to a buffer sufficiently large to hold the resulting string. If *n* is less than zero, `mbstowcs()` will convert the entire string; otherwise *n* characters of the string will be converted. The number of `ZafIChars` required for *pwcs* is returned.

NativeMapping

```
bool NativeMapping(void) const;  
bool SetNativeMapping(bool mapping);
```

Unicode only. If `NativeMapping()` is true, then the native environment uses the Unicode code set internally, so no mapping is necessary; otherwise, mapping between the native code set and Unicode is necessary. This attribute may be changed using `SetNativeMapping()`, but should normally not be changed by the programmer, since it is initialized at application startup according to the native environment.

UnicodeToISO

```
char *UnicodeToIso(const ZafIChar *unicodeText, char  
                    *isoText = ZAF_NULLP(char));
```

Unicode only. `UnicodeToIso()` converts the double-byte Unicode string *unicodeText* to a single-byte ISO 8859-1 string and returns the ISO 8859-1 string both as the return value for the function and in *isoText*. The Unicode string should not include any characters beyond the first 256 Unicode characters. Since the first 256 Unicode characters are also ISO 8859-1 characters, `UnicodeToIso()` simply copies the low byte of each Unicode character into the resulting string. If *isoText* is null, `UnicodeToIso()` allocates the return buffer, and the programmer must delete the buffer when appropriate.

wcstombs

```
int wcstombs(char *s, const ZafIChar *pwcs, int n = -1);
```

Unicode only. `wcstombs()` converts the native wide character string *pwcs* to the multi-byte string equivalent and returns the result in *s*. *s* must point to a buffer sufficiently large to hold the resulting string. If *n* is less than zero, `wcstombs()` will convert the entire string; otherwise *n* characters of the string will be converted. The number of bytes (chars) required for *s* is returned.

ZafComboBox

AutoSortData	SetBackgroundColor	SetTextColor
list	SetFont	ViewOnly

Inheritance ZafComboBox : ZafWindow : (ZafWindowObject : ZafElement), ZafList

Declaration #include <z_combo.hpp>

Description The ZafComboBox object provides support for a list of items, where one item at a time is selected and visible. When the user selects the ZafComboBox object, the list of items becomes visible, and the user may select another of the items. When an item is selected, the list closes, and the ZafComboBox object displays the newly-selected item. Any item may be selected programmatically by calling SetSelected(true) for the item. In fact, if the programmer doesn't call SetSelected(true) on one of the children, the ZafComboBox object initially appears empty. See ZafWindowObject::Selected() for more information on selecting child objects.

A ZafComboBox object may be editable (see ViewOnly() for more information), in which case the user may type one of the items to select it without bringing up the list. The first list item beginning with the characters typed by the user is selected. If the ZafComboBox object is ViewOnly(), then the first item beginning with a key typed is selected.

A ZafComboBox object may contain children that display more than textual information, such as ZafButton objects with bitmap information. For the children's bitmap information to be displayed, the ZafComboBox object's OSDraw() attribute must not be true; otherwise only the text of the children will be displayed.

In initializing a newly-created ZafComboBox to select a child other than the first, child->SetSelected(true) should be called. To initialize the editable textual portion of a newly-created non-ViewOnly() ZafComboBox, comboBox->SetText() should be called.

Like other ZAF classes, ZafComboBox utilizes the native API when available, so the look-and-feel is what the end user expects. ZafComboBox by its very nature implies a relatively small number of items to choose from, since it only takes up as much space as a single item in its list. Though there is programmatically not a limit to the number of items allowable in a ZafComboBox object, no more than approximately 100 items should be placed in a single ZafComboBox object. For a large number of items, consider using ZafVtList for better usability.

Constructors All ZafComboBox constructors initialize the member variables associated with an instantiated ZafComboBox object. The default values set by the ZafComboBox 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 ZafComboBox. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafComboBox

AutoSortData()	false
ViewOnly()	false

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Modal()	false [†]
Moveable()	false [†]
NormalHotKeys()	false [†]
SelectionType()	ZAF_SINGLE_SELECTION [†]
Sizeable()	false [†]
Temporary()	false [†]

ZafWindowObject

Bordered()	true [†]
ZafElement	
ClassID()	ID_ZAF_COMBO_BOX
ClassName()	"ZafComboBox"

ZafComboBox(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, while *width* and *height* specify the width and height of the drop down list. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired.

```
ZafComboBox(const ZafComboBox &copy);
```

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

```
ZafComboBox(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 combo-box with strings follows:

```
// Create a sample window with an editable combo-box with  
    strings.  
ZafWindow *window1 = new ZafWindow(0, 0, 50, 10);  
ZafComboBox *comboBox = new ZafComboBox(0, 0, 20, 5);  
// Children are automatically positioned by the combo-box.  
comboBox->Add(new ZafString(0, 0, 20, "Black", -1));  
comboBox->Add(new ZafString(0, 0, 20, "Blue", -1));  
comboBox->Add(new ZafString(0, 0, 20, "Green", -1));  
comboBox->Add(new ZafString(0, 0, 20, "Red", -1));  
comboBox->Add(new ZafString(0, 0, 20, "White", -1));  
// Add the combo-box to the window.  
window1->Add(comboBox);
```

Destructor

```
virtual ~ZafComboBox(void);
```

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

Generally, the programmer will not directly destroy a ZafComboBox 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.

AutoSortData

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

If `AutoSortData()` is true, the combo-box will automatically sort its children as they are added to the combo-box. 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 make changes.

SetBackgroundColor

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

To provide consistency in the appearance of combo-box children, the `ZafComboBox` 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 combo-box.

SetFont

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

To provide consistency in the appearance of combo-box children, the `ZafComboBox` 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 combo-box.

list

```
ZafVtList *list;
```

The list member is used internally by ZAF to house the children of a combo-box. The programmer should never need to access this member, since it is maintained by the `ZafComboBox` class.

SetTextColor

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

To provide consistency in the appearance of combo-box children, the `ZafComboBox` 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 combo-box.

ViewOnly

```
bool ViewOnly(void) const;  
virtual bool SetViewOnly(bool viewOnly);
```

A `ViewOnly()` `ZafComboBox` object may not be edited, but may be the current object of a window, and arrow keys may be used to navigate it. If a `ViewOnly()` `ZafComboBox` object is `OSDraw()`, text may be selected and copied to the clipboard, but it may not be modified. `ViewOnly()` is false by default, but the user may call `SetViewOnly()` to make changes.

A common use of a `ZafComboBox` object with `ViewOnly()` false is to allow the end user to type a new value that is to be dynamically added as a new child when `<Enter>` or `<Return>` is typed, such as with a selection of font sizes. This behavior may be implemented by adding a new child to the `ZafComboBox` object using the `Add()` function. See `ZafWindow::Add()` for more information on adding child objects.

ZafConstraint

	Event	Object	ObjNumberID
Inheritance	ZafConstraint : ZafElement		
Declaration	#include <z_gmgr.hpp>		
Description	<p>ZafConstraint is provided solely as a base class for all geometry management constraint classes to be used with ZafGeometryManager (see ZafGeometry-Manager for more information). ZafConstraint is an abstract class, since it defines a pure virtual function. ZAF provides three derived classes of ZafConstraint: ZafAttachment, ZafDimensionConstraint, and ZafRelativeConstraint.</p> <p>Since ZafConstraint is the abstract base class for all constraint classes, the information presented in this chapter applies to all the derived classes.</p>		
Constructors	<p>All ZafConstraint constructors initialize the member variables associated with an instantiated ZafConstraint object. The default values set by the ZafConstraint and its base class constructors follow, if they differ from those set by the base class constructor.</p>		

Member Initializations

ZafConstraint

Object()	user-supplied parameter
----------	-------------------------

ZafElement

ClassID()	ID_ZAF_CONSTRAINT
ClassName()	"ZafConstraint"

ZafConstraint(ZafWindowObject *object);

This constructor is called by derived class constructors to create a ZafConstraint, attached to *object*. (See [Object\(\)](#) for more information.)

ZafConstraint(const ZafConstraint ©);

The copy constructor is used in conjunction with the overloaded Duplicate() function. It is provided solely so that derived class instances may be copied via a ZafConstraint pointer.

```
ZafConstraint(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 constraint follows:

```
// Create a status bar with geometry-managed children.
ZafStatusBar *stat = new ZafStatusBar(0, 0, 0, 1);
ZafString *string = new ZafString(0, 0, 15, new
    ZafStringData("String"));
stat->Add(string);
ZafTime *time = new ZafTime(15, 0, 15, new ZafTimeData);
stat->Add(time);

// Time field will remain at the right side of status bar.
ZafAttachment *attach = new ZafAttachment(time, ZAF_ATCF_RIGHT);
attach->SetOffset(0);

// Create the geometry manager and add the first constraint.
ZafGeometryManager *geo = new ZafGeometryManager;
geo->Add(attach);

// String field will occupy the remaining space.
attach = new ZafAttachment(string, ZAF_ATCF_RIGHT);
attach->SetStretch(true);
attach->SetReference(time);
attach->SetOppositeSide(true);
attach->SetOffset(1);

// Add the second constraint to the geometry manager.
geo->Add(attach);

// Add the geometry manager to the status bar.
stat->Add(geo);
```

Destructor

```
virtual ~ZafConstraint(void);
```

The destructor is used to free the memory associated with a [ZafConstraint](#) object. It chains to the [ZafElement](#) destructor.

Generally, the programmer will not directly destroy a [ZafConstraint](#) object, since it is automatically destroyed when its parent geometry manager is destroyed. For more information on child object deletion, see [ZafWindow::~ZafWindow\(\)](#).

Members

- Event* `virtual ZafEventType Event(const ZafEventStruct &event);`
- This overloaded function handles all events sent to the ZafConstraint object. The only event handled by ZafConstraint is S_INITIALIZE, which causes the constraint to initialize its numberID, stringID, and Object().
- Object* `ZafWindowObject *Object(void) const;`
`void SetObject(ZafWindowObject *setObject);`
- Object() is the window object to which the constraint applies. There may be many constraints that apply to a single object, but a constraint applies to exactly one window object.
- ObjNumberID* `ZafNumberID ObjNumberID(void) const;`
`void SetObjNumberID(ZafNumberID tObjNumberID);`
- SetObjNumberID() specifies to the constraint the numberID of the window object to which it applies. When reading a constraint from a persistent data file, the constraint is tied to its window object via the numberID. *tObjNumberID* is the numberID of the window object.

ZafCoordinateType

Inheritance Enumerated type

Declaration `#include <z_coord.hpp>`

Description ZafCoordinateType provides support for various coordinate systems in ZAF. ZAF_CELL is the default coordinate system based on the size of the system font on each environment. Since this system is fully portable to all environments (including non-graphical character-based environments) the programmer should use coordinate systems other than ZAF_CELL only when greater precision is needed. ZAF_MINICELL coordinates are fractional ZAF_CELLS and are portable between graphical environments only. ZAF_PIXEL coordinates are not portable. Other coordinate systems are available as well and are discussed below. Since coordinate systems depend on a display device, [ZafDisplay](#) provides conversion values and functions for converting between the different supported coordinate systems in ZAF.

The coordinate systems defined in ZAF are as follows:

Coordinate Type	Description
ZAF_CELL	Specifies the cell coordinate system, based on the system font, and the only coordinate system portable to both graphical and character-based environments.
ZAF_INCH_1000	Specifies the inch coordinate system, based on thousandths of an inch. Inches are portable between graphical environments.
ZAF_MILLIMETER_10	Specifies the millimeter coordinate system, based on tenths of a millimeter. Millimeters are portable between graphical environments.
ZAF_MINICELL	Specifies the mini-cell coordinate system, based on fractional cells calculated as specified above. Mini-cells are portable between graphical environments.
ZAF_PIXEL	Specifies the pixel coordinate system using native display values. Pixels are not portable.
ZAF_POINTS	Specifies the point coordinate system commonly used in typography. There are 72 points in an inch, but inches are not rigidly defined on a display. Points are not fully portable but may be appropriate for internal calculations requiring greater precision.

Coordinate Type	Description
ZAF_TWIPS	Specifies the twip coordinate system. There are 20 twips in a point, or 1440 twips in an inch, but inches are not rigidly defined on a display. Twips are not fully portable but may be appropriate for internal calculations requiring greater precision.

ZafCursor

BlinkRate	ImageType
Event	Poll

Inheritance	ZafCursor : ZafDevice : ZafElement
Declaration	#include <z_cursor.hpp>
Description	ZafCursor is the class that defines cursor device support. A cursor shows the end user where keyboard focus is, and usually blinks for noticability. Any keys typed will be inserted where the cursor is.
Constructors	All ZafCursor constructors initialize the member variables associated with an instantiated ZafCursor object. Default values set by the ZafCursor follow, as well as base class values when overridden by ZafCursor.

Member Initializations

ZafCursor	
BlinkRate()	environment-specific
ImageType()	user-supplied parameter
ZafDevice	
DeviceType()	E_CURSOR
ZafElement	
ClassID()	ID_ZAF_CURSOR
ClassName()	"ZafCursor"
NumberID()	ID_ZAF_CURSOR
StringID()	"ZafCursor"

ZafCursor(ZafDeviceState state = D_ON, ZafDeviceImage imageType = DC_INSERT);

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

```
ZafCursor(const ZafCursor &copy);
```

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

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

Destructor

```
virtual ~ZafCursor(void);
```

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

Generally, the programmer will not directly destroy a ZafCursor 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.

BlinkRate

```
static int BlinkRate(void);  
static int SetBlinkRate(int blinkRate);
```

BlinkRate() is a static member used by environments that don't automatically blink the cursor device. It should normally not be referred to by the programmer.

Event

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

This overloaded function handles all events sent to the ZafCursor object. ZafCursor handles all the DC_* messages discussed in ImageType(), as well as D_STATE, which causes a device to return its state.

ImageType

```
ZafDeviceImage ImageType(void) const;  
virtual ZafDeviceImage SetImageType(ZafDeviceImage  
    imageType);
```

ImageType() returns a constant that indicates the cursor's current image type. **SetImageType()** may be called to change a cursor's image type, if supported by the environment. The different image types supported by ZafCursor are as follows:

ImageType()	Description
DC_INSERT	causes the cursor to show the insert image
DC_OVERSTRIKE	causes the cursor to show the overstrike image

Poll

```
virtual void Poll(void);
```

In some environments, the **Poll()** function checks the cursor device for any input events and posts them on the event manager's queue, if the ZafCursor's state is not **D_OFF**. In other environments where cursor events are handled automatically by the native environment's event queue, **Poll()** simply blocks cursor events from coming through ZAF's event manager queue if the ZafCursor's state is **D_OFF**.

ZafData

Clear	Duplicate	Event
Destroyable	Error	GetObject

Inheritance ZafData : [ZafElement](#), [ZafNotification](#)

Declaration #include <z_data.hpp>

Description 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 an abstract class since it has some pure virtual functions and a protected constructor. Since you cannot directly instantiate a ZafData class, you must 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	Internationalization
ZafBignumData	ZafBitmapData	ZafCodeSetData
ZafDateData	ZafIconData	ZafLanguageData
ZafIntegerData	ZafMouseData	ZafLocaleData
ZafStringData	ZafScrollData	
ZafRealData		
ZafTimeData		
ZafUTimeData		

Constructors The ZafData class constructor 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.

Clear

```
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.

The following code shows how to use this method.

```
// 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.

Destroyable

```
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()`.

Duplicate

```
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();
```

Error

```
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
ZAF_ERROR_NONE	No error exists.
ZAF_ERROR_INVALID	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.
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 or too small for the return value. For example, a value of 1000 does not fit into a "%c" sprintf() style argument.
ZAF_ERROR_INVALID_TARGET	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.
ZAF_ERROR_INVALID_SOURCE	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;
```

Event

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

Event() processes all the events that come to the data object. At this level, no events are handled. Derived classes must handle events in overloaded Event() functions.

GetObject

```
virtual ZafData *GetObject(ZafNumberID numberID);
virtual ZafData *GetObject(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.

ZafDataManager

[AllocateData](#)

Inheritance	<code>ZafDataManager : ZafDataRecord : ZafData : (ZafNotification, ZafElement), ZafList</code>
Declaration	<code>#include <z_data.hpp></code>
Description	ZafDataManager cooperates with ZafDataPersistence through the zafDataPersistence global in maintaining ZafData objects to be used in an application. By calling AllocateData(), the programmer can ask the ZafDataManager to load a particular data object out of persistence, or simply return the data object if it has already been loaded.
Constructor	The ZafDataManager constructor initializes the member variables associated with an instantiated ZafDataManager object. The default values set by the ZafDataManager and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_DATA_MANAGER</code>
<code>ClassName()</code>	<code>"ZafDataManager"</code>

ZafDataManager(void);

The constructor allocates a ZafDataManager instance.

The following code snippet shows how to use the global zafDataManager:

```
// Ask the data manager for a bitmap data object
// whose stringID is "MyBitmap".
// AllocateData() will load the bitmap data object
// from persistence if necessary.
ZafBitmapData *bitmapData = DynamicPtrCast(zafDataManager-
    >AllocateData("MyBitmap", ZafBitmapData::className,
        ZafBitmapData::classID), ZafBitmapData);
```

Destructor

```
virtual ~ZafDataManager(void);
```

This virtual destructor is used to free the memory associated with an instantiated ZafDataManager object. All ZafData objects associated with the ZafDataManager are destroyed.

Members

```
virtual ZafData *AllocateData(const ZafIChar *stringID,  
                              ZafClassName className, ZafClassID classID);
```

AllocateData

AllocateData() returns a pointer to the object identified by a stringID of *stringID*. If it hasn't been loaded yet, AllocateData() asks ZafDataPersistence through the global zafDataPersistence to load the object using GetDataConstructor(). If the object cannot be found, null is returned. *className* specifies the constant string identifier for the class of the object to be loaded, and *classID* specifies the constant number identifier for the class of the object to be loaded.

ZafDataPersistence

AddDataConstructor	CurrentFile	PopLanguage
AddFileSystem	CurrentFileSystem	PushLanguage
AllocateFile	CurrentLanguage	PopLevel
ClearDataConstructors	Error	PushLevel
ClearFileSystems	FirstFileSystem	SetDataConstructors
CurrentClassID	LastFileSystem	SubtractDataConstructor
CurrentClassName	Merge	SubtractFileSystem

Inheritance

Root class

Declaration

#include <z_data.hpp>

Description

ZafDataPersistence allows objects derived from ZafData to be written to, and/or read from a storage object—usually a persistent data file.

Using ZafDataPersistence, a [ZafData](#) object can be constructed directly from information stored in the data file, or stored there for later use. A table of persistent data objects is maintained by this class and is used to reference each object’s Read() function during construction. All ZAF-supplied data classes are included in this table by default and the programmer may add support for derived classes. See [AddDataConstructor\(\)](#).

ZafData constructors (and thereby ZafDataPersistence members) are often called by ZafWindowObject constructors, or by ZafDataManager::AllocateData(). ZafDataPersistence uses [ZafFileSystem](#) and [ZafFile](#) to maintain and manipulate one or more persistent object data files. See [ZafStorage](#) for more information.

Constructors

ZafDataPersistence initializes its members to the following default values:

Member Initializations

ZafDataPersistence

CurrentLanguage()	0
Error()	ZAF_ERROR_NONE

```
ZafDataPersistence(ZafFileSystem *fileSystem,  
DataConstructor *dataConstructor);
```

The ZafDataPersistence class constructor creates a ZafDataPersistence object based on the open file *fileSystem* and the persistent data constructor table *dataConstructor*.

Below is the definition of `ZafDataPersistence::DataConstructor`.

```
struct DataConstructor
{
    ZafClassID classID;
    ZafClassName className;
    ZafDataConstructor constructor;
};
```

classID is the numeric class identification constant for the data class, *className* is the string class identification constant for the data class, and *constructor* is a pointer to the function used to construct an instance of the data class. The *className* identifier is used by `ZafDataPersistence` to locate the correct persistent constructor for a data class. The definition of `ZafDataConstructor` follows:

```
typedef ZafElement *(*ZafDataConstructor)(const ZafIChar *,
    ZafDataPersistence &);
```

```
ZafDataPersistence(const DataConstructor &copy);
```

```
ZafDataPersistence(const ZafDataPersistence &copy);
```

The copy constructors create a new `ZafDataPersistence` object and initialize its data from *copy*.

The following code snippet shows how to create a `ZafDataPersistence` object and use it to load a data object from storage:

```
// Create a storage object, which opens the data file.
ZafStorage *storage = new ZafStorage("test.znc",
    ZAF_FILE_READ);

// Create a persistence object for loading the data object.
zafDataPersistence = new ZafDataPersistence(storage,
    zafDefaultDataConstructor);

// Load the data object from the data file.
ZafBignumData *bignumData = new ZafBignumData("BignumData1",
    *zafDataPersistence);
```

Often it is useful to derive a `ZafData` class and store custom information to the persistent file. The following code snippet shows how to derive a class that stores its own data in addition to information normally stored by base classes:

```
// Define the static members of MyStringData.
```

```
ZafClassID MyStringData::classID = 3500;
ZafClassNameChar MyStringData::className[] = "MyStringData";

// Persistent constructor of MyStringData.
// PushLevel() before reading the base class information.
MyStringData::MyStringData(const ZafIChar *name,
    ZafDataPersistence &persist) : ZafStringData(name,
    persist.PushLevel(className, classID, ZAF_PERSIST_FILE))
{
    if (persist.Error() == ZAF_ERROR_NONE)
    {
        // Read the MyStringData class information.
        ZafFile *file = persist.CurrentFile();
        *file >> MyStringData::member;
    }
    // PopLevel() after the class information has been read.
    persist.PopLevel();

    // Save the error into the object if the data couldn't be read.
    if (persist.Error() != ZAF_ERROR_NONE)
        SetError(persist.Error());
}

// Define Read(), since ZafData must add it to the
// ZafDataPersistence data constructor table.
ZafElement *MyStringData::Read(const ZafIChar *name,
    ZafDataPersistence &persist)
{
    {
        return (new MyStringData(name, persist));
    }
}

// Write() stores the class information in the data file.
// Base class info is stored first followed by custom info.
void MyStringData::Write(ZafDataPersistence &persist)
{
    // Write the base class information.
    // PushLevel() before writing the base class information.
    // (AllocateFile() is called internally to prep the file.
    ZafStringData::Write(persist.PushLevel(className, classID,
        ZAF_PERSIST_FILE));

    if (persist.Error() == ZAF_ERROR_NONE)
    {
        // Write the MyStringData class information.
        ZafFile *file = persist.CurrentFile();
        *file << MyStringData::member;
    }

    // PopLevel() after writing the class information.
```



```

persist.PopLevel();

// Save the error into the object if data couldn't be written.
if (persist.Error() != ZAF_ERROR_NONE)
    SetError(persist.Error());
}

```

Destructor

```
virtual ~ZafDataPersistence(void);
```

This virtual destructor is used to free the memory associated with an instantiated ZafDataPersistence object. Generally, the programmer will not directly destroy a ZafDataPersistence object, since it is destroyed when the application is closed.

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.

AddData-Constructor

```
virtual DataConstructor *AddDataConstructor(ZafClassName
    className, ZafClassID classID, ZafDataConstructor
    constructor);
```

AddDataConstructor() adds a data class constructor to ZafDataPersistence's table of constructors. *className* is the string class identification constant for the data class, *classID* is the numeric class identification constant for the data class, and *constructor* is a pointer to the function used to construct an instance of the data class. If an entry with the same class identification constants is already present in the table, its constructor is updated to *constructor*. A pointer to the DataConstructor element in the table is returned. Internally, ZAF uses *className* for all lookups.

AddFileSystem

```
ZafFileSystem *AddFileSystem(ZafFileSystem *fileSystem,
    ZafFileSystem *position = ZAF_NULLP(ZafFileSystem));
```

AddFileSystem() adds a data file to a table (list) of data files to search when trying to load a data object. *fileSystem* specifies the data file to add to the table. If *position* is null, the data file is added to the end of the table, in effect becoming the last data file to search. If *position* is non-null, *fileSystem* is added to the table before the data file specified by *position*, causing *fileSystem* to be searched before *position*. *fileSystem* is returned.

When `ZafDataPersistence` reads data it first calls `AllocateFile()` to search for the data in each of the file systems in its list, moving from first to last. The read is successful when the first matching entry is found and read, and fails if all file systems are searched unsuccessfully.

AllocateFile

```
ZafError AllocateFile(const ZafIChar *name, ZafFileMode  
mode) ;
```

`AllocateFile()` finds the object specified by the stringID *name* in the data files that have been added to the data file table by `AddFileSystem()`. `CurrentFile()` then points to the object.

If *mode* is `ZAF_FILE_READ` and the object was not found, `Error()` is set to either `ZAF_ERROR_INVALID_ID` (meaning the object wasn't found), or `ZAF_ERROR_FILE_OPEN` (meaning the file was not open); otherwise, `Error()` is set to `ZAF_ERROR_NONE`.

If *mode* is `ZAF_FILE_READWRITE`, `ZAF_FILE_CREATE`, or `ZAF_FILE_OPENCREATE` and the object wasn't found, the file is prepared for the object to be written. The value `Error()` is returned.

ClearData-Constructors

```
virtual void ClearDataConstructors(void) ;
```

`ClearDataConstructors()` clears the table of data constructors and destroys the memory associated with the table, so that no data constructor is associated with the `ZafDataPersistence` object.

ClearFileSystems

```
virtual void ClearFileSystems(void) ;
```

`ClearFileSystems()` clears the table of file systems, so that no file is associated with the `ZafDataPersistence` object.

CurrentClassID

```
virtual ZafClassID CurrentClassID(void) ;
```

`CurrentClassID()` returns the numeric class identification constant for the object currently being accessed.

CurrentClassName

```
virtual ZafClassName CurrentClassName(void) ;
```

`CurrentClassName()` returns the string class identification constant for the object currently being accessed.

CurrentFile `ZafFile *CurrentFile(void) const;`

`CurrentFile()` returns the location in the file system where the object was found or created by a previous call to `AllocateFile()`. `CurrentFile()` is called before reading or writing an object's data in a data file.

CurrentFileSystem `ZafFileSystem *CurrentFileSystem(void) const;`

`CurrentFileSystem()` returns the file system in which the object was found or created by a previous call to `AllocateFile()`.

CurrentLanguage `const ZafIChar *CurrentLanguage(void) const;`

`CurrentLanguage()` returns the current language of the object being read or written, specified by a two-character ISO code. This language ID is appended to the file name when a language has been specified using `PushLanguage()`. (see [ZafI18nData](#) for more information).

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

`Error()` stores the last error that occurred with the ZafDataPersistence 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 ZafDataPersistence object:

Error()	Description
<code>ZAF_ERROR_NONE</code>	No error exists.
<code>ZAF_ERROR_INVALID_ID</code>	No object with the ID given was found in the table.
<code>ZAF_ERROR_FILE_OPEN</code>	The file to be searched was not open.

FirstFileSystem `ZafFileSystem *FirstFileSystem(void) const;`

`FirstFileSystem()` returns the first file system added to the search table by `AddFileSystem()`.

GetDataConstructor virtual ZafDataConstructor **GetDataConstructor**(ZafClassID
 classID, ZafClassName className = 0);

GetDataConstructor() finds and returns a data class constructor in ZafDataPersistence's table of constructors according to the class identification constants. *classID* is the numeric class identification constant for the data class and *className* is the string class identification constant for the data class. If no matching class is found, null is returned.

LastFileSystem ZafFileSystem ***LastFileSystem**(void) const;

LastFileSystem() returns the last file system added to the search table by AddFileSystem().

Merge bool **Merge**(ZafDataPersistence ©);

Merge() merges *copy* and all its data (including tables and file systems) into this ZafDataPersistence object and returns true. The merged object's tables and file systems are removed from *copy* and added to this ZafDataPersistence object, so *copy* is in effect empty after calling this function.

PopLanguage int **PopLanguage**(void);
PushLanguage int **PushLanguage**(const ZafIChar *language);

To support multilingual application data (multiple languages in a single data file), PushLanguage() must be called with the two-character ISO code that specifies the language before reading or writing the data. (See [Zaf118nData](#) for more information.)

PushLanguage() causes the object data to be stored in a different location using the *language* specifier. After the data is read or written, PopLanguage() must be called to finalize the operation.

Both functions return a zero-based index into ZafDataPersistence's internal language table. The following code snippet shows how to use these functions:

```
// Get the Spanish version of the data.
zafDataPersistence->PushLanguage("es");

// Load Spanish messages from data file.
ZafLanguage esMessages("MyMessages", *zafDataPersistence);

// Finalize the read operation.
zafDataPersistence->PopLanguage();
```

```

PopLevel      ZafDataPersistence &PopLevel(void);
PushLevel     ZafDataPersistence &PushLevel(ZafClassName className,
                                             ZafClassID classID, ZafPersistEntryType type);

```

Since each class may write its own data and chain to other objects to write data pieces, these methods maintain a stack to facilitate the opening and closing of the object in the file system. Each time a derived class calls its base class to read or write information, it calls `PushLevel()`. The root class calls `AllocateFile()` to find the object and open it for reading or writing. Then after each class reads or writes its information, it calls `PopLevel()`. The last `PopLevel()` causes the object to be closed in the file system.

Using `PushLevel()` and `PopLevel()` the programmer can preserve a file system state and return to it later. For example, one object may cause another object to be written into a different directory and file (a change in file system state). To ensure that the file system state is not destroyed by the secondary write, `PushLevel()` and `PopLevel()` are called before and after the secondary operation.

className specifies the string class identification constant for the class. Only the derived-most class name is important when reading and writing the object. *classID* specifies the numeric class identification constant for the class, and only the derived-most class ID is important when reading and writing the object. *type* specifies where to look for the object in the file system. The possible values of *type* are as follows:

ZafPersistEntryType	Description
ZAF_PERSIST_DATA	Directs <code>AllocateFile()</code> to look for the object in the data section of the file system
ZAF_PERSIST_DIRECTORY	Directs <code>AllocateFile()</code> to look for the object in the file system's directory specified by the persistent constructor's <i>name</i> parameter
ZAF_PERSIST_FILE	Directs <code>AllocateFile()</code> to look for the object in the current directory of the file system
ZAF_PERSIST_ROOT_DIRECTORY	Directs <code>AllocateFile()</code> to look for the object in the <i>className</i> directory in the root of the file system
ZAF_PERSIST_ROOT_FILE	Directs <code>AllocateFile()</code> to look for the object in the root directory of the file system

SetData-Constructors

```
virtual bool SetDataConstructors(DataConstructor  
    *dataConstructor);
```

SetDataConstructors() clears ZafDataPersistence's table of constructors, then creates a new table of constructors based on the pre-built *dataConstructor* table. SetDataConstructors() is an internal routine, and is normally not called by the programmer.

SubtractData-Constructor

```
virtual bool SubtractDataConstructor(ZafClassID classID,  
    ZafClassName className = 0);
```

SubtractDataConstructor() removes a data class constructor from ZafDataPersistence's table of constructors. *classID* is the numeric class identification constant for the data class and *className* is the string class identification constant for the data class. If the entry was successfully removed, true is returned; otherwise, false is returned.

SubtractFileSystem

```
ZafFileSystem *SubtractFileSystem(ZafFileSystem  
    *fileSystem);
```

SubtractFileSystem() removes a data file from the table of data files to search when trying to load a data object. *fileSystem* specifies the data file to remove from the table. If an error occurs, null is returned; otherwise a pointer to *fileSystem* is returned.

ZafDataRecord

	<div>Clear</div> <div>GetObject</div>
Inheritance	<div>ZafDataRecord : ZafData : (ZafNotification, ZafElement), ZafList</div>
Declaration	<div>#include <z_data.hpp></div>
Description	<div>ZafDataRecord is provided strictly as a base class for ZafDataManager. ZafDataRecord provides service routines used by ZafDataManager to maintain its list of ZafData objects. Please refer to ZafDataManager for more information.</div>
Constructors	<div>The ZafDataRecord constructors initialize the member variables associated with an instantiated ZafDataRecord object. The default values set by the ZafDataRecord and its base class constructors follow, if they differ from those set by the base class constructor.</div> <div><div>Member Initializations</div><div><div>ZafElement</div><div>ClassID()ID_ZAF_DATA_RECORD</div><div>ClassName()"ZafDataRecord"</div></div></div>
	<div><div>ZafDataRecord(void);</div><div>This constructor allocates a ZafDataRecord instance.</div></div> <div><div>ZafDataRecord(const ZafIChar *name, ZafDataPersistence &persist);</div><div>This constructor is used for persistence. Refer to ZafWindow for more information, since most persistence is done at the ZafWindow level.</div></div>
Destructor	<div><div>virtual ~ZafDataRecord(void);</div><div>This virtual destructor is used to free the memory associated with an instantiated ZafDataRecord object. All ZafData objects associated with the ZafDataRecord are destroyed.</div></div>

Members

Clear `virtual void Clear(void);`

`Clear()` provides a hook for clearing the data associated with the `ZafDataRecord` object. This is an advanced routine, and should not be used by the programmer.

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

These functions return a pointer to the `ZafData` object in the `ZafDataRecord` object's list with either the identifier *numberID* or the identifier *stringID*. If a data object cannot be found with the identifier provided, null is returned. See [ZafElement](#) for more information on these identifiers.

ZafDate

[DateData](#) [Event](#)

Inheritance ZafDate : [ZafString](#) : [ZafWindowObject](#) : [ZafElement](#)

Declaration `#include <z_date1.hpp>`

Description 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. See [ZafString::AllowInvalid\(\)](#) and [ZafString::ReportInvalid\(\)](#) for information on these attributes and how they affect validation for this class.

All ZafDate objects refer to data contained in a [ZafDateData](#) object (refer to this class for additional essential information). ZafDate includes “year 2000 compliance,” meaning that through the underlying ZafDateData object, a ZafDate object keeps track of the exact year, rather than just the last two digits.

Formats ZafString and derived classes parse input and display output based on default or programmer-defined input and output formats. Programmers may use the [SetInputFormat\(\)](#) and [SetOutputFormat\(\)](#) families of functions to change the default format. The functions are documented in the [ZafString](#) reference.

Each class derived from ZafFormatData handles a different set of formatting arguments, in addition to those supported by its base classes. ZafDateData (and therefore ZafDate) handles the following “date” subset of the arguments supported by ZafUTimeData (the parent class to ZafDateData):

Format Argument	Substitution
%%	'%' character
%a	Locale-specific abbreviated weekday name
%A	Locale-specific full weekday name
%b	Locale-specific abbreviated month name
%B	Locale-specific full month name
%C	Century number (the year divided by 100 and truncated to an integer) as a decimal number [00-99]
%d	Day of month as a decimal number [01, 31]
%D	Same as %m/%d/%y
%e	Day of the month as a decimal number [1, 31]; a single digit is preceded by a space

Format Argument	Substitution
%EC	Number of the base year (period) in the locale's alternative representation
%Ey	The offset from %EC (year only) in the locale's alternative representation
%EY	The full alternative year representation
%g	Month as an abbreviated month name
%G	Day as an abbreviated day name
%j	Day of the year as a decimal number [01, 366]
%m	Month as a decimal number [01, 12]
%M	Minute as a decimal number [00, 59]
%u	Weekday as a decimal number [1, 7], with 1 representing Monday
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00, 53]
%v	Month number as a decimal [1, 12]
%V	Week number of the year (Monday as the first day of the week) as a decimal number [01, 53]; if the week containing 1 January has four or more days in the new year then it is considered week 1; otherwise, it is week 53 of the previous year, and the next week is week 1
%w	Replaced by the weekday as a decimal number [0, 6], with 0 representing Sunday
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00, 53]; all days in a new year preceding the first Sunday are considered to be in week 0
%x	Locale-specific date representation
%y	Year without century as a decimal number [00, 99]
%Y	Year with century as a decimal number

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 con-

structor, 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 the ZafDate object is 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 that 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 ZafDateData 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 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

DateData

```
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](#). `SetDate-`

Data() may be called to change the date data object used by the ZafDate object. SetDateData() will delete the previous DateData() object if it is Destroyable() and no other object uses it.

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 [Constructors](#) code snippet for an example using ZafDateData objects with ZafDate.

Event

```
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
N_RESET_I18N	causes the object to redisplay its data according to the new internationalization values
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

	Clear	FormattedText	SetDate
Inheritance	ZafDateData : ZafUTimeData : ZafFormatData : ZafData : ZafElement, ZafNotification		
Declaration	#include <z_date.hpp>		
Description	<p>ZafDateData combines date encapsulation with data notification and object notification from ZafData. It is most often used in conjunction with the ZafDate user interface object but may be used as a stand-alone object if desired. Refer to the ZafUTimeData documentation for a discussion of member methods, inherited by ZafDateData, used to retrieve and set date-specific information (e.g., day, month, year, etc.).</p> <p>All ZafData objects may make use of printf-style formatting and parsing arguments during string operations. In addition, all ZafUTimeData objects may make use of strftime- and strptime-style formatting and parsing arguments during string operations. Refer to standard library documentation for detailed information on printf, strftime, and strptime functions as well as their corresponding conversion characters.</p>		
Constructors	ZafDateData constructors initialize the member variables associated with a new ZafDateData object and allocate space to hold the date data. The default values set by ZafDateData follow, if they are overridden from those set by base class constructors:		

Member Initializations

ZafUTimeData

BasisYear()	(varies by constructor)
Day()	(varies by constructor)
DayName()	(varies by constructor)
DayOfWeek()	(varies by constructor)
DaysInMonth()	(varies by constructor)
DaysInYear()	(varies by constructor)
LeapYear()	(varies by constructor)
Month()	(varies by constructor)
MonthName()	(varies by constructor)
Value()	(varies by constructor)
Year()	(varies by constructor)

Member Initializations

ZafElement

ClassID()	ID_ZAF_DATE_DATA
ClassName()	"ZafDateData"

ZafDateData(void);

The basic constructor allocates a ZafDateData instance and initializes its value to the current system date.

ZafDateData(int year, int month, int day);

This constructor allocates a ZafDateData instance and initializes its contents to the date corresponding to *year*, *month* and *day*.

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

This constructor allocates a ZafDateData instance and initializes its contents to the date equivalent of *string*. The conversion uses the strptime-style specifier format to interpret the string. If format is null ZafDateData uses its locale-specific default format.

ZafDateData(const ZafDateData ©);

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

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

This constructor is the persistent constructor. It allocates a new ZafDateData 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 ZafDateData creation techniques.
ZafDateData date1(1997, 3, 20);
ZafDateData copyDate = date1;
ZafDateData date2("1997 2 14", "%y %m %d");
ZafDateData systemDate;
```

Destructor

```
virtual ~ZafDateData(void);
```

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

Members*Clear*

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

Clear() sets the value of a ZafDateData object to the current system date.

FormattedText

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

FormattedText() fills *buffer* with a string representation of the ZafDateData using the strftime-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* characters. FormattedText() returns the integer value it receives from its call to strftime().

```
// Show various results of FormattedText().  
ZafIChar buffer[256];  
  
ZafDateData date(1997, 7, 4);  
date.FormattedText(buffer, 256, "%Y/%m/%d");  
printf("date - %s\n", buffer);  
  
date.FormattedText(buffer, 256, "%B%e, %Y");  
printf("date - %s\n", buffer);  
  
date.FormattedText(buffer, 256, "%A,%e %B %Y");  
printf("date - %s\n", buffer);  
  
=====  
date - 97/07/04  
date - July 4, 1997  
date - Friday, 4 July 1997
```

SetDate

```
virtual ZafError SetDate(int year, int month, int day);  
virtual ZafError SetDate(const ZafIChar *buffer, const  
    ZafIChar *format);  
virtual ZafError SetDate(const ZafDateData &number);
```

SetDate() functions set the value of the ZafDateData object from numeric input, another date, or an interpreted string. Refer to [FormattedText\(\)](#) for more information on date/string conversions.

ZafDevice

display	Event	Poll
DeviceState	Installed	Previous
DeviceType	Next	

Inheritance	ZafDevice : ZafElement
Declaration	#include <z_device.hpp>
Description	<p>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.</p> <p>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.</p> <p>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.</p>
Constructor	<p>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.</p>

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.

display

```
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.

DeviceState

```
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 cur-

rently 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);
```

DeviceType

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.

Event

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.

Installed

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.

Next

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

Previous

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;
```

Poll

```
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().

ZafDialogWindow

Control

Inheritance	ZafDialogWindow : ZafWindow : (ZafWindowObject : ZafElement), ZafList
Declaration	#include <z_dlgwin.hpp>
Description	ZafDialogWindow supports the native look and feel of a dialog window. Usually, this includes a border or title bar appearing different than those of a regular window. A dialog window is not sizeable by nature, since its border indicates a dialog window rather than sizability and since a dialog's contents should promote understandability. A dialog window is also by its nature not a temporary window, since the end user is expected to dismiss the dialog after interacting with it.
Constructors	All ZafDialogWindow constructors initialize the member variables associated with an instantiated ZafDialogWindow object. The default values set by the ZafDialogWindow 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 ZafDialogWindow. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafWindow

Destroyable()	false
Modal()	true
Sizeable()	false [†]
Temporary()	false [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	false [†]
Disabled()	false [†]
Noncurrent()	false [†]
ParentPalette()	false [†]
RegionType()	ZAF_INSIDE_REGION [†]

ZafElement

Member Initializations

<code>ClassID()</code>	<code>ID_ZAF_DIALOG_WINDOW</code>
<code>ClassName()</code>	<code>"ZafDialogWindow"</code>

ZafDialogWindow(int left, int top, int width, int height);

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 the window manager. *width* and *height* specify the width and height of the client region of the object. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired.

ZafDialogWindow(const ZafDialogWindow ©);

The copy constructor creates a new **ZafDialogWindow** object and initializes its data from *copy*.

ZafDialogWindow(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 dialog window follows:

```
// Create a modal dialog window.
ZafDialogWindow *window = new ZafDialogWindow(1, 1, 60, 16);
window->AddGenericObjects(new ZafStringData("Modal Dialog"));
window->SetModal(true);

// Add a message to the dialog.
window->Add(new ZafString(18, 1, 24, "Continue?", -1));

// Add OK and Cancel buttons to cause Control() to return.
ZafButton *ok = new ZafButton(18, 4, 10, 1, "OK");
ok->SetSendMessageWhenSelected(true);
ok->SetValue(S_DLG_OK);
window->Add(ok);
ZafButton *cancel = new ZafButton(32, 4, 10, 1, "Cancel");
cancel->SetSendMessageWhenSelected(true);
cancel->SetValue(S_DLG_CANCEL);
window->Add(cancel);
```

```
// Add the dialog to the window manager.
// Use Control() only if Modal() is true.
// Control() returns only when the user has made a selection.
if (window->Control() != S_DLG_OK)
    break;
```

Destructor

```
virtual ~ZafDialogWindow(void);
```

The destructor is used to free the memory associated with a ZafDialogWindow object. It chains to the ZafWindow, ZafWindowObject, ZafList, and ZafElement destructors. For more information on child object deletion, see [ZafWindow::~ZafWindow\(\)](#).

Members

Control

```
ZafDialogEvent Control(ZafQFlags flags = Q_NORMAL);
```

Control() causes the dialog to appear on the screen. Control() doesn't return control to the caller until the end user has made a selection, after which the dialog is removed from the screen. When any button with a Value() between S_DIALOG_FIRST and S_DIALOG_LAST is selected, Control() will return. The following such values are predefined in ZAF:

- S_DLG_OK
- S_DLG_CANCEL
- S_DLG_YES
- S_DLG_NO
- S_DLG_ABORT
- S_DLG_RETRY
- S_DLG_IGNORE.

When one these event messages is handled in a derived class Event() method, the programmer must return the value from the derived Event() method to cause the dialog to close.

Control() should only be called if Destroyable() is false and Modal() is true; otherwise, simply add the dialog to the window manager via the Add() function or the + operator. If Control() is called on a dialog when Destroyable() is true, S_ERROR is returned.

ZafDimensionConstraint

Event	Minimum
Maximum	Type

Inheritance	ZafDimensionConstraint : ZafConstraint : ZafElement
Declaration	#include <z_gmgr.hpp>
Description	ZafDimensionConstraint allows a window object to be constrained to a maximum or minimum height or width (see Maximum() and Minimum() for more information). ZafDimensionConstraint objects must be added to the ZafGeometryManager object that has been added to the managed object's parent (see ZafGeometryManager for more information).
Constructors	All ZafDimensionConstraint constructors initialize the member variables associated with an instantiated ZafDimensionConstraint object. The default values set by the ZafDimensionConstraint and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafDimensionConstraint	
Maximum()	0
Minimum()	0
Type()	user-supplied parameter
ZafElement	
ClassID()	ID_ZAF_DIMENSION_CONSTRAINT
ClassName()	"ZafDimensionConstraint"

```
ZafDimensionConstraint( ZafWindowObject *object,
                        ZafDimensionConstraintType type);
```

This constructor is useful in straight-code situations to create a ZafDimensionConstraint object. *object* specifies the window object the constraint applies to, and *type* specifies the type of dimension constraint. See [ZafConstraint::Object\(\)](#) and [ZafDimensionConstraint::Type\(\)](#) for more information.

```
ZafDimensionConstraint(const ZafDimensionConstraint
                        &copy);
```


The copy constructor creates a new `ZafDimensionConstraint` object and initializes its data from *copy*.

```
ZafDimensionConstraint(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 dimension constraint follows:

```
// Create a status bar with geometry-managed children.
ZafStatusBar *stat = new ZafStatusBar(0, 0, 0, 1);
ZafString *string = new ZafString(0, 0, 15, new
    ZafStringData("String"));
stat->Add(string);
ZafTime *time = new ZafTime(15, 0, 15, new ZafTimeData);
stat->Add(time);

// Time field will remain at the right side.
ZafAttachment *attach = new ZafAttachment(time, ZAF_ATCF_RIGHT);
attach->SetOffset(0);

// Create geometry manager and add the first constraint.
ZafGeometryManager *geo->Add(attach);

// The string field will occupy
// the remaining space of the status bar.
attach = new ZafAttachment(string, ZAF_ATCF_RIGHT);
attach->SetStretch(true);
attach->SetReference(time);
attach->SetOppositeSide(true);
attach->SetOffset(1);

// Add the second constraint to the geometry manager.
geo->Add(attach);

// The string field will never be
// smaller than 5 cells wide.
ZafDimensionConstraint *dim = new ZafDimensionConstraint(string,
    ZAF_DNCF_WIDTH);
dim->SetMinimum(5);

// Add the third constraint to the geometry manager.
geo->Add(dim);

// Add the geometry manager to the status bar.
```

```
stat->Add(geo);
```

Destructor

```
virtual ~ZafDimensionConstraint(void);
```

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

Generally, the programmer will not directly destroy a ZafDimensionConstraint object, since it is automatically destroyed when its parent geometry manager 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.

Event

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

This overloaded function handles all events sent to the ZafDimensionConstraint object. The events handled by ZafDimensionConstraint are as follows:

ZafEventType	Description
S_COMPUTE_SIZE	causes the constraint to compute and modify the size of its window object
S_INITIALIZE	causes the constraint to initialize its numberID, stringID, and Object()

Maximum

```
int Maximum(void) const;  
int SetMaximum(int hzMaximum);
```

Maximum() specifies the maximum size Object() may be, and is specified in the same coordinate type as Object()->Region(). For example, if Type() is ZAF_DNCF_HEIGHT and Maximum() is 20, then the maximum height of Object() is 20. If Maximum() is 0, this constraint does not impose a maximum size. This attribute defaults to 0, but the programmer may change it with SetMaximum().

Minimum

```
int Minimum(void) const;  
int SetMinimum(int hzMinimum);
```

Minimum() specifies the minimum size Object() may be, and is specified in the same coordinate type as Object()->Region(). For example, if Type() is ZAF_DNCF_WIDTH and Minimum() is 5, then the minimum width of Object() is 5. If Minimum() is 0, this constraint does not impose a minimum size. This attribute defaults to 0, but the programmer may change it with SetMinimum().

Type

```
ZafDimensionConstraintType Type(void) const;
ZafDimensionConstraintType
    SetType(ZafDimensionConstraintType type);
```

Type() is the dimension constraint's type, which specifies the dimension (height or width) of Object() used when calculating its maximum and minimum size. The programmer may use SetType() to change this attribute. The possible values for Type() follow:

Type()	Description
ZAF_DNCF_HEIGHT	causes the constraint to affect the height of Object()
ZAF_DNCF_WIDTH	causes the constraint to affect the width of Object()

ZafDiskFile

fileCreator	ReadData	WriteData
fileType	Seek	
Length	Tell	

Inheritance ZafDiskFile : [ZafFile](#) : [ZafElement](#)

Declaration `#include <z_dskfil.hpp>`

Description ZafDiskFile and ZafDiskFileSystem provide support for file access on a disk drive, and may be used portably on any environment. ZafDiskFile encapsulates information for a single file on disk. See [ZafDiskFileSystem](#) for more information.

Constructor `ZafDiskFile(const ZafIChar *pathName, ZafFileMode mode);`

This constructor initializes the members associated with a ZafDiskFile object, and chains to the ZafFile and ZafElement constructors. *pathName* specifies the file to be opened, and may be a full pathname, partial pathname, or just the name of the file if it is in the current directory. *mode* specifies the mode in which the file is opened (see the [ZafFile](#) constructor for more information).

Destructor `virtual ~ZafDiskFile(void);`

This virtual destructor is used to free the memory associated with an instantiated ZafDiskFile object and chains to the ZafFile and ZafElement destructors.

Members

fileCreator `static OSType fileCreator;`
fileType `static OSType fileType;`

Used only on the Macintosh, these public static members provide a way for the programmer to specify the Macintosh file creator and file type of a disk file created by ZafDiskFile. *fileCreator* defaults to 'Anon' and *fileType* defaults to 'TEXT', but the programmer may set them to any desired value.

Length `virtual ZafOffset Length(void) const;`

Length() returns the length of the disk file, or it returns -1 if an error occurs.

ReadData

```
virtual int ReadData(void *buffer, int size);
```

ReadData() reads data from the disk file. The data is read into *buffer*, which must have already been allocated by the programmer, and *size* specifies the number of bytes to read. On success, ReadData() returns the number of bytes read into *buffer*; otherwise zero is returned. Error() is also set to an appropriate value.

Seek

```
virtual int Seek(ZafOffset offset, ZafSeek location);
```

Seek() moves the file pointer of the disk file. *location* specifies the position in the file from where *offset* specifies. Possible values of *location* and what they mean follow:

ZafSeek	ZafOffset
ZAF_SEEK_START	the offset is measured from the beginning of the file toward the end of the file
ZAF_SEEK_CURRENT	a positive offset is measured from the current file pointer, and a negative offset is measured toward the beginning of the file.
ZAF_SEEK_END	the offset is measured from the end of the file toward the beginning of the file

Seek() returns 0 if successful, and -1 if an error occurs (in which case Error() is also set appropriately).

Tell

```
virtual ZafOffset Tell(void) const;
```

Tell() returns the current offset of the file pointer in the disk file, or it returns -1 if an error occurs.

WriteData

```
virtual int WriteData(const void *buffer, int size);
```

WriteData() writes data to a derived file object. *buffer* is a pointer to the data to be written, and *size* specifies the number of bytes to write. On success, WriteData() returns the number of bytes written; otherwise zero is returned. Error() is also set to an appropriate value.

ZafDiskFileSystem

ChangeExtension	GetCWD	RmDir
ChDir	GetDriveNames	SetDrive
Close	MakeFullPath	StripFullPath
DeleteDriveNames	MkDir	TempName
FindClose	Open	ValidName
FindFirst	Remove	
FindNext	Rename	

Inheritance	ZafDiskFileSystem : ZafFileSystem : ZafElement
Declaration	#include <z_dskfil.hpp>
Description	ZafDiskFileSystem and ZafDiskFile provide support for file access on a disk drive, and may be used portably on any environment. ZafDiskFileSystem provides support for accessing the individual ZafDiskFiles. See ZafDiskFile for more information.
Constructor	<div>ZafDiskFileSystem(void);</div> <div>This constructor initializes the members associated with a ZafDiskFileSystem object, and chains to the ZafFileSystem and ZafElement constructors.</div>
Destructor	<div>virtual ~ZafDiskFileSystem(void);</div> <div>This virtual destructor is used to free the memory associated with an instantiated ZafDiskFileSystem object and chains to the ZafFileSystem and ZafElement destructors.</div>
Members	<div>static void ChangeExtension(ZafIChar *name, const ZafIChar *newExtension);</div> <div><i>ChangeExtension</i> ChangeExtension() changes the file extension (the part of the name after the period, if there is one) of the file on disk specified by <i>name</i> to <i>newExtension</i>.</div> <div><i>ChDir</i> virtual int ChDir(const ZafIChar *newPath, ZafStringID stringID = ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0);</div> <div>ChDir() changes the current directory on disk to <i>newPath</i>. <i>stringID</i> and <i>numberID</i> are ignored in this class since they are used only by Zinc file systems (such as ZafStorage), but are included in the prototype in order to provide the correct method signature inherited from ZafFileSystem::ChDir(). ChDir()</div>

returns 0 on success, and -1 on failure. `Error()` is also set to an appropriate value.

Close

```
virtual void Close(ZafFile *file);
```

`Close()` closes file, which is a file previously opened with `Open()`.

DeleteDriveNames

```
void DeleteDriveNames(ZafIChar *driveName[]);
```

`DeleteDriveNames()` deletes the array of drive names created by `GetDriveNames()`.

FindClose

```
virtual int FindClose(ZafFileInfoStruct &fileInfo);
```

`FindClose()` finalizes a find operation begun with `FindFirst()`. *fileInfo* is the same `ZafFileInfoStruct` object used by `FindFirst()` and `FindNext()`, and memory allocated in *fileInfo* is deleted by `FindClose()`. `FindClose()` returns 0 if the operation was successful; otherwise it returns -1.

FindFirst

```
virtual int FindFirst(ZafFileInfoStruct &fileInfo, const  
    ZafIChar *searchName, ZafStringID stringID =  
    ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0);
```

`FindFirst()` initializes a find operation. `FindFirst()` searches for a file with the search pattern specified by *searchName*. *searchName* may contain wildcards appropriate to the system. *fileInfo* is a `ZafFileInfoStruct` object allocated by the programmer, and *fileInfo* is initialized by `FindFirst()`. *stringID* and *numberID* are ignored. `FindFirst()` returns 0 if a file was found; otherwise it returns -1.

FindNext

```
virtual int FindNext(ZafFileInfoStruct &fileInfo, const  
    ZafIChar *searchName, ZafStringID stringID =  
    ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0);
```

`FindNext()` continues a find operation initiated by `FindFirst()`. `FindNext()` searches for the next file with the same *searchName* as was specified in the call to `FindFirst()`. *fileInfo* is the same `ZafFileInfoStruct` object passed to `FindFirst()`. *stringID* and *numberID* are ignored. `FindNext()` returns 0 if a file was found; otherwise it returns -1.

GetCWD

```
virtual int GetCWD(ZafIChar *pathName, int pathLength);
```

GetCWD() copies the path name of the current directory of the disk file system (without a terminating path separating character) into the buffer specified by *pathName*. *pathName* must be allocated by the programmer, and *pathLength* specifies the number of ZafIChar characters in *pathName*. GetCWD() returns 0 on success, and -1 on failure. Error() is also set to an appropriate value.

GetDriveNames

```
int GetDriveNames(ZafIChar **driveName[]);
```

GetDriveNames() finds all the drives mounted on the system and allocates an array where their names are copied. The array is returned in *driveName*. *driveName* must be passed to DeleteDriveNames() to delete the memory when it is no longer needed. GetDriveNames() returns -1 if no drive names were found (and *driveName* is null); otherwise, the number of drive names found is returned.

MakeFullPath

```
static void MakeFullPath(ZafIChar *fullPath, const  
    ZafIChar *pathName, const ZafIChar *fileName, const  
    ZafIChar *extension = ZAF_NULLP(ZafIChar));
```

MakeFullPath() creates a full pathname for the file specified in the parameters and returns the full pathname in *fullPath*. *pathName* specifies the directory, *fileName* specifies the name of the file, and *extension* specifies the file name extension (the characters after the dot).

MkDir

```
virtual int MkDir(const ZafIChar *pathName, ZafStringID  
    stringID = ZAF_NULLP(ZafIChar), ZafNumberID numberID =  
    0);
```

MkDir() creates a new directory on disk. *pathName* specifies the path name of the directory to be created. *stringID* and *numberID* are ignored. MkDir() returns 0 on success, and -1 on failure. Error() is also set to an appropriate value.

Open

```
virtual ZafFile *Open(const ZafIChar *fileName, const  
    ZafFileMode mode, ZafStringID stringID =  
    ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0);
```

Open() opens the disk file specified by *fileName*. *mode* specifies the mode the file is opened with (see the [ZafFile](#) constructor for more information). *stringID* and *numberID* are ignored. A pointer to the file opened is returned.

Remove

```
virtual int Remove(const ZafIChar *name);
```


Remove() deletes the file on disk specified by *name*. Remove() returns 0 on success, and -1 on failure. Error() is also set to an appropriate value.

Rename

```
virtual int Rename(const ZafIChar *oldName, const
    ZafIChar *newName, ZafStringID stringID =
    ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0);
```

Rename() renames the disk file or directory specified by *oldName* to *newName*. *stringID* and *numberID* are ignored. Rename() returns 0 on success, and -1 on failure. Error() is also set to an appropriate value.

RmDir

```
virtual int RmDir(const ZafIChar *pathName, bool
    deleteContents = false);
```

RmDir() deletes the directory specified by *pathName*. If *deleteContents* is false, RmDir() will only delete the directory if it is empty; otherwise, RmDir() will delete the contents of the directory along with the directory itself. RmDir() returns 0 on success, and -1 on failure. Error() is also set to an appropriate value.

SetDrive

```
int SetDrive(const ZafIChar *driveName);
```

On systems that support it, SetDrive() sets the current drive to the drive whose name is passed in *driveName*. SetDrive() returns 0 on success, and -1 on failure.

StripFullPath

```
static void StripFullPath(const ZafIChar *fullPath,
    ZafIChar *pathName = ZAF_NULLP(ZafIChar), ZafIChar
    *fileName = ZAF_NULLP(ZafIChar), ZafIChar *objectName
    = ZAF_NULLP(ZafIChar), ZafIChar *objectPathName =
    ZAF_NULLP(ZafIChar));
```

StripFullPath() does the opposite of MakeFullPath(). StripFullPath() returns the pieces of a full pathname specified by *fullPath*. The directory is returned in *pathName* and the name of the file is returned in *fileName*. If the file is a ZAF storage file and it specifies an object's ZafStorageFile path, the object's name is returned in *objectName* and the object's storage pathname is returned in *objectPathName*. If any of the return parameters is null, that piece is not returned.

TempName

```
static void TempName(ZafIChar *tempname);
```

TempName() returns in *tempname* a unique file name generated by the system that may be used to create a temporary file. If *tempname* contains a path name

on entry, the returned file name is unique for that path and the full path name for the file is returned; otherwise the file name returned will be unique for the current directory, and the returned file name is returned without any path information. *tempname* must be allocated by the programmer before calling TempName().

ValidName

```
static bool ValidName(const ZafIChar *name, bool create =  
    false);
```

ValidName() returns true if *name* specifies the name of an existing disk file. If the file doesn't exist and *create* is true, ValidName() tries to create the file and returns true if it was successful. If ValidName() creates the file, it also deletes it. Otherwise, false is returned.

ZafDisplay

AddColor	DestroyZafBitmap	Mode
AddFont	DestroyZafIcon	MonoBackground
Background	DestroyZafMouse	MonoForeground
BeginDraw	DisplayContext	Mouse
Bitmap	DisplayMode	Origin
CellHeight	DisplayType	Palette
CellWidth	DrawContext	Pixel
ClipRegion	Ellipse	pixelsPerInchX
ColorInfo	EndDraw	pixelsPerInchY
columns	FillPattern	Polygon
ConvertToOSBitmap	Font	preSpace
ConvertToOSIcon	FontInfo	postSpace
ConvertToOSMouse	Foreground	Rectangle
ConvertToZafBitmap	Icon	RectangleXORDiff
ConvertToZafIcon	InitializeOSBitmap	RegionCopy
ConvertToZafMouse	InitializeOSIcon	ResetOSBitmap
ConvertXValue	InitializeOSMouse	ResetOSIcon
ConvertYValue	Line	ResetOSMouse
CoordinateType	lines	RestoreDisplayContext
DestroyColor	LineStyle	RestoreDrawContext
DestroyFont	miniDenominatorX	Scale
DestroyOSBitmap	miniDenominatorY	SetCellSize
DestroyOSIcon	miniNumeratorX	Text
DestroyOSMouse	miniNumeratorY	TextSize

Inheritance

Root class

Declaration

```
#include <z_dsp.hpp>
```

Description

ZafDisplay is the abstract class that defines the basic functionality necessary to interface with a display device such as a screen or a printer. Many of the methods defined in ZafDisplay are pure virtual, and must be defined in derived classes such as ZafScreenDisplay and ZafPrinter.

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. In specifying coordinates at which a drawing operation is to occur, it is useful to utilize the ZafCoordinate typedef for greater precision. The ZafCoordinate typedef is as follows:

```
typedef long ZafCoordinate;
```

ZafDisplay is important to displayable ZAF classes because it provides the interface necessary to interact with the screen. Since ZafDisplay is an abstract

class, the programmer may not instantiate a ZafDisplay object. A ZafScreenDisplay object is instantiated by the ZafApplication class. When needed, the method ZafWindowObject::Display() should be used to access the ZafScreenDisplay object instantiated automatically by the ZafApplication class.

Constructor

The ZafDisplay constructor initializes the member variables associated with an instantiated ZafDisplay object. The default values set by the ZafDisplay constructor follow, but these members may be initialized by derived class constructors.

Member Initializations

ZafDisplay

CellHeight()	font-dependent
CellWidth()	font-dependent
columns	display-dependent
CoordinateType()	display-dependent
DisplayMode()	ZAF_DISPLAY_COLOR
DisplayType()	"ZafDisplay"
lines	display-dependent
miniDenominatorX	10
miniDenominatorY	10
miniNumeratorX	1
miniNumeratorY	1
Mode()	ZAF_MODE_COPY
Origin()	(0, 0)
pixelsPerInchX	display-dependent
pixelsPerInchY	display-dependent
postSpace	environment-specific
preSpace	environment-specific
Scale()	1/1

ZafDisplay(void);

The constructor is useful in straight-code situations. This constructor should not be called by the programmer, but will be called by the derived constructor.

Destructor

virtual ~**ZafDisplay**(void);

The destructor is used to free the memory associated with a ZafDisplay object. The destructor is called by the derived destructor.

Members*AddColor*

```
virtual ZafLogicalColor AddColor(ZafLogicalColor index,  
    ZafUInt8 red, ZafUInt8 green, ZafUInt8 blue) = 0;
```

DestroyColor

```
virtual ZafError DestroyColor(ZafLogicalColor color) = 0;
```

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()`. *index* specifies the zero-based index (or logical color) into the ZAF internal color table where the color is to be inserted (each display object has a unique color table). The 16 colors preloaded by ZAF occupy indices 0-15. Zinc also reserves indices 16-31 for internal use. For convenience, the `ZAF_CLR_USER_FIRST` constant is defined to be 32, indicating the first index not reserved by Zinc. The programmer may add colors with indices 32-126. Indices of 0-31 may be used, but may affect other parts of the ZAF application.

If `ZAF_CLR_NULL` is passed into *index*, the new color is loaded into the next unused `ZafLogicalColor` index (beginning with 32). *red*, *green*, and *blue* specify the 8-bit RGB values of the new color.

Different native environments and different display modes may uniquely handle `AddColor()`. For example, a Motif palette-mapped environment may be

unable to return the exact RGB color requested and may instead return a “closest match” or no color at all. The color returned from `AddColor()`, and any other entry in the color table may be checked using `ColorInfo()`.

`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 already used, 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);
```

AddFont

```
virtual ZafLogicalFont AddFont(ZafLogicalFont index, char
    *fontFamily, int pointSize, ZafFontWeight weight =
    ZAF_FNT_WEIGHT_NORMAL, ZafFontSlant slant =
    ZAF_FNT_SLANT_NORMAL) = 0;
```

DestroyFont

```
virtual ZafError DestroyFont(ZafLogicalFont font) = 0;
```

`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
<code>ZAF_FNT_SMALL</code>	Small font, commonly used in icons
<code>ZAF_FNT_DIALOG</code>	Dialog font, commonly used in dialog windows
<code>ZAF_FNT_APPLICATION</code>	Application font, commonly used in multi-line text fields
<code>ZAF_FNT_SYSTEM</code>	System font, commonly used in buttons and menus
<code>ZAF_FNT_FIXED</code>	Fixed-width font, commonly used in code snippets

Run-time support for any available font may be added via `AddFont()`. *index* specifies the zero-based index (or logical font) into the ZAF internal font table where the font is to be inserted (each display object has a unique font table). The 5 fonts preloaded by ZAF occupy indices 0-4.

To allow more than 10 fonts in the internal font table, simply modify the `ZAF_MAXFONTS` constant in the header file `z_dsp.hpp` and recompile the libraries. Due to the size of the typedef `ZafLogicalFont`, `ZAF_MAXFONTS` may be at most 127.

If `ZAF_FNT_NULL` is passed into *index*, the new font is loaded into the next available `ZafLogicalFont` index. *fontFamily* specifies the name of the font family, and *pointSize* specifies the desired point size of the font to be supported.

weight specifies the desired weight of the font, and may be one of the following:

- `ZAF_FNT_WEIGHT_NORMAL`
- `ZAF_FNT_WEIGHT_BOLD`

slant specifies the desired slant of the font, and may be one 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 already used, or some other error occurs, `ZAF_FNT_NULL` is returned.

It is important to note that `AddFont()` can only return a valid font when that font exists in the native environment. Each native environment has different methods of “closest matching” fonts based on a request. Therefore, an `AddFont()` call that may return an acceptable font on one environment may return an unacceptable font on another environment, or no font at all. [FontInfo\(\)](#) may be used to examine the characteristics of the font returned, if any.

`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);
```

Background

```
virtual ZafLogicalColor Background(void) const;  
virtual 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()**.

BeginDraw

```
virtual ZafError BeginDraw(OSDisplayContext  
    displayContext, OSDrawContext drawContext, const  
    ZafRegionStruct &draw, const ZafRegionStruct &clip) =  
    0;
```

EndDraw

```
virtual ZafError EndDraw(void) = 0;
```

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.

Each call to **BeginDraw()** must be matched with a call to **EndDraw()**, and **BeginDraw()/EndDraw()** pairs should not be nested. In other words, a call to **BeginDraw()** should not be followed by another call to **BeginDraw()** before calling **EndDraw()** first. Nested calls do not properly restore display characteristics.

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.

Bitmap

```
virtual ZafError Bitmap(ZafCoordinate column,  
    ZafCoordinate line, ZafBitmapStruct &bitmap) = 0;
```


Bitmap() is used to display the bitmap specified by *bitmap*, 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.

Normally, *bitmap* should be a ZafBitmapData object, so that the OS bitmap created by ConvertToOSBitmap() will be deleted by the ZafBitmapData destructor when the ZafBitmapData object is deleted. If *bitmap* is a ZafBitmapStruct, the programmer is responsible for deleting the OS bitmap created by ConvertToOSBitmap().

<i>CellHeight</i>	int CellHeight () const;
<i>CellWidth</i>	int CellWidth () const;
<i>SetCellSize</i>	void SetCellSize (int cellHeight, int cellWidth);
<i>miniDenominatorX</i>	long miniDenominatorX ;
<i>miniDenominatorY</i>	long miniDenominatorY ;
<i>miniNumeratorX</i>	long miniNumeratorX ;
<i>miniNumeratorY</i>	long miniNumeratorY ;
<i>pixelsPerInchX</i>	long pixelsPerInchX ;
<i>pixelsPerInchY</i>	long pixelsPerInchY ;

These members are all initialized by each environment in the derived class constructors. CellHeight() and CellWidth() depend on the system font, and correspond to the number of pixels per cell in the ZAF_CELL coordinate system. SetCellSize() is an advanced method only to be called before any window is displayed. The programmer should normally not call SetCellSize().

A mini-cell corresponds to the ZAF_MINICELL coordinate system, and is calculated using the pixel values of a cell. CellWidth() multiplied by *miniNumeratorX* divided by *miniDenominatorX* specifies the number of pixels per mini-cell along the x-axis. CellHeight() multiplied by *miniNumeratorY* divided by *miniDenominatorY* specifies the number of pixels per mini-cell along the y-axis.

pixelsPerInchX specifies the number of pixels in an inch along the x-axis, and *pixelsPerInchY* specifies the number of pixels in an inch along the y-axis.

<i>ClipRegion</i>	virtual ZafRegionStruct ClipRegion (void) const;
	virtual ZafRegionStruct SetClipRegion (const ZafRegionStruct ®ion);

`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 3-D 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();
```

ColorInfo

```
virtual ZafError ColorInfo(ZafLogicalColor index,
    ZafUInt8 &red, ZafUInt8 &green, ZafUInt8 &blue) = 0;
```

`ColorInfo()` returns information about the logical ZAF color corresponding to the zero-based *index* into the ZAF internal color table where the color is stored (each display object has a unique color table). On return, *red*, *green*, and *blue* specify the 8-bit RGB values of the logical color. If *index* is out of range *red*, *green*, and *blue* are not modified and `ZAF_ERROR_INVALID_INDEX` is returned. See [AddColor\(\)](#) for more information about logical colors.

columns

```
int columns;
```

lines

```
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. On multiple-monitor systems such as the Apple Macintosh, the main monitor is the display that contains the menu bar.

*ConvertToOS-
Bitmap*
ConvertToOSIcon
*ConvertToOS-
Mouse*

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

*ConvertToZaf-
Bitmap*
ConvertToZafIcon
*ConvertToZaf-
Mouse*

```
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

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

ConvertXValue

long **ConvertXValue**(long value, ZafCoordinateType typeIn, ZafCoordinateType ZafCoordinateType typeOut);

ConvertXValue() converts *value*, specified along the x-axis in the *typeIn* coordinate system to the *typeOut* coordinate system, and returns the result.

ConvertYValue

long **ConvertYValue**(long value, ZafCoordinateType typeIn, ZafCoordinateType ZafCoordinateType typeOut);

ConvertYValue() converts *value*, specified along the y-axis in the *typeIn* coordinate system to the *typeOut* coordinate system, and returns the result.

CoordinateType

ZafCoordinateType **CoordinateType**(void) const;
virtual ZafCoordinateType
 SetCoordinateType(ZafCoordinateType coordinateType);

CoordinateType() specifies the coordinate system used by the display device. See [ZafCoordinateType](#) for more information on coordinate systems in ZAF. The default value for CoordinateType() is display-dependent, but it may be changed by calling SetCoordinateType(). SetCoordinateType() should normally not be called for ZafScreenDisplay, but may be called for ZafPrinter.

DestroyColor

virtual ZafError **DestroyColor**(ZafLogicalColor color) = 0;
See [AddColor](#)().

DestroyFont

virtual ZafError **DestroyFont**(ZafLogicalFont font) = 0;
See [AddFont](#)().

DestroyOSBitmap

virtual ZafError **DestroyOSBitmap**(ZafBitmapStruct &bitmap);

DestroyOSIcon

virtual ZafError **DestroyOSIcon**(ZafIconStruct &icon);

DestroyOSMouse

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.

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

```
DisplayContext     virtual OSDisplayContext DisplayContext(void) const = 0;
                    virtual OSDisplayContext
                        SetDisplayContext(OSDisplayContext context) = 0;
RestoreDisplay-    virtual OSDisplayContext RestoreDisplayContext(void) = 0;
Context
```

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.

```
DrawContext       virtual OSDrawContext DrawContext(void) const = 0;
                    virtual OSDrawContext SetDrawContext(OSDrawContext
                    context) = 0;
RestoreDraw-      virtual OSDrawContext RestoreDrawContext(void) = 0;
Context
```

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.

```
DisplayMode       virtual 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 Macin-

tosh, `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
<code>ZAF_DISPLAY_COLOR</code>	Indicates that the run-time system supports color or grayscale
<code>ZAF_DISPLAY_MONO</code>	Indicates that the run-time system supports only black and white

DisplayType

```
virtual ZafClassName DisplayType(void) const;
```

`DisplayType()` returns a string that uniquely identifies the derived display class. This allows a programmer to perform display-specific tasks when outputting to various devices. “ScreenDisplay” or “ZafPrinter” are returned for the default ZAF display devices.

Ellipse

```
virtual ZafError Ellipse(ZafCoordinate left,
    ZafCoordinate top, ZafCoordinate right, ZafCoordinate
    bottom, float startAngle, float endAngle, int width =
    ZAF_HAIR_LINE, bool fill = false) = 0;
```

`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 and moving counter-clockwise (using the polar model). 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 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()`. `SetForeground()`, `SetBackground()`, `SetFillPattern()`, and `SetLineStyle()` (or just `SetPalette()`) must be called before calling `Ellipse()`, to ensure that the correct settings are used when the ellipse is drawn; otherwise, the palette information used may have been previously set by some other object. 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();
}
```

FillPattern

```
virtual ZafLogicalFillPattern FillPattern(void) const;
virtual 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

Font

```
virtual ZafLogicalFont Font(void) const;
virtual 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().

FontInfo

```
virtual ZafError FontInfo(ZafLogicalFont index, char
    *fontFamily, int bufferSize, int *pointSize,
```

```
ZafFontWeight *weight = ZAF_NULLP(ZafFontWeight),  
ZafFontSlant *slant = ZAF_NULLP(ZafFontSlant)) = 0;
```

FontInfo() returns information about the logical ZAF font corresponding to the zero-based *index* into the ZAF internal font table where the font is stored (each display object has a unique font table). If *fontFamily* is not null, the font family name is copied into the buffer pointed to by *fontFamily* (at most *bufferSize* characters are copied into the buffer). If *pointSize* is not null, the point size is copied into the integer pointed to by *pointSize*. If *weight* is not null, the font weight is copied into the ZafFontWeight pointed to by *weight*. If *slant* is not null, the font slant is copied into the ZafFontSlant pointed to by *slant*. If *index* is out of range *fontFamily*, *pointSize*, *weight*, and *slant* are not modified and ZAF_ERROR_INVALID_INDEX is returned. See [AddFont\(\)](#) for more information about logical fonts.

Foreground

```
virtual ZafLogicalColor Foreground(void) const;  
virtual 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().

Icon

```
virtual ZafError Icon(ZafCoordinate column, ZafCoordinate  
    line, ZafIconStruct &icon) = 0;
```

Icon() is used to display the icon specified by *icon*, 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.

Normally, *icon* should be a ZafIconData object, so that the OS icon created by ConvertToOSIcon() will be deleted by the ZafIconData destructor when the ZafIconData object is deleted. If *icon* is a ZafIconStruct, the programmer is responsible for deleting the OS icon created by ConvertToOSIcon().

InitializeOSBitmap

```
static ZafError InitializeOSBitmap(ZafBitmapStruct  
    &bitmap);
```

InitializeOSIcon

```
static ZafError InitializeOSIcon(ZafIconStruct &icon);
```

InitializeOSMouse

```
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.

Line

```
virtual ZafError Line(ZafCoordinate column1,
    ZafCoordinate line1, ZafCoordinate column2,
    ZafCoordinate line2, int width = ZAF_HAIR_LINE) = 0;
```

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, and the line is drawn with the Foreground() color, using LineStyle(). SetForeground() and SetLineStyle() (or just SetPalette()) must be called before calling Line(), to ensure that the correct settings are used when the line is drawn; otherwise, the palette information used may have been previously set by some other object. Normally, Line() returns ZAF_ERROR_NONE.

lines

```
int lines;
```

See [columns](#).

LineStyle

```
virtual ZafLogicalLineStyle LineStyle(void) const;
virtual 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().

Mode

```
virtual ZafLogicalMode Mode(void) const;
virtual 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
ZAF_MODE_COPY	Draws normally, regardless of what was previously in the same position
ZAF_MODE_XOR	Performs a logical exclusive OR between the pixels being drawn, and the pixels that were already in the same position

MonoBackground

```
virtual ZafLogicalColor MonoBackground(void) const;  
virtual 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().

MonoForeground

```
virtual ZafLogicalColor MonoForeground(void) const;  
virtual 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().

Mouse

```
virtual ZafError Mouse(ZafCoordinate column,  
    ZafCoordinate line, ZafMouseStruct &mouse) = 0;
```

Mouse() is used to display the mouse cursor specified by *mouse*, 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`.

Normally, *mouse* should be a `ZafMouseData` object, so that the OS mouse cursor created by `ConvertToOSMouse()` will be deleted by the `ZafMouseData` destructor when the `ZafMouseData` object is deleted. If *mouse* is a `ZafMouseStruct`, the programmer is responsible for deleting the OS mouse cursor created by `ConvertToOSMouse()`.

Origin

```
void Origin(ZafCoordinate &x, ZafCoordinate &y) const;
virtual void SetOrigin(ZafCoordinate x, ZafCoordinate y);
```

`Origin()` specifies the drawing origin used by the display device within the current drawing context. For example, after calling `ZafWindowObject::BeginDraw()`, an object's drawing context is itself (meaning its top left corner is at (0, 0)). To draw relative to the position (2, 2), the object sets the origin to (2, 2). Thereafter, a call to draw a bitmap at position (0, 0) will actually draw within the object's region at (2, 2). The default value for `Origin()` is (0, 0), but it may be changed by calling `SetOrigin()`.

Palette

```
virtual ZafPaletteStruct Palette(void) const;
virtual 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>

Pixel

```
virtual ZafError Pixel(ZafCoordinate column,
    ZafCoordinate line, ZafLogicalColor color =
```

```
ZAF_CLR_DEFAULT, ZafLogicalColor mono =  
ZAF_MONO_DEFAULT) = 0;
```

`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`.

Polygon

```
virtual ZafError Polygon(int numPoints, const  
    ZafCoordinate *polygonPoints, int width =  
    ZAF_HAIR_LINE, bool fill = false, bool close = false)  
    = 0;
```

`Polygon()` is used to display a polygon specified by the parameters. The parameter *numPoints* specifies the number of points in the parameter *polygonsPoints*, and *polygonsPoints* is a list of points that specify the vertices of the polygon. The *width* parameter specifies the width 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. `SetForeground()`, `SetBackground()`, `SetFillPattern()`, and `SetLineStyle()` (or just `SetPalette()`) must be called before calling `Polygon()`, to ensure that the correct settings are used when the polygon is drawn; otherwise, the palette information used may have been previously set by some other object. 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.  
    ZafCoordinate 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();
}

```

preSpace
postSpace

```
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.

Rectangle

```

ZafError Rectangle(const ZafRegionStruct &region, int
    width = ZAF_HAIR_LINE, bool fill = false);
virtual ZafError Rectangle(ZafCoordinate left,
    ZafCoordinate top, ZafCoordinate right, ZafCoordinate
    bottom, int width = ZAF_HAIR_LINE, bool fill = false)
    = 0;

```

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 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()*. *SetForeground()*, *SetBackground()*, *SetFillPattern()*, and *SetLineStyle()* (or just *SetPalette()*) must be called before calling *Rectangle()*, to ensure that the correct settings are used when the rectangle is drawn; otherwise, the palette information used may have been previously set by some other object. Normally, *Rectangle()* returns *ZAF_ERROR_NONE*.

RectangleXORDiff

```
virtual ZafError RectangleXORDiff(const ZafRegionStruct
    *oldRegion, const ZafRegionStruct *newRegion) = 0;
```

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();
}
```

RegionCopy

```
virtual ZafError RegionCopy(const ZafRegionStruct
    &oldRegion, int newColumn, int newLine) = 0;
```

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.

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

```
RestoreDisplayCon- virtual OSDisplayContext RestoreDisplayContext(void) = 0;
text                See DisplayContext\(\).
```

```
RestoreDrawCon- virtual OSDrawContext RestoreDrawContext(void) = 0;
text                See DrawContext\(\).
```

```
Scale              int Scale(int *scaleNumerator = ZAF_NULLP(int), int
                                *scaleDenominator = ZAF_NULLP(int)) const;
virtual int SetScale(int scaleNumerator, int
                                scaleDenominator);
```

Scale() specifies the scaling fraction used by the display device. For example, if Scale() is 1/2, any drawing will occur at one-half the size specified and at one-half the position specified. So if a bitmap is requested at position (4, 4) and Scale() is 1/2, it will actually be drawn at (2, 2) and at half its normal size. The default value for Scale() is 1/1, but it may be changed by calling SetScale().

Text

```
virtual ZafError Text(ZafCoordinate left, ZafCoordinate
    top, const ZafIChar *text, int length = -1, int
    hotKeyIndex = -1, bool fill = false) = 0;

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(ZafCoordinate left, ZafCoordinate
    top, ZafCoordinate right, ZafCoordinate bottom, const
    ZafIChar *text, int length = -1, ZafHzJustify
    hzJustify = ZAF_HZ_LEFT, ZafVtJustify vtJustify =
    ZAF_VT_CENTER, int hotKeyIndex = -1, bool fill =
    false) = 0;
```

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()`, `SetForeground()`, `SetBackground()`, `SetFillPattern()`, `SetFont()`, and `SetLineStyle()` (or just `SetPalette()`) must be called before calling `Text()`, to ensure that the correct settings are used when the text is drawn; otherwise, the palette information used may have been previously set by some other object. 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.

TextSize

```
virtual ZafRegionStruct TextSize(const ZafIChar *text,  
    int length = -1) = 0;
```

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	IsA	NumberID
ClassName	ListIndex	StringID
EvaluatelsA	Next	
Find	Previous	

Inheritance Root class

Declaration `#include <z_list.hpp>`

Description 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

Constructor ZafElement initializes its members to the following default values:

Member Initializations

ZafElement

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

Member Initializations

NumberID()	0
Previous()	null
StringID()	null

ZafElement(void);

The ZafElement class constructor should only be instantiated from a derived class's constructor such as ZafButton, ZafStringData, or ZafKeyboard.

ZafElement(const ZafElement ©);

The copy constructor creates a new ZafElement object and initializes its data from *copy*.

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.

ClassID

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

ClassName

```
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”).

The use of `ClassName()` is preferred over the use of `ClassID()`. Every class in ZAF has a unique `ClassID()`, but classes that the programmer derives from ZAF classes are all assigned the same `ClassID()` unless specifically set to another value by the programmer.

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,
    ZAF_NULLP(ZafIChar), myBitmap);
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,
    ZAF_NULLP(ZafIChar), myBitmap);
printf("The element is: %s\n", element->
    ZafWindowObject::ClassName());
=====
The element is: ZafWindowObject
```

EvaluateIsA

```
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.

Find

```
virtual ZafElement *Find(ZafNumberID numberID);  
virtual ZafElement *Find(ZafStringID stringID);
```

Find() provides a base for doing searches in lists of ZafElement objects. ZafElement::Find() returns null if *numberID* or *stringID* do not match this object, and it returns "this" if it is a match.

Though Find() returns a ZafElement pointer, the programmer may use the overloaded function GetObject() in classes derived from ZafList to get a pointer of the same type.

IsA

```
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 (data->IsA(ZafDateData::className))
    date = DynamicPtrCast(data, ZafDateData);
```

ListIndex

```
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.

Next
Previous

```
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
ZafData	ZafData *	ZafDataManager
ZafDevice	ZafDevice *	ZafEventManager
ZafWindowObject	ZafWindowObject *	derived ZafWindows
ZafConstraint	ZafConstraint *	ZafGeometryManager
ZafRegionElement	ZafRegionElement *	ZafDisplay
ZafQueueElement	ZafQueueElement *	ZafEventManager
ZafTableRecord	ZafTableRecord *	ZafTable

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.

NumberID

```
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(ZafIChar),
    ZAF_NULLP(ZafBitmapData)));
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

Class	NumberID()
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

StringID

```
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(ZafIChar),
    ZAF_NULLP(ZafBitmapData)));
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"

ZafEraStruct

direction	eraFormat	offset
endDate	eraName	startDate

Inheritance Root struct

Declaration `#include <z_loc.hpp>`

Description ZafEraStruct is used by [ZafLocaleStruct](#) to store information about an era of time. An array of ZafEraStruct objects is used for each locale, where each ZafEraStruct object stores information about a single era.

Members

direction `ZafUInt16 direction;`

If direction is non-zero, the era begins on or after the year 1 AD, and the year value increases chronologically for the era; otherwise the year value decreases chronologically for the era, such as BC values.

endDate `ZafUInt32 endDate;`

endDate is the Julian value of the era's last day.

eraFormat `ZafIChar eraFormat[8];`

eraFormat is the output format string for the era when the format specifier "%E" is used. Follow source code in ZafUTime::OutputFormat() for details.

eraName `ZafIChar eraName[6];`

eraName is the name of the era. For example, "BC" is the name of the era before the year 1 AD.

offset `ZafUInt16 offset;`

offset specifies the local beginning year number of the era, usually 1 or 2 (or 4713 for the BC era). For example, the "AD" era has an offset of 1, meaning the first year of the era is 1 AD.

startDate ZafUInt32 **startDate**;

startDate is the Julian value of the era's first day.

ZafErrorStub

	Beep	ErrorMessage	ReportError
Inheritance	Root class		
Declaration	#include <z_error.hpp>		
Description	ZafErrorStub serves solely as a base class for ZafErrorSystem, so that ZafErrorSystem is not linked into a program that does not make use of it. ZafErrorStub provides a ReportError() function that formats a scanf-style error message and calls the pure virtual function ErrorMessage() for ZafErrorSystem to display the error. The programmer should normally not derive from this class.		
Constructor	ZafErrorStub (void); This constructor creates a ZafErrorStub object. Since ZafErrorStub is a virtual class, the programmer will normally not call this constructor.		
Destructor	virtual ~ ZafErrorStub (void); The destructor is used to free the memory associated with a ZafErrorStub object. Since ZafErrorStub is a virtual class, the programmer will not normally call this destructor.		
Members			
Beep	static void Beep (void); Beep() causes the system to emit an audible alert appropriate to the system. On systems that allow the end user to modify the system's audible alert, Beep() emits the alert chosen by the end user.		
ErrorMessage	virtual ZafDialogEvent ErrorMessage (ZafWindowObject *object, const ZafIChar *title, ZafDialogFlags dlgFlags, const ZafIChar *message) = 0; ErrorMessage() is a pure virtual function. ZafErrorSystem overloads this function to provide basic error display functionality.		

ReportError

```
ZafDialogEvent ReportError(ZafWindowObject *object, const  
    ZafIChar *title, ZafDialogFlags dlgFlags, const  
    ZafIChar *format, ...);
```

```
ZafDialogEvent ReportError(ZafWindowObject *object, const  
    ZafIChar *title, ZafDialogFlags dlgFlags, ZafUInt16  
    bufferSize, const ZafIChar *format, ...);
```

`ReportError()` formats a `scanf`-style error message and calls the pure virtual function `ErrorMessage()` for `ZafErrorSystem` to display the error. *object* is the window object that generated the error message, *title* is the text to appear in the title bar of the error window, and *format* specifies the format string used to build the error message (along with the variable arguments) to appear in the window. *dlgFlags* specifies which buttons are to be placed on the error window, controlling what the user may respond. The first method allocates a temporary 1024 character buffer for the entire message, and the second allocates a temporary buffer of *bufferSize* characters for the entire message. See [ErrorMessage\(\)](#) for more information.

ZafErrorSystem

[ErrorMessage](#)

Inheritance	ZafErrorSystem : ZafErrorStub
Declaration	<code>#include <z_error.hpp></code>
Description	<p>ZafErrorSystem is ZAF’s built-in support for error-reporting. If a ZafErrorSystem has been instantiated, the error system may be used to beep and present a modal error window. The user must respond by selecting one of the available buttons such as "OK" or "Cancel". As with other ZAF classes, ZafErrorSystem utilizes the native environment’s API, if available.</p> <p>Some classes have built-in error handling, and the programmer needn’t do anything beyond instantiating a ZafErrorSystem for it to work. Some of these classes are ZafBignum, ZafDate, ZafInteger, ZafReal, ZafTime, and ZafU-Time. The programmer may utilize ZafErrorSystem at any time.</p>
Constructors	<p>All ZafErrorSystem constructors initialize the member variables associated with an instantiated ZafErrorSystem object. The default values set by the ZafErrorSystem follow.</p>

Member Initializations

ZafErrorSystem

<code>ClassID()</code>	<code>ID_ZAF_ERROR_SYSTEM</code>
<code>ClassName()</code>	<code>"ZafErrorSystem"</code>

ZafErrorSystem(void);

This constructor creates a ZafErrorSystem object, initializes information used by ZafErrorSystem, and sets the global zafErrorSystem. There should only be one ZafErrorSystem instantiated during the life of an application.

ZafErrorSystem(const ZafErrorSystem ©);

The copy constructor creates a new ZafErrorSystem object and initializes its data from copy.

Destructor

```
virtual ~ZafErrorSystem(void);
```

The destructor is used to free the memory associated with a ZafErrorSystem object. It chains to the ZafErrorStub destructor.

Members*ErrorMessage*

```
virtual ZafDialogEvent ErrorMessage(ZafWindowObject
    *object, const ZafIChar *title, ZafDialogFlags
    dlgFlags, const ZafIChar *message);
```

ErrorMessage() causes the error system to present a modal error dialog window. *object* is the window object that generated the error message, *title* is the text to appear in the title bar of the error window, and *message* is the error message to appear in the window. The error window is sized such that the entire message is visible. *dlgFlags* specifies which buttons are to be placed on the error window, controlling what the user may respond. *dlgFlags* is made up of one or more of the following values, that may be logically OR'd together:

ZafDialogFlags	Description
ZAF_DIALOG_ABORT	Provides an "Abort" button
ZAF_DIALOG_CANCEL	Provides a "Cancel" button
ZAF_DIALOG_HELP	Provides a "Help" button for those environments that allow it
ZAF_DIALOG_IGNORE	Provides an "Ignore" button
ZAF_DIALOG_NO	Provides a "No" button
ZAF_DIALOG_OK	Provides an "OK" button
ZAF_DIALOG_RETRY	Provides a "Retry" button
ZAF_DIALOG_YES	Provides a "Yes" button

Since ZafErrorSystem may be used to indicate errors in ZAF itself, ErrorMessage() uses a native window when available to display the message instead of a ZafWindow.

ZafEventManager

Blocked	GetObject	ReadFromEnd
DestroyEvent	PollDevices	SetDeviceImage
Event	Put	SetDevicePosition
Get	ReadFromBeginning	SetDeviceState

Inheritance ZafEventManager : [ZafList](#)

Declaration #include <z_evtmgr.hpp>

Description 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.

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

Blocked

```
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.

DestroyEvent

```
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.

Event

```
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.

Get

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

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 ZafWindowOb-

ject::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.

GetObject

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

These overloaded functions return a pointer to the device attached to the event manager specified by *numberID* or *stringID*. If the device was not found, null is returned.

PollDevices

```
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.

Put

```
virtual bool 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. Put() returns true if the event was successfully added to the queue; otherwise Put() returns false.

ReadFromBeginning

```
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.

ReadFromEnd `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.

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

The `SetDeviceImage()` 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->SetDeviceImage(E_MOUSE, DM_WAIT);
```

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

The `SetDevicePosition()` 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->SetDevicePosition(E_MOUSE, 0, 0);
```

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

The `SetDeviceState()` 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->SetDeviceState(E_MOUSE,  
    D_STATE);
```

ZafEventManager

eventType	modifiers
logicalValue	rawCode

Inheritance	Root struct
Declaration	#include <z_win.hpp>
Description	ZafEventManager objects are used in tables to associate native events with ZAF logical events. For example, when the <Enter> key is typed, a ZafEventManager entry for a default button would specify the equivalent logical ZAF event L_SELECT. The last entry in a table of ZafEventManager entries should specify the special logical ZAF event L_NONE to terminate a table search. See ZafWindowObject::LogicalEvent() for more information.
Members	ZafLogicalEvent logicalValue ; <i>logicalValue</i> specifies the logical ZAF event to be associated with the native event. Possible values of this member are any of the logical event values defined in the header file z_logevt.hpp. This member should be L_NONE for the last entry in a table.
<i>eventType</i>	ZafEventType eventType ; <i>eventType</i> specifies the type of the native event. Possible values of this member are E_KEY (for keyboard events) and E_MOUSE (for mouse events).
<i>rawCode</i>	ZafRawCode rawCode ; <i>rawCode</i> specifies the raw value of the native event. For keyboard events (eventType == E_KEY), rawCode is the raw key value, such as ENTER or F1 (see the header file z_keymap.hpp for the possible values). For mouse events (eventType == E_MOUSE), rawCode is the raw mouse value, which may be any combination of the following bitwise values:

Raw Code	Description
M_LEFT	Indicates the left (or only) mouse button is down
M_LEFT_CHANGE	Indicates the left (or only) mouse button was just pressed or released
M_MIDDLE	Indicates the middle mouse button (if any) is down

Raw Code	Description
M_MIDDLE_CHANGE	Indicates the middle mouse button (if any) was just pressed or released
M_RIGHT	Indicates the right mouse button (if any) is down
M_RIGHT_CHANGE	Indicates the right mouse button (if any) was just pressed or released
S_DOUBLE_CLICK	Indicates the button was double-clicked

modifiers

ZafRawCode **modifiers**;

modifiers specifies modifier keys that were pressed at the time of the event. This member may be any combination of the following bitwise values:

Modifiers	Description
S_ALT	Indicates that the <alt> key was depressed
S_CAPS_LOCK	Indicates that the <caps lock> key was depressed
S_CMD	Indicates that the <command> key was depressed
S_CTRL	Indicates that the <control> key was depressed
S_INSERT	Indicates that the <insert> key was depressed
S_KEYDOWN	Indicates that the key was pressed
S_KEYUP	Indicates that the key was released
S_LEFT_SHIFT	Indicates that the left <shift> key was depressed
S_NUM_LOCK	Indicates that the <num lock> key was depressed
S_OPT	Indicates that the <option> key was depressed
S_RIGHT_SHIFT	Indicates that the right <shift> key was depressed
S_SCROLL_LOCK	Indicates that the <scroll lock> key was depressed
S_SHIFT	Indicates that either <shift> key was depressed

ZafEventStruct

converted	modifiers	Scroll
data	osEvent	text
Device	Position	type
Display	rawCode	VoidData
EventManager	Region	Window
InputType	route	WindowManager
Key	ScreenID	WindowObject

Inheritance	Root structure
Declaration	#include <z_event.hpp>
Description	ZafEventStruct provides support for ZAF’s portable event structure. Portable ZAF events and native events alike are sent via ZafEventStruct objects through the ZAF system. Native events are packaged up into ZafEventStruct objects for portable event passing.
Constructors	All ZafEventStruct constructors initialize the member variables associated with an instantiated ZafEventStruct object. The default values set by the ZafEventStruct follow.

Member Initializations

ZafEventStruct

converted	null
data	null
modifiers	0
osEvent	user-supplied parameter
rawCode	0
route	null
text	null
type	0

ZafEventStruct(void);

The default constructor creates a null event. This constructor is called when an event is declared but not defined, and should be used carefully. If this constructor is used, the programmer must initialize all event information necessary for the desired event before any object may try to evaluate it.

```
ZafEventStruct(ZafEventType type, OSEventStruct  
    *osEvent);
```

This constructor initializes the event's type to *type* and copies all the native event information from *osEvent* into the event's *osEvent*.

```
ZafEventStruct(ZafEventType type, ZafRawCode rawCode =  
    0);
```

This constructor initializes the event's type to *type* and the event's *rawCode* is copied from *rawCode*. Since *rawCode* is a default parameter, this constructor may be used to create an event whose type requires no additional event information.

```
ZafEventStruct(ZafEventType type, ZafRawCode rawCode,  
    ZafKeyStruct key);
```

This constructor initializes the event's type to *type*, the event's *rawCode* to *rawCode*, and the key flavor of the event's union to *key*.

```
ZafEventStruct(ZafEventType type, ZafRawCode rawCode,  
    ZafPositionStruct position);
```

This constructor initializes the event's type to *type*, the event's *rawCode* to *rawCode*, and the position flavor of the event's union to *position*.

```
ZafEventStruct(ZafEventType type, ZafRawCode rawCode,  
    ZafRegionStruct region);
```

This constructor initializes the event's type to *type*, the event's *rawCode* to *rawCode*, and the region flavor of the event's union to *region*.

```
ZafEventStruct(ZafEventType type, ZafRawCode rawCode,  
    ZafScrollStruct scroll);
```

This constructor initializes the event's type to *type*, the event's *rawCode* to *rawCode*, and the scroll flavor of the event's union to *scroll*.

An example of how to create and send an event follows:

```
// Create and post a close event.  
ZafEventStruct closeEvent(S_CLOSE);  
zafEventManager->Put(closeEvent);
```

```
// Create a copy event and send it to the string field.
ZafEventStruct copyEvent(S_COPY);
string->Event(copyEvent);
```

Members

converted

ZafWindowObject ***converted**;

If this member is null, the event's information has not been converted in the context of a window object. If it is non-null, it points to the object in whose context the event information was converted. For example, a mouse event's position is specified with the object's top left being (0,0), so the event's position information must be converted in the context of the object in question.

```
union
{
    ZafKeyStruct key;
    ZafRegionStruct region;
    ZafPositionStruct position;
    ZafScrollStruct scroll;

    ZafDevice *device;
    ZafDisplay *display;
    ZafEventManager *eventManager;
    const ZafIChar *helpContext;
    ZafWindowObject *windowObject;
    ZafWindow *window;
    ZafWindowManager *windowManager;

    OSWindowID screenID;

    void *data;
};
```

data

The anonymous union stores additional information needed by various events. User events may utilize the union as needed. Union information may be accessed using the following methods.

Device

```
ZafDevice *Device(void) const;
void SetDevice(ZafDevice *tDevice);
```

Device() returns the device flavor of the event’s anonymous union, and SetDevice() may be used to set it.

Display

```
ZafDisplay *Display(void) const;
void SetDisplay(ZafDisplay *tDisplay);
```

Display() returns the display flavor of the event’s anonymous union, and SetDisplay() may be used to set it.

EventManager

```
ZafEventManager *EventManager(void) const;
void SetEventManager(ZafEventManager *tEventManager);
```

EventManager() returns the eventManager flavor of the event’s anonymous union, and SetEventManager() may be used to set it.

HelpContext

```
const ZafIChar *HelpContext(void) const;
void SetHelpContext(const ZafIChar *tHelpContext);
```

HelpContext() returns the helpContext flavor of the event’s anonymous union, and SetHelpContext() may be used to set it.

InputType

```
ZafEventType InputType(ZafEventType ccode = 0) const;
```

InputType() returns the type of event this is, whether it be a native event or a ZAF event. The possible return values are as follows:

InputType()	Description
E_KEY	Keyboard events
E_MOUSE	Mouse events
S_SYSTEM_EVENT	ZAF system events
L_LOGICAL_EVENT	ZAF logical events
E_DEVICE_EVENT	ZAF device events

Key

```
const ZafKeyStruct *Key(void) const;
void SetKey(ZafKeyStruct *tKey);
```

Key() returns the key flavor of the event’s anonymous union, and SetKey() may be used to set it.

modifiers

`ZafRawCode modifiers;`

This member stores the modifier keys that are associated with a keyboard or mouse event. The different modifier keys are represented with flags that may be combined into bit patterns. See `ZafKeyStruct::shiftState` for more information.

osEvent

`OSEventStruct osEvent;`

This member stores the native event information for environment-specific events. When the event's type member is `E_OSEVENT`, this member contains native event information.

Position

`const ZafPositionStruct *Position(void) const;`
`void SetPosition(ZafPositionStruct *tPosition);`

`Position()` returns the position flavor of the event's anonymous union, and `SetPosition()` may be used to set it.

rawCode

`ZafRawCode rawCode;`

This member stores raw information about an event. If the event is a keyboard event, `rawCode` stores the raw key value returned from the native environment (see `ZafKeyStruct::value` for more information). If the event is a mouse event, `rawCode` stores which buttons are depressed as bit values combined together. The possible values for `rawCode` in a mouse event are as follows:

Mouse event's raw-Code	Description
M_LEFT	The left mouse button has been depressed
M_LEFT_CHANGE	The mouse is moving with the left button depressed, or the left button has been released
M_MIDDLE	The middle mouse button has been depressed
M_MIDDLE_CHANGE	The mouse is moving with the middle button depressed, or the middle button has been released
M_RIGHT	The right mouse button has been depressed
M_RIGHT_CHANGE	The mouse is moving with the right button depressed, or the right button has been released

Region

`const ZafRegionStruct *Region(void) const;`
`void SetRegion(ZafRegionStruct *tRegion);`

Region() returns the region flavor of the event's anonymous union, and SetRegion() may be used to set it.

route

ZafWindowObject ***route**;

If this member is non-null, the window manager sends the event directly to the object specified.

ScreenID

OSWindowID **ScreenID**(void) const;
void **SetScreenID**(OSWindowID tScreenID);

ScreenID() returns the screenID flavor of the event's anonymous union, and SetScreenID() may be used to set it.

Scroll

const ZafScrollStruct ***Scroll**(void) const;
void **SetScroll**(ZafScrollStruct *tScroll);

Scroll() returns the scroll flavor of the event's anonymous union, and SetScroll() may be used to set it.

text

ZafIChar ***text**;

This member allows textual information to be associated with an event. If the event was generated by ZAF or the native environment, the object handling an event where text is non-null must delete it. See the appendix on Event Definitions under the application events for more information.

type

ZafEventType **type**;

This member specifies the event's type. See the appendix on Event Definitions for more information on the event types supported by ZAF.

VoidData

void ***VoidData**(void) const;
void **SetVoidData**(void *tData);

VoidData() returns the data flavor of the event's anonymous union, and SetVoidData() may be used to set it. This particular flavor of the event's anonymous union is provided for programmer use to allow associating data of any type with an event.

Window

```
ZafWindow *Window(void) const;  
void SetWindow(ZafWindow *tWindow);
```

Window() returns the window flavor of the event's anonymous union, and SetWindow() may be used to set it.

WindowManager

```
ZafWindowManager *WindowManager(void) const;  
void SetWindowManager(ZafWindowManager *tWindowManager);
```

WindowManager() returns the windowManager flavor of the event's anonymous union, and SetWindowManager() may be used to set it.

WindowObject

```
ZafWindowObject *WindowObject(void) const;  
void SetWindowObject(ZafWindowObject *tWindowObject);
```

WindowObject() returns the windowObject flavor of the event's anonymous union, and SetWindowObject() may be used to set it.

ZafFile

BinaryMode	Read	Temporary
Create	ReadData	TextMode
DerivedAccess	ReadOnly	Write
Error	ReadWrite	WriteData
Length	Seek	operator >>
OpenCreate	Tell	operator <<

Inheritance ZafFile : ZafElement

```
Declaration      #include <z_file.hpp>
```

Description	ZafFile is the base class for all file classes such as ZafDiskFile and ZafStorageFile . ZafFile is an abstract class since it defines pure virtual functions.
--------------------	---

Constructor The ZafFile class constructor initializes the members associated with a ZafFile object and its base classes. The default values set by ZafFile are listed below.

Member Initializations

ZafFile

Error() ZAF_ERROR_NONE

```
ZafFile(ZafFileMode mode);
```

This constructor is called by subclass constructors, initializes the members associated with a `ZafFile` object, and chains to the `ZafElement` constructor. *mode* specifies the mode in which the file is opened, and may be a combination of the following bitwise values:

ZafFileMode	Description
ZAF_FILE_BINARY	Causes the file to open in binary mode, meaning that <code>\n</code> and <code>\r</code> characters are not translated when reading or writing
ZAF_FILE_CREATE	Causes the file to be created, even if it already exists
ZAF_FILE_DERIVED_ACCESS	Causes the file to open in derived access mode, meaning that it is not actually opened or closed by ZAF, but by a derived class

ZafFileMode	Description
ZAF_FILE_OPENCREATE	Causes the file to be opened if it exists, or to be created if it doesn't exist
ZAF_FILE_READ	Causes the file to be opened in read-only mode, meaning that the file cannot be written
ZAF_FILE_READWRITE	Causes the file to be opened in read-write mode, meaning the file can be read and written
ZAF_FILE_TEMPORARY	Causes a temporary file to be opened, meaning that the file is deleted automatically after it is closed
ZAF_FILE_TEXT	Causes the file to open in text mode, meaning that '\n' and '\r' characters are translated appropriately for the environment when reading or writing

Destructor

```
virtual ~ZafFile(void);
```

This virtual destructor is used to free the memory associated with an instantiated ZafFile object and chains to the ZafElement destructor.

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.

BinaryMode

```
bool BinaryMode(void) const;
```

If BinaryMode() is true, the file is opened in binary mode, meaning that '\n' and '\r' characters are not translated when reading or writing.

Create

```
bool Create(void) const;
```

If Create() is true, the file is created, even if it already exists.

DerivedAccess `bool DerivedAccess(void) const;`

If `DerivedAccess()` is true, the file is opened in derived access mode, meaning that it is not actually opened or closed by ZAF, but by a derived class.

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

`Error()` stores the last error that occurred with the file. The default value of `Error()` is `ZAF_ERROR_NONE`, but it may be set by the file or file system 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 file:

Error()	Description
ZAF_ERROR_FILE_OPEN	The file specified could not be opened
ZAF_ERROR_FILE_POSITION	The file pointer could not be positioned
ZAF_ERROR_FILE_READ	An error occurred trying to read the file
ZAF_ERROR_FILE_WRITE	An error occurred trying to write the file
ZAF_ERROR_INVALID_ID	A file or directory could not be found with the numeric or string ID specified
ZAF_ERROR_INVALID_NAME	A file or directory could not be found with the name specified
ZAF_ERROR_INVALID_TARGET	The file could not be saved, since it is read-only
ZAF_ERROR_NONE	No error has occurred
ZAF_ERROR_STORAGE_VERSION	The storage file specified is the wrong version, and must be converted

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

Length `virtual ZafOffset Length(void) const = 0;`

`Length()` abstractly defines returning the length of a derived file object. `Length()` returns -1 if an error occurs.

OpenCreate

```
bool OpenCreate(void) const;
```

If `OpenCreate()` is true, the file is opened if it exists, or it is created if it doesn't exist.

Read

```
virtual int Read(ZafInt8 &value);
virtual int Read(ZafUInt8 &value);
virtual int Read(ZafInt16 &value);
virtual int Read(ZafUInt16 &value);
virtual int Read(ZafInt32 &value);
virtual int Read(ZafUInt32 &value);
int Read(int &value);
```

operator >>

```
ZafFile &operator>>(ZafInt8 &value);
ZafFile &operator>>(ZafUInt8 &value);
ZafFile &operator>>(ZafInt16 &value);
ZafFile &operator>>(ZafUInt16 &value);
ZafFile &operator>>(ZafInt32 &value);
ZafFile &operator>>(ZafUInt32 &value);
ZafFile &operator>>(int &value);
ZafFile &operator>>(double &value);
```

These overloaded members and operators simply call `ReadData()` and pass in *value* and the appropriate size of *value*. The `Read()` members return what `ReadData()` returns to them.

```
virtual int Read(ZafIChar *string, int length);
ZafFile &operator>>(ZafIChar *string);
```

This overloaded function and operator read a string out of the file into the *string* buffer. If *length* is -1, the whole string is read into *string*; otherwise, if the string to be read is longer than *length*, it is not read. The number of bytes read is returned. On success, this is the number of bytes in the string plus two for the size of the string. `Error()` is also set to an appropriate value.

```
virtual int Read(ZafIChar **string);
ZafFile &operator>>(ZafIChar **string);
```

This overloaded function and operator create a buffer large enough for the string to be read plus a null terminator, read the string out of the file into the

new buffer, and return a pointer to the buffer in *string*. The number of bytes read is returned. On success, this is the number of bytes in the string plus two for the size of the string. `Error()` is also set to an appropriate value.

```
virtual int Read(void *buffer, int size, int num);
```

This overloaded function simply calls `ReadData()`, passing it *buffer* and *size* * *num*. `Read()` returns whatever `ReadData()` returns.

ReadData

```
virtual int ReadData(void *buffer, int size) = 0;
```

`ReadData()` abstractly defines reading data from a derived file object. The data is read into *buffer*, which must have already been allocated by the programmer, and *size* specifies the number of bytes to read. On success, `ReadData()` returns the number of bytes read into *buffer*; otherwise zero is returned. `Error()` is also set to an appropriate value.

ReadOnly

```
bool ReadOnly(void) const;
```

If `ReadOnly()` is true, the file cannot be written.

ReadWrite

```
bool ReadWrite(void) const;
```

If `ReadWrite()` is true, the file may be read and written.

Seek

```
virtual int Seek(ZafOffset offset, ZafSeek location) = 0;
```

`Seek()` abstractly defines moving the file pointer of a derived file object. *location* specifies the position in the file from where *offset* specifies. Possible values of *location* and what they mean follow:

ZafSeek	ZafOffset
ZAF_SEEK_START	The offset is measured from the beginning of the file toward the end of the file
ZAF_SEEK_CURRENT	A positive offset is measured from the current file pointer, and a negative offset is measured toward the beginning of the file.
ZAF_SEEK_END	The offset is measured from the end of the file toward the beginning of the file

`Seek()` returns 0 if successful, and -1 if an error occurs (in which case `Error()` is also set appropriately).

Tell virtual ZafOffset **Tell**(void) const = 0;

Tell() abstractly defines returning the current offset of the file pointer in a derived file object. -1 is returned if an error occurs.

Temporary bool **Temporary**(void) const;

If Temporary() is true, the file is temporary, and is deleted automatically after it is closed.

TextMode bool **TextMode**(void) const;

If TextMode() is true, the file is opened in text mode, meaning that '\n' and '\r' characters are translated appropriately for the environment when reading or writing.

Write virtual int **Write**(ZafInt8 value);
 virtual int **Write**(ZafUInt8 value);
 virtual int **Write**(ZafInt16 value);
 virtual int **Write**(ZafUInt16 value);
 virtual int **Write**(ZafInt32 value);
 virtual int **Write**(ZafUInt32 value);
 int **Write**(int value);
 virtual int **Write**(double value);

operator << ZafFile &**operator<<**(ZafInt8 value);
 ZafFile &**operator<<**(ZafUInt8 value);
 ZafFile &**operator<<**(ZafInt16 value);
 ZafFile &**operator<<**(ZafUInt16 value);
 ZafFile &**operator<<**(ZafInt32 value);
 ZafFile &**operator<<**(ZafUInt32 value);
 ZafFile &**operator<<**(int value);
 ZafFile &**operator<<**(double value);

These overloaded members and operators simply call WriteData() and pass in *value* and the appropriate size of *value*. The Write() members return what WriteData() returns to them.

virtual int **Write**(const ZafIChar *string);
 ZafFile &**operator<<**(const ZafIChar *string);

This overloaded function and operator write *string* to the file. The number of bytes written is returned. On success, this is the number of bytes in the string plus two for the size of the string. `Error()` is also set to an appropriate value.

```
virtual int Write(const void *buffer, int size, int num);
```

This overloaded function simply calls `WriteData()`, passing it *buffer* and *size* * *num*. `Write()` returns whatever `WriteData()` returns.

WriteData

```
virtual int WriteData(const void *buffer, int size) = 0;
```

`WriteData()` abstractly defines writing data to a derived file object. *buffer* is a pointer to the data to be written, and *size* specifies the number of bytes to write. On success, `WriteData()` returns the number of bytes written; otherwise zero is returned. `Error()` is also set to an appropriate value.

ZafFileDialog

allFilesFilter	File	FullPath
Directory	Filter	GetFile

Inheritance	ZafFileDialog : ZafDialogWindow : ZafWindow : ((ZafWindowObject : ZafElement), ZafList)
Declaration	#include <z_fildlg.hpp>
Description	ZafFileDialog provides support for easily presenting a dialog that asks the end user to either choose an existing file or type the name of a new file to be created. ZafFileDialog utilizes the native file services on each environment, if available, so the dialog is familiar to the end user.
Constructors	All ZafFileDialog constructors initialize the member variables associated with an instantiated ZafFileDialog object. The default values set by the ZafFileDialog 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 ZafFileDialog. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafFileDialog	
allFilesFilter[]	(platform-dependent)
Directory()	" " (empty string)
File()	" " (empty string)
Filter()	" " (empty string)
ZafWindow	
Modal()	true†
ZafElement	
ClassID()	ID_ZAF_FILE_DIALOG
ClassName()	"ZafFileDialog"

ZafFileDialog(int left, int top, int width, int height);

This constructor is useful in straight-code situations, creating a file dialog object without putting it on the screen (see GetFile()). *left*, *top*, *width*, and *height* specify the size and position of the file dialog for environments that sup-

port custom size and location for file dialogs; otherwise the file dialog is sized and positioned appropriately by the environment.

Since `Destroyable()` is false by default, the programmer may create one `ZafFileDialog` and use it many times by calling `GetFile()` repeatedly for each new filename.

```
ZafFileDialog(const ZafFileDialog &copy);
```

The copy constructor creates a new `ZafFileDialog` object and initializes its data from *copy*.

An example of how to create a file dialog follows:

```
// Create a file dialog.
ZafFileDialog fileDialog(5, 5, 60, 14);

// Bring up the file dialog to open an existing file.
ZafEventType result = fileDialog.GetFile(ZAF_FILE_DIALOG_OPEN);

// Get the file name without any path information.
const ZafIChar *filename = fileDialog.File();

// Get the full path name for the file.
ZafIChar fullPath[ZAF_MAXPATHLEN];
fileDialog.FullPath(fullPath, ZAF_MAXPATHLEN);
```

Destructor

```
virtual ~ZafFileDialog(void);
```

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

allFilesFilter

```
static ZafIChar ZAF_FARDATA allFilesFilter[];
```

`allFilesFilter[]` is a portable string that specifies a filter that accepts all files. See [Filter\(\)](#) for more information.

Directory

```
const ZafIChar *Directory(void) const;
virtual void SetDirectory(const ZafIChar *directory);
```

Directory() specifies the full pathname of the parent directory of **File()** without a terminating path separator. **SetDirectory()** may be used to change it, but calls to **GetFile()** will change this attribute according to the file chosen by the end user.

File

```
const ZafIChar *File(void) const;
virtual void SetFile(const ZafIChar *file);
```

File() specifies the filename chosen by the end user. **SetFile()** may be used to change it, but calls to **GetFile()** will change this attribute according to the file chosen by the end user.

Filter

```
const ZafIChar *Filter(void) const;
virtual void SetFilter(const ZafIChar *filter);
```

Filter() specifies the non-portable file filter used by some environments when listing files the end user may choose from. For example, the Zinc Designer for Microsoft Windows sets the filter to "*.znc", allowing the user to choose Zinc data files. The end user may also type any preferred file name, but the environment's file dialog only displays the files the filter specifies in the file list. [all-FilesFilter\[\]](#) may be used to portably specify all files. **SetFilter()** may be used to change this attribute.

FullPath

```
void FullPath(ZafIChar *pathBuffer, int bufferSize);
```

FullPath() specifies the full pathname of the file chosen by the end user. *path-Buffer* must be allocated by the programmer before calling **FullPath()**, and *bufferSize* is the maximum number of characters that will be written to *path-Buffer*, guarding against writing past the end of the buffer.

GetFile

```
ZafDialogEvent GetFile(ZafFileDialogRequest request);
```

GetFile() causes the file dialog to be presented on the screen, and **GetFile()** returns after the user has either cancelled the file dialog, or successfully chosen

a file. *request* specifies the type of file dialog to present, and may be one of the following:

ZafFileDialogRequest	Description
ZAF_FILE_DIALOG_NEW	Allows the end user to browse to the desired directory and type a new or used filename
ZAF_FILE_DIALOG_OPEN	Allows the end user to browse to the desired directory and choose an existing file
ZAF_FILE_DIALOG_SAVEAS	Allows the end user to browse to the desired directory and type a new or used filename

GetFile() returns one of the following values, indicating the result of the file dialog:

Return value	Description
S_DLG_CANCEL	A file was not successfully chosen, so Directory(), File(), and FullPath() are invalid
S_DLG_OK	A file was successfully chosen, so Directory(), File(), and FullPath() are valid

ZafFileInfoStruct

attributes	length	ReadOnly
Directory	name	stringID
internalHandle	numberID	

Inheritance Root struct

Declaration `#include <z_file.hpp>`

Description ZafFileInfoStruct is used internally by [ZafFileSystem](#)'s subclasses' Find*() methods during file searches. The programmer must provide a ZafFileInfoStruct object to these methods, but FindFirst() initializes the object. See [ZafFileSystem](#) for more information.

Members

attributes

ZafFileAttribute **attributes**;

attributes specifies what type of object was found. The possible values of attribute are as follows:

ZafFileAttribute	Description
ZAF_FATTRIB_NONE	The object is a file, and may be written to
ZAF_FATTRIB_DIRECTORY	The object is a directory
ZAF_FATTRIB_READONLY	The object is read-only, so it may not be written to

Directory

bool **Directory**(void);

If Directory() is true, the object found is a directory; otherwise it is a file.

internalHandle

void ***internalHandle**;

internalHandle is used internally by subclasses of ZafFileSystem to keep track of the current file search, and should not be accessed by the programmer.

length

ZafOffset **length**;

If the object found is a file, length specifies the number of bytes the file contains.

<i>name</i>	<code>const ZafIChar *name;</code> name specifies the name of the file or directory that was found. No path information is included in the name.
<i>numberID</i>	<code>ZafNumberID numberID;</code> numberID specifies the ZafNumberID of the object found, if appropriate. For example, since ZafDiskFileSystem finds objects on disk, they don't have numberIDs. But ZafStorage finds objects in a Zinc data file, which do have numberIDs.
<i>ReadOnly</i>	<code>bool ReadOnly(void);</code> If ReadOnly() is true, the object found is read-only, meaning that it cannot be written to.
<i>stringID</i>	<code>ZafStringID stringID;</code> stringID specifies the ZafStringID of the object found, if appropriate. For example, since ZafDiskFileSystem finds objects on disk, they don't have stringIDs. But ZafStorage finds objects in a Zinc data file, which do have stringIDs.

ZafFileSystem

ChDir	FindFirst	parentDirectoryName
Close	FindNext	Remove
currentDirectoryName	GetCWD	Rename
Error	MkDir	Rmdir
FindClose	Open	rootDirectoryName

- Inheritance**

ZafFileSystem : ZafElement
- Declaration**

#include <z_file.hpp>
- Description**

ZafFileSystem is the base class for all file system classes such as ZafDiskFile-System and ZafStorage. ZafFileSystem is an abstract class since most of its functions are pure virtual.
- Constructor**

The ZafFileSystem class constructor initializes the members associated with a ZafFileSystem object and its base classes. The default values set by ZafFile-System are listed below.

Member Initializations

ZafFileSystem

Error()ZAF_ERROR_NONE

ZafFileSystem(void);

This constructor is called by subclass constructors, initializes the members associated with a ZafFileSystem object, and chains to the ZafElement constructor.

Destructor

virtual ~ZafFileSystem(void);

This virtual destructor is used to free the memory associated with an instantiated ZafFileSystem object and chains to the ZafElement destructor.

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.

ChDir

```
virtual int ChDir(const ZafIChar *newPath, ZafStringID
    stringID = ZAF_NULLP(ZafIChar), ZafNumberID numberID =
    0) = 0;
```

ChDir() abstractly defines changing the current directory for a derived file system. *newPath* specifies the path name of the directory to be changed to. *stringID* specifies the string identification constant associated with the directory, if applicable (such as in ZafStorage). *numberID* specifies the numeric identification constant associated with the directory, if applicable (such as in ZafStorage). ChDir() returns 0 on success, and -1 on failure. Error() is also set to an appropriate value.

Close

```
virtual void Close(ZafFile *file) = 0;
```

Close() abstractly defines closing *file*, which is a file in the file system previously opened with Open().

currentDirectory-
Name

```
static ZafIChar *currentDirectoryName;
```

currentDirectoryName is a public static specifying the name of the current directory in a file system, which is ".".

Error

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

Error() stores the last error that occurred with the file system. The default value of Error() is ZAF_ERROR_NONE, but it may be set by the file system 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 file system:

Error()	Description
ZAF_ERROR_FILE_OPEN	The file specified could not be opened
ZAF_ERROR_FILE_POSITION	The file pointer could not be positioned
ZAF_ERROR_FILE_READ	An error occurred trying to read the file
ZAF_ERROR_FILE_WRITE	An error occurred trying to write the file

Error()	Description
ZAF_ERROR_INVALID_ID	A file or directory could not be found with the numeric or string ID specified
ZAF_ERROR_INVALID_NAME	A file or directory could not be found with the name specified
ZAF_ERROR_INVALID_TARGET	The file could not be saved, since it is read-only
ZAF_ERROR_NONE	No error has occurred
ZAF_ERROR_STORAGE_VERSION	The storage file specified is the wrong version, and must be converted

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

FindClose

```
virtual int FindClose(ZafFileInfoStruct &fileInfo) = 0;
```

FindClose() abstractly defines finalizing a find operation begun with FindFirst(). *fileInfo* is the same ZafFileInfoStruct object used by FindFirst() and FindNext(), and memory allocated in *fileInfo* is deleted by FindClose(). FindClose() returns 0 if the operation was successful; otherwise it returns -1.

FindFirst

```
virtual int FindFirst(ZafFileInfoStruct &fileInfo, const
    ZafIChar *searchName, ZafStringID stringID =
    ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0) = 0;
```

FindFirst() abstractly defines initializing a find operation. FindFirst() searches for a file with the search pattern specified by *searchName*, or with the string identification constant (if applicable, such as in ZafStorage) specified by *stringID*, or with the numeric identification constant (if applicable, such as in ZafStorage) specified by *numberID*. *searchName* may contain wildcards appropriate to the file system. For example, in ZafStorage, "?" means any single character, and "*" means any string of characters. *fileInfo* is a ZafFileInfoStruct object allocated by the programmer, and *fileInfo* is initialized by FindFirst(). FindFirst() returns 0 if a file was found; otherwise it returns -1.

FindNext

```
virtual int FindNext(ZafFileInfoStruct &fileInfo, const
    ZafIChar *searchName, ZafStringID stringID =
    ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0) = 0;
```

FindNext() abstractly defines continuing a find operation initiated by FindFirst(). FindNext() searches for the next file with the same values for *searchName*, *stringID*, or *numberID* as were specified in the call to FindFirst().

fileInfo is the same `ZafFileInfoStruct` object passed to `FindFirst()`. `FindNext()` returns 0 if a file was found; otherwise it returns -1.

GetCWD

```
virtual int GetCWD(ZafIChar *pathName, int pathLength) = 0;
```

`GetCWD()` abstractly defines copying the path name of the current directory for a derived file system (without a terminating path separating character) into the buffer specified by *pathName*. *pathName* must be allocated by the programmer, and *pathLength* specifies the number of `ZafIChar` characters in *pathName*. `GetCWD()` returns 0 on success, and -1 on failure. `Error()` is also set to an appropriate value.

MkDir

```
virtual int MkDir(const ZafIChar *pathName, ZafStringID  
    stringID = ZAF_NULLP(ZafIChar), ZafNumberID numberID =  
    0) = 0;
```

`MkDir()` abstractly defines creating a new directory for a derived file system. *pathName* specifies the path name of the directory to be created. *stringID* specifies the string identification constant associated with the directory, if applicable (such as in `ZafStorage`). *numberID* specifies the numeric identification constant associated with the directory, if applicable (such as in `ZafStorage`). `MkDir()` returns 0 on success, and -1 on failure. `Error()` is also set to an appropriate value.

Open

```
virtual ZafFile *Open(const ZafIChar *fileName, const  
    ZafFileMode mode, ZafStringID stringID =  
    ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0) = 0;
```

`Open()` abstractly defines opening the file in the file system specified by *fileName*. *mode* specifies the mode the file is opened with (see the `ZafFile` constructor for more information). *stringID* specifies the string identification constant associated with the file, if applicable (such as in `ZafStorage`). *numberID* specifies the numeric identification constant associated with the file, if applicable (such as in `ZafStorage`). A pointer to the file opened is returned.

parentDirectory- Name

```
static ZafIChar *parentDirectoryName;
```

`parentDirectoryName` is a public static specifying the name of the parent directory in a file system, which is `".."`.

Remove

```
virtual int Remove(const ZafIChar *name) = 0;
```

Remove() abstractly defines deleting the file in the file system specified by *name*. **Remove()** returns 0 on success, and -1 on failure. **Error()** is also set to an appropriate value.

Rename

```
virtual int Rename(const ZafIChar *oldName, const  
    ZafIChar *newName, ZafStringID stringID =  
    ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0) = 0;
```

Rename() abstractly defines renaming the file or directory in the file system specified by *oldName* to *newName*. A non-null *stringID* specifies the string identification constant to be copied to the file or directory, if applicable (such as in **ZafStorage**). A non-zero *numberID* specifies the numeric identification constant to be copied to the file or directory, if applicable (such as in **ZafStorage**). If *oldName* and *newName* are the same, *stringID* and *numberID* are still copied to the file or directory (if not null and 0, respectively). **Rename()** returns 0 on success, and -1 on failure. **Error()** is also set to an appropriate value.

RmDir

```
virtual int RmDir(const ZafIChar *pathName, bool  
    deleteContents = false) = 0;
```

RmDir() abstractly defines deleting a directory for a derived file system. *pathName* specifies the path name of the directory to be deleted. If *deleteContents* is false, **RmDir()** will only delete the directory if it is empty; otherwise, **RmDir()** will delete the contents of the directory along with the directory itself. **RmDir()** returns 0 on success, and -1 on failure. **Error()** is also set to an appropriate value.

rootDirectoryName

```
static ZafIChar *rootDirectoryName;
```

rootDirectoryName is a public static specifying the name of the root directory in a file system, which is "~".

ZafFormatData

[FormattedText](#)

Inheritance	ZafFormatData : ZafData : (ZafElement, ZafNotification)
Declaration	#include <z_fdata.hpp>
Description	<p>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.</p> <p>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.</p>
Constructors	<p>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:</p>

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

FormattedText

```
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, 256);
printf("date - %s\n", buffer);
```

```
ZafIntegerData integer(100);
integer->FormattedText(buffer, 256, "%d");
printf("integer - %s\n", buffer);

ZafStringData string("hello");
string->FormattedText(buffer, 256);
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.

ZafFormattedString

CompressedData	FormatData	SetFormatText
DeleteData	SetCompressedText	SetExpandedText
ExpandedData	SetDeleteText	

Inheritance	ZafFormattedString : ZafString : ZafWindowObject : ZafElement
Declaration	<code>#include <z_fmtstr.hpp></code>
Description	ZafFormattedString is a single-line text object that allows keyboard input using a programmer-defined input format. All strings entered into a ZafFormattedString are validated on a character-by-character basis against a pre-defined input mask. Additional functionality is inherited from ZafString.
Constructors	All ZafFormattedString constructors initialize the member variables associated with an instantiated ZafFormattedString object. The default values set by the ZafFormattedString 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 ZafFormattedString. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafFormattedString

<code>CompressedData()</code>	<code>null</code>
<code>DeleteData()</code>	<code>null</code>
<code>ExpandedData()</code>	<code>null</code>
<code>FormatData()</code>	<code>null</code>

ZafString

<code>LowerCase()</code>	<code>false</code> [†]
<code>OutputFormat()</code>	<code>null</code> [†]
<code>Password()</code>	<code>false</code> [†]
<code>RangeData()</code>	<code>null</code> [†]
<code>UpperCase()</code>	<code>false</code> [†]
<code>VariableName()</code>	<code>false</code> [†]

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_FORMATTED_STRING</code>
<code>ClassName()</code>	<code>"ZafFormattedString"</code>

```
ZafFormattedString(int left, int top, int width, const
    ZafIChar *compressedText, const ZafIChar *format,
    const ZafIChar *deleteText);
```

This constructor is useful in straight-code situations, particularly if the `ZafFormattedString` object is to create, maintain and destroy its own `ZafStringData` objects 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 (see `ZafWindowObject::SetCoordinateType`). *compressedText* is the unformatted value that initially appears in the new `ZafFormattedString` object, *format* is the format “mask”, and *deleteText* specifies the “null” characters to appear in the `ZafFormattedString` when no valid character has been entered. See [CompressedData\(\)](#), [FormatData\(\)](#), and [DeleteData\(\)](#) for more information on these parameters.

```
ZafFormattedString(int left, int top, int width,
    ZafStringData *compressedText, ZafStringData *format,
    ZafStringData *deleteText);
```

This constructor is useful in straight-code situations where `ZafStringData` objects have already been created. For more information on using `ZafStringData` objects, see the chapter on [ZafStringData](#). *left*, *top*, and *width* are the same as the previous constructor.

```
ZafFormattedString(const ZafFormattedString &copy);
```

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

```
ZafFormattedString(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 `ZafFormattedString` creation techniques follow:

```
// Create a sample window with formatted string objects.
ZafWindow *window1 = new ZafWindow(0, 0, 40, 10);
// Create a formatted string and pass in the values directly.
```

```

window1->Add(new ZafFormattedString(1, 1, 20, "8017858900",
    "LNNNLNNNLNNNN", "(...) ...-...."));

// Create string data objects.
ZafStringData *compressData = new ZafStringData("8017858996");
ZafStringData *formatData = new ZafStringData("LNNNLNNNLNNNN");
ZafStringData *deleteData = new ZafStringData("(...) ...-....");
// Create a formatted string that uses the data
// previously created.
ZafFormattedString *phoneNumber = new ZafFormattedString(1, 2,
    20, compressData, formatData, deleteData);
window1->Add(phoneNumber);

// Get the expanded text of the formatted string object.
// expandedText will be "(801) 785-8900".
ZafIChar *expandedText = phoneNumber->ExpandedData()->Text();

```

Destructor

```
virtual ~ZafFormattedString(void);
```

The destructor is used to free the memory associated with a ZafFormattedString 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 ZafFormattedString 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.

CompressedData

```

ZafStringData *CompressedData(void) const;
virtual ZafError SetCompressedData(ZafStringData
    *compressedData);

```

SetCompressedText

```

virtual ZafError SetCompressedText(const ZafIChar
    *compressedText);

```

The CompressedData() object stores the actual raw data without the format literals (see [FormatData\(\)](#) for more information on format literals). For example, if a formatted phone number were “(801) 785-8900” the CompressedData would probably be “8017858900”.

CompressedData() may be shared among several objects, or it may belong to a single ZafFormattedString object. If shared, all the associated objects will be notified when the CompressedData() changes. SetCompressedData() may be used to associate a CompressedData() object with a ZafFormattedString object. For more information on data sharing in ZAF, see [ZafDataManager](#). SetCompressedData() will delete the previous CompressedData() object if it is Destroyable() and no other object uses it.

SetCompressedText() may be used instead of SetCompressedData(). In this case *compressedText* is placed into the existing CompressedData() member of ZafFormattedString.

The return value for CompressedData() is a pointer to the CompressedData() object associated with the ZafFormattedString object. The return value for SetCompressedData() and SetCompressedText() is normally ZAF_ERROR_NONE.

DeleteData

```
ZafStringData *DeleteData(void) const;
```

```
virtual ZafError SetDeleteData(ZafStringData  
    *deleteData);
```

SetDeleteText

```
virtual ZafError SetDeleteText(const ZafIChar  
    *deleteText);
```

The DeleteData() object stores the data to be placed in character positions for which no CompressedData() character is present. DeleteData() also stores the characters to be used as literal, uneditable characters (specified in the corresponding FormatData() character by 'L'). See the constructor code snippet for an example.

DeleteData() may be shared by multiple objects, or it may belong to a single ZafFormattedString. If shared, all the associated objects will be notified when DeleteData() changes. For more information on data sharing in ZAF, see [ZafDataManager](#). SetDeleteData() will delete the previous DeleteData() object if it is Destroyable() and no other object uses it.

SetDeleteText() may be used instead of SetDeleteData(). In this case, *deleteText* is placed into the existing DeleteData() member of ZafFormattedString.

The return value for DeleteData() is a pointer to the DeleteData() object associated with the ZafFormattedString object. The return value for SetDeleteData() and SetDeleteText() is normally ZAF_ERROR_NONE.

ExpandedData

```
ZafStringData *ExpandedData(void) const;
```

```
virtual ZafError SetExpandedData(ZafStringData  
    *expandedData);
```


SetExpandedText `virtual ZafError SetExpandedText(const ZafIChar
 *expandedText);`

The ExpandedData() object stores the actual data displayed, including format literals and applicable DeleteData(). (see [FormatData\(\)](#) for more information on format literals). The constructor code snippet shows an example.

ExpandedData() piece may be shared by multiple objects, or it may belong to a single ZafFormattedString object. If shared, all the associated objects will be notified when ExpandedData() changes. SetExpandedData() may be used to associate a ExpandedData() object with a ZafFormattedString object. For more information on data sharing in ZAF, see [ZafDataManager](#). SetExpandedData() will delete the previous ExpandedData() object if it is Destroyable() and no other object uses it.

SetExpandedText() may be used instead of SetExpandedData(). In this case, expandedText is placed into the existing ExpandedData() member of ZafFormattedString.

The return value for ExpandedData() is a pointer to the ExpandedData() object associated with the ZafFormattedString object. The return value for SetExpandedData() and SetExpandedText() is normally ZAF_ERROR_NONE.

FormatData `ZafStringData *FormatData(void) const;
virtual ZafError SetFormatData(ZafStringData
 *formatData);`

SetFormatText `virtual ZafError SetFormatText(const ZafIChar
 *formatText);`

FormatData() stores the formatting data including format literals and validation options. This formatData provides the input mask used to format and validate all input.

For example, to allow entry in a US phone number format, FormatData() might be set to

```
"LNNNLLNNNLLNNNN aaaa"
```

and DeleteData() set to

```
"(    )       -               " .
```

If the user then typed the characters

```
"123456abc7890x321"
```

The final, formatted text (ExpandedData()) would be

```
"(123) 456-7890 x321".
```

And the final CompressedData() would be

"1234567890x321" .

Note that the embedded "abc" keystrokes would be seen as invalid due to the FormatData() and therefore discarded.

Valid characters in the FormatData() input mask include the following:

FormatData() option	Description
a	Allows a space (' ') or any alphabetic character ('a' through 'z' or 'A' through 'Z')
A	Same as the 'a' option except that a lower-case character is automatically converted to upper-case
c	Allows a space (' '), any numeric digit ('0' through '9'), or any alphabetic character ('a' through 'z' or 'A' through 'Z')
C	Same as the 'c' option except that a lower-case character is automatically converted to upper-case
L	Uses the literal character specified in DeleteData(), and skips this position while the end user types
N	Allows any numeric digit
x	Allows any printable character
X	Same as the 'x' option except that a lower-case character is automatically converted to upper-case

FormatData() may be shared by multiple objects, or it may belong to a single ZafFormattedString. If shared, all the associated objects will be notified when the FormatData() changes. SetFormatData() may be used to associate a FormatData() object with a ZafFormattedString object. For more information on data sharing in ZAF, see [ZafDataManager](#). SetFormatData() will delete the previous FormatData() object if it is Destroyable() and no other object uses it.

SetFormatText() may be used instead of SetFormatData(). In this case *format-Text* is placed into the existing FormatData() member of ZafFormattedString.

The return value for FormatData() is a pointer to the FormatData() object associated with the ZafFormattedString object. The return value for SetFormatData() and SetFormatText() is normally ZAF_ERROR_NONE.

ZafGdiDisplay

Inheritance ZafGdiDisplay : [ZafDisplay](#)

Declaration `#include <w_gdidsp.hpp>`

Description ZafGdiDisplay defines the basic functionality necessary to interface with Microsoft Windows display devices. For Microsoft Windows only, ZafScreenDisplay and ZafPrinter are both derived from ZafGdiDisplay. All the functionality documented for ZafScreenDisplay and ZafPrinter are portable, and some of this functionality is provided for Microsoft Windows transparently through ZafGdiDisplay. See the base class [ZafDisplay](#) for complete descriptions of the display functions provided by ZafScreenDisplay and ZafPrinter.

ZafGeometryManager

	Event
Inheritance	<code>ZafGeometryManager : (ZafWindowObject : ZafElement), ZafList</code>
Declaration	<code>#include <z_gmgr.hpp></code>
Description	<p>ZafGeometryManager provides support for non-absolute positioning of child objects. For example, an object may be positioned relative to an edge of another object; or an object may size itself relative to the size of a window, perhaps constrained to a minimum and maximum size.</p> <p>The programmer may specify combinations of constraints and attachments for any child object by adding ZafAttachment, ZafDimensionConstraint, and ZafRelativeConstraint objects to the ZafGeometryManager for the window. See ZafConstraint, ZafAttachment, ZafDimensionConstraint, and ZafRelativeConstraint for more information.</p>
Constructors	<p>All ZafGeometryManager constructors initialize the member variables associated with an instantiated ZafGeometryManager object. The default values set by the ZafGeometryManager 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 ZafGeometryManager. “†” Indicates a blocking function that prevents changes to the attribute in this class.</p>

Member Initializations

ZafWindowObject

<code>AcceptDrop()</code>	<code>false[†]</code>
<code>AutomaticUpdate()</code>	<code>true[†]</code>
<code>BackgroundColor()</code>	<code>ZAF_CLR_NULL[†]</code>
<code>Bordered()</code>	<code>false[†]</code>
<code>CoordinateType()</code>	<code>ZAF_PIXEL[†]</code>
<code>CopyDraggable()</code>	<code>false[†]</code>
<code>Disabled()</code>	<code>true[†]</code>
<code>Focus()</code>	<code>false[†]</code>
<code>Font()</code>	<code>ZAF_FNT_NULL[†]</code>
<code>HelpContext()</code>	<code>null[†]</code>
<code>HelpObjectTip()</code>	<code>null[†]</code>
<code>LinkDraggable()</code>	<code>false[†]</code>
<code>MoveDraggable()</code>	<code>false[†]</code>

Member Initializations

Noncurrent()	true [†]
OSDraw()	false [†]
ParentDrawBorder()	false [†]
ParentDrawFocus()	false [†]
ParentPalette()	false [†]
QuickTip()	null [†]
RegionType()	ZAF_AVAILABLE_REGION [†]
Selected()	false [†]
SupportObject()	true [†]
TextColor()	ZAF_CLR_NULL [†]
UserFunction()	null [†]
UserPalette()	null [†]
Visible()	false [†]

ZafElement

ClassID()	ID_ZAF_GEOMETRY_MANAGER
ClassName()	"ZafGeometryManager"
NumberID()	ZAF_NUMID_GEOMETRY
StringID()	"ZAF_NUMID_GEOMETRY"

ZafGeometryManager(void);

This constructor is useful in straight-code situations, and creates a ZafGeometryManager object to be added to a window.

ZafGeometryManager(const ZafGeometryManager ©);

The copy constructor creates a new ZafGeometryManager object and initializes its data from *copy*.

ZafGeometryManager(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 geometry manager follows:

```
// Create a status bar with geometry-managed children.
ZafStatusBar *stat = new ZafStatusBar(0, 0, 0, 1);
```

```
ZafString *string = new ZafString(0, 0, 15, new
    ZafStringData("String"));
stat->Add(string);
ZafTime *time = new ZafTime(15, 0, 15, new ZafTimeData);
stat->Add(time);

// Time field will remain at the right side of the status bar.
ZafAttachment *attach = new ZafAttachment(time, ZAF_ATCF_RIGHT);
attach->SetOffset(0);

// Create the geometry manager and add the first constraint.
ZafGeometryManager *geo = new ZafGeometryManager;
geo->Add(attach);

// String field will occupy the remaining space.
attach = new ZafAttachment(string, ZAF_ATCF_RIGHT);
attach->SetStretch(true);
attach->SetReference(time);
attach->SetOppositeSide(true);
attach->SetOffset(1);

// Add the second constraint to the geometry manager.
geo->Add(attach);

// Add the geometry manager to the status bar.
stat->Add(geo);
```

Destructor

```
virtual ~ZafGeometryManager(void);
```

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

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

Members

Event

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

This overloaded function handles all events sent to the ZafGeometryManager object. The following events are handled:

ZafEventType	Description
S_ADD_OBJECT	causes event.windowObject to be added to the geometry manager's constraint list

ZafEventType	Description
S_COMPUTE_SIZE	causes the geometry manager to recompute its size and to notify all its constraints that the parent window has changed size by passing the S_COMPUTE_SIZE message to them
S_INITIALIZE	causes the geometry manager to initialize itself and to pass the S_INITIALIZE message to all its constraints
S_SUBTRACT_OBJECT	causes event.windowObject to be removed from the geometry manager's constraint list

ZafGroup

AutoSelect	HotKeyIndex
HotKeyChar	StringData

Inheritance ZafGroup : ZafWindow : (ZafWindowObject : ZafElement),
 ZafList

Declaration #include <z_group.hpp>

Description ZafGroup is a container object used to logically group other objects. A ZafGroup may have a title and or a border. ZafGroups are typically used to contain controls such as radio buttons that must act as a set and are therefore more sophisticated than simple line boxes. Once “inside” a ZafGroup, common key-strokes may act somewhat differently than in other contexts. For example, arrow keys may move between controls in a ZafGroup, and the tab key moves away from the group.

Constructors All ZafGroup constructors initialize the member variables associated with an instantiated ZafGroup object. The default values set by the ZafGroup 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 ZafGroup. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafGroup

AutoSelect()	false
HotKeyChar()	'\0'
HotKeyIndex()	-1
StringData()	null

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Modal()	false [†]
Moveable()	false [†]
Sizeable()	false [†]
Temporary()	false [†]

Member Initializations

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	true

ZafElement

ClassID()	ID_ZAF_GROUP
ClassName()	"ZafGroup"

```
ZafGroup(int left, int top, int width, int height, const
          ZafIChar *text);
```

This constructor is useful in straight-code situations and allows the ZafGroup object to create, maintain and destroy its own ZafStringData object. *left*, *top*, *width*, and *height* specify the position and size on a parent window. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. See ZafWindowObject::Set[CoordinateType](#)() for more information. *text* specifies the string to initially appear in the new ZafGroup object.

```
ZafGroup(int left, int top, int width, int height,
          ZafStringData *stringData);
```

This constructor is useful in straight-code situations where a ZafStringData object has already been created. This constructor could be used when manually maintaining a ZafStringData object, rather than having the ZafGroup class create and maintain the data object automatically. (See [ZafStringData](#)). See the previous constructor for a description of *left*, *top*, *width* and *height*.

```
ZafGroup(const ZafGroup &copy);
```

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

```
ZafGroup(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 code shows how to create a `ZafGroup` object.

```
// Create a sample window with a group of radio buttons.
ZafWindow *window = 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, "Choice 1",
    ZAF_NULLP(ZafBitmapData), ZAF_RADIO_BUTTON));
group->Add(new ZafButton(1, 1, 25, 1, "Choice 2",
    ZAF_NULLP(ZafBitmapData), ZAF_RADIO_BUTTON));
group->Add(new ZafButton(1, 2, 25, 1, "Choice 3",
    ZAF_NULLP(ZafBitmapData), ZAF_RADIO_BUTTON));
group->Add(new ZafButton(1, 3, 25, 1, "Choice 4",
    ZAF_NULLP(ZafBitmapData), ZAF_RADIO_BUTTON));

// Finally, add the group of radio buttons to the window.
window->Add(group);
```

Destructor

```
virtual ~ZafGroup(void);
```

The destructor is used to free the memory associated with a `ZafGroup` 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 `ZafGroup` 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.

AutoSelect

```
bool AutoSelect(void) const;
virtual bool SetAutoSelect(bool autoSelect);
```

If `AutoSelect()` is true, the first group item in a `ZAF_SINGLE_SELECTION` group becomes selected when the group initially appears on screen. `AutoSelect()` also causes a group item to be selected when it receives focus. The default value of this attribute is false, but `SetAutoSelect()` may be called to change it.

HotKeyChar
HotKeyIndex

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

When typed with a modifier key (such as <ALT> or <Command>) the *hotKeyChar* causes a group to receive keyboard focus. The hot key *index* is the zero-based index specifying which character to visually distinguish in the group's title.

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 group, and the default value of `HotKeyIndex()` is -1, indicating that no character is to be visually distinguished as the hot key character.

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

```
// Create a group with a hot key.
ZafGroup *group = new ZafGroup(1, 1, 12, 4, "Group");
group->SetHotKey('G', 0);
```

StringData

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

The `StringData()` object is where the textual data for the title is stored for the `ZafGroup` object. The `StringData()` piece may be shared by multiple “users”, or it may belong to a single object. If shared, all the associated objects will be notified when the `StringData()` changes. `SetStringData()` may be used to associate a `StringData()` object with an object. For more information on data sharing in ZAF, see [ZafDataManager](#). `SetStringData()` will delete the previous `StringData()` object if it is `Destroyable()` and no other object uses it.

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

The function `SetText()`, inherited from `ZafWindowObject`, may be used to set the title on the group without the programmer having to create a `ZafStringData` object.

ZafHelpStub

[DisplayHelp](#)

Inheritance	Root class
Declaration	<code>#include <z_help.hpp></code>
Description	<p>ZafHelpStub serves solely as a base class for ZafHelpSystem, so that ZafHelpSystem and its base classes are not linked into a program that does not make use of them. ZafHelpStub declares a pure virtual function <code>DisplayHelp()</code> that is defined in ZafHelpSystem. The programmer should normally not derive from this class.</p>
Constructor	<p><code>ZafHelpStub(void);</code></p> <p>This constructor creates a ZafHelpStub object. Since ZafHelpStub is a virtual class, the programmer will normally not call this constructor.</p>
Destructor	<p><code>virtual ~ZafHelpStub(void);</code></p> <p>The destructor is used to free the memory associated with a ZafHelpStub object. Since ZafHelpStub is a virtual class, the programmer will not normally call this destructor.</p>
Members	<p><code>virtual void DisplayHelp(const ZafIChar *name) = 0;</code></p> <p><i>DisplayHelp</i></p> <p><code>DisplayHelp()</code> is a pure virtual function. ZafHelpSystem overloads this function to provide basic help display functionality</p>

ZafHelpSystem

	DisplayHelp	Event	ResetStorage
Inheritance	<code>ZafHelpSystem : ZafHelpStub, ZafWindow : (ZafWindowObject : ZafElement), ZafList</code>		
Declaration	<code>#include <z_help.hpp></code>		
Description	<p>ZafHelpSystem is ZAF’s built-in support for context-sensitive help. If a ZafHelpSystem has been instantiated (see the code snippet under the constructors), and the end user types the help key (such as <F1> or <Command-?>), the ZafHelpSystem is presented. If the window object that has keyboard focus has been assigned a help context via ZafWindowObject::HelpContext(), that help context is presented; otherwise, a list of help contexts available is presented.</p> <p>ZafHelpSystem presents a help context from the user-specified data file in a top-level window with a scrollable text field, and provides a button that when clicked causes an index of all available help contexts to be displayed. When the index of help contexts is presented, the end user may choose any of the help contexts with the keyboard or the mouse and select it to be displayed.</p> <p>Advanced programmers may derive from ZafHelpSystem to tie into native or third-party engines.</p>		
Constructors	<p>All ZafHelpSystem constructors initialize the member variables associated with an instantiated ZafHelpSystem object. The default values set by the ZafHelpSystem and its base class constructors follow, if they differ from those set by the base class constructor.</p>		
<hr/>			
Member Initializations			
<hr/>			
ZafWindow			
	<code>Destroyable()</code>	<code>false</code>	
ZafElement			
	<code>ClassID()</code>	<code>ID_ZAF_HELP_SYSTEM</code>	
	<code>ClassName()</code>	<code>"ZafHelpSystem"</code>	
<hr/>			

```
ZafHelpSystem(ZafFileSystem *helpStorage);
```

This constructor creates a ZafHelpSystem object, initializes information used by ZafHelpSystem, and sets the global zafHelpSystem. There should only be one ZafHelpSystem instantiated during the life of an application. *helpStorage* specifies an instantiated data file that contains the help contexts to be used by ZafHelpSystem.

```
ZafHelpSystem(const ZafHelpSystem &copy);
```

The copy constructor creates a new ZafHelpSystem object and initializes its data from copy.

The following code shows how to create a ZafHelpSystem object.

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

    // Open the data file.
    ZafStorage *storage = new ZafStorage("main.znc",
        ZAF_FILE_READWRITE);
    // Create the help system.
    if (!storage->Error())
        zafHelpSystem = new ZafHelpSystem(storage,
            ZAF_NULLP(ZafIChar));

    . . .

    // Get user input.
    Control();

    // Clean up.
    if (zafHelpSystem)
        delete zafHelpSystem;
    if (storage)
        delete storage;

    // Return success.
    return (0);
}
```

Destructor `virtual ~ZafHelpSystem(void);`

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

Since the ZafHelpSystem constructor sets Destroyable() to false, the programmer must destroy a ZafHelpSystem object when appropriate to the application. Generally, the ZafHelpSystem should be deleted as the application is shutting down, as seen in the code snippet above.

Members `virtual void DisplayHelp(const ZafIChar *name);`

DisplayHelp
DisplayHelp() causes the help system window to appear. If it is minimized, then it is first restored. If it is already open, it is brought to the front of all the other windows on the window manager. name is the name of the help context that will be displayed in the help system window.

Generally, ZAF calls DisplayHelp() automatically with the appropriate help context name when the end user requests help, but the programmer may call it at any time. The end user may resize, minimize, or close the help system window, and it may be left open in the background if the user selects another window.

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

This overloaded function handles all events sent to ZafHelpSystem, either by processing the events itself, or by passing them to ZafWindow::Event() for base class processing. Events (actually "requests") handled directly by ZafHelpSystem follow:

Event	Description
S_HLP_CLOSE	causes the help system window to close
S_HLP_SHOW_INDEX	causes the help system window to display the index of all available help contexts
S_HLP_SHOW_TOPIC	causes the help system window to display event.windowObject's help context
S_HLP_SELECT_TOPIC	causes the help system window to display the help context corresponding to the name displayed in the help system window's string field
S_HLP_UPDATE_NAME	causes the help system window to set its string field corresponding to its selected list item

See [ZafWindow](#) for more information.

ResetStorage

```
virtual void ResetStorage(ZafFileSystem *helpStorage);
```

ResetStorage() causes the help system to find help contexts in a different storage file. **helpStorage** specifies the instantiated data file that contains the help contexts to be used by **ZafHelpSystem**.

ResetStorage() is particularly useful when changing languages for help contexts. The following code snippet shows how to use this function:

```
// Use the spanish help contexts instead of english.
ZafStorage *oldHelpStorage = myHelpStorage;
myHelpStorage = new ZafStorage("help_es.znc",
    ZAF_FILE_READWRITE);

if (!myHelpStorage->Error())
{
    // Reset the help system storage.
    myHelpSystem->ResetStorage(myHelpStorage);
    delete oldHelpStorage;
}
```

ZafHelpTips

Event	InitialDelay	SetUserPalette
HelpObject	NewHelpTipDelay	
HelpTipsType	Poll	

Inheritance ZafHelpTips : [ZafDevice](#) : [ZafElement](#)

Declaration `#include <z_https.hpp>`

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](#).

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()	500 (milliseconds)
NewHelpTipDelay()	500 (milliseconds)

ZafDevice

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

ZafElement

ClassID()	ID_ZAF_HELPTIPS
ClassName()	"ZafHelpTips"
NumberID()	ID_ZAF_HELPTIPS
StringID()	"ZafHelpTips"

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

This constructor is used to instantiate a ZafHelpTips object to be added to a ZafEventManager object. *helpTipsType* 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

Event

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().

HelpObject

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_HELP TIPS);
    }
    break;
```

HelpTipsType

```
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_HELP TIPS_QUICKTIP but the user may call SetHelpTipsType() to change it. Here are the possible types of help tips:

ZafHelpTipsType	Description
ZAF_HELP TIPS_QUICKTIP	Causes ZafHelpTips display only quick tips.
ZAF_HELP TIPS_HELP OBJECT	Causes ZafHelpTips to display only help tips.
ZAF_HELP TIPS_BOTH	Causes ZafHelpTips to manage both quick tips and help tips.

InitialDelay

```
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.

NewHelpTipDelay

```
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.

Poll `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.

SetUserPalette `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	SelectionType	SetTextColor
CellHeight	SetBackgroundColor	
CellWidth	SetFont	

Inheritance `ZafHzList : ZafWindow : (ZafWindowObject : ZafElement), ZafList`

Declaration `#include <z_hlist.hpp>`

Description 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, and list children may not be Noncurrent().

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

<code>AutoSortData()</code>	<code>false</code>
<code>CellHeight()</code>	<code>user-supplied parameter</code>
<code>CellWidth()</code>	<code>user-supplied parameter</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>Modal()</code>	<code>false[†]</code>
<code>Moveable()</code>	<code>false[†]</code>
<code>NormalHotKeys()</code>	<code>false[†]</code>
<code>Sizeable()</code>	<code>false[†]</code>

ZafWindowObject

<code>Bordered()</code>	<code>true</code>
<code>ZafElement</code>	

Member Initializations

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,  
    ZAF_NULLP(ZafScrollData), ZAF_HORIZONTAL_SCROLL));  
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, ZAF_NULLP(ZafIChar),
    bitmap1));
hList2->Add(new ZafButton(0, 0, 20, 1, ZAF_NULLP(ZafIChar),
    bitmap2));
hList2->Add(new ZafButton(0, 0, 20, 1, ZAF_NULLP(ZafIChar),
    bitmap3));
hList2->Add(new ZafButton(0, 0, 20, 1, ZAF_NULLP(ZafIChar),
    bitmap4));

// 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.

AutoSortData

```
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 SetCom-

pareFunction() 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.

SetBackground-Color

```
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.

*CellHeight
CellWidth*

```
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.

SelectionType

```
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.

SelectionType()	Description
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.

SetFont

```
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.

SetTextColor

```
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.

ZafI18nData

AddStaticModule	I18nName	SubtractStaticModule
CountryCodeToZafLocale	OSI18nName	ZafLanguageToZafLocale
I18nAllocate	OSLanguageToZafLanguage	ZafLocaleToZafLanguage
I18nFree	ResetI18n	

Inheritance ZafI18nData : ZafData : ZafNotification, ZafElement

Declaration #include <z_idata.hpp>

Description ZafI18nData is the base class for the three internationalization classes: [ZafCodeSetData](#), [ZafLanguageData](#), and [ZafLocaleData](#). It is not intended to be instantiated directly and has no public constructor. ZafI18nData provides a standard interface used to manipulate the individual internationalization components. [ZafLanguageManager](#) is also a core internationalization class.

Programmers will typically use only the ResetI18n() member of this class, but a thorough understanding of ZafI18nData is essential to developers intending to manipulate the internationalization capabilities of Zinc Application Framework.

ZafI18nData performs many functions, but three are key:

- Determine the proper language, locale, and codeset (mapping) during application initialization.
- Initialize global ZafCodeSetData, ZafLocaleData, and ZafLanguageManager pointers.
- Maintain global internationalization information used by other objects.

I18n initialization process ZAF initializes internationalization components during library initialization prior to calling ZafApplication::Main(). These components may be reinitialized by the programmer later. The algorithm used to initialize the internationalization components follows:

- Identify i18n context
1. Get the internationalization identification information using the following process:
 - Obtain the ZAF_LANG environment variable if it exists;
 - or obtain information from the operating system if possible;
 - or finally use the values compiled into the application if no better alternative is available.

A complete internationalization specification takes the form “en_US.437” and indicates the language (en=English), country (US=United States), and codeset (437=map local codepage 437 to/from ISO or Unicode). This string may be directly replaced using [SetI18nName\(\)](#) or [ResetI18n\(\)](#).

- | | |
|----------------------------------|--|
| Initialize global variables | 2. Each of the individual internationalization components is initialized and global variables are set. Important variables include <code>zafI18nStorage</code> , <code>zafLocaleData</code> , <code>zafCodeSetData</code> , and <code>zafLanguageManager</code> . |
| Load or build locale information | 3. A complete locale definition is built using the following process (based on the country code found in step 1): <ul style="list-style-type: none"> • <code>I18N.ZNC</code> is opened, if possible, and the correct locale information loaded; • or the compiled-in locale defaults are used to initialize the locale and individual locale members replaced with as much information as can be obtained from the operating system. |
| Load message strings | 4. International message strings are loaded by the <code>ZafLanguageManager</code> as they are requested by <code>ZafWindowObjects</code> . This portion of <code>i18n</code> initialization does not happen before <code>ZafApplication::Main()</code> but rather on demand as the application runs. For more information on this process, see ZafLanguageManager . |

Constructors

`ZafI18nData` constructors are all protected. See the constructors for [ZafCodeSetData](#), [ZafLanguageData](#), and [ZafLocaleData](#).

`ZafI18nData` is initially constructed inside ZAF prior to `ZafApplication::Main`. Programmers can reset this initialization using `ResetI18n()` as shown below.

```
int ZafApplication::Main()
{
    LinkMain();

    // Reinitialize for specified language, locale, and codeset.
    ZafI18nData::ResetI18n("FR_fr.ISO8859-1");

    zafWindowManager->Add(new HellowWindow());

    Control();
}
```

Destructor

```
virtual ~ZafI18nData(void);
```

The destructor is used to free the memory associated with the `ZafI18nData` object. This destructor chains to the `ZafData`, `ZafNotification`, and `ZafElement` destructors.

Members

AddStaticModule

```
static bool AddStaticModule(ZafFreeModule freeModule,
    ZafAllocateModule allocateModule =
    ZAF_NULLF(ZafAllocateModule), ZafClassID category =
    ID_DEFAULT);
```

SubtractStaticModule

```
static bool SubtractStaticModule(ZafFreeModule
    freeModule, ZafAllocateModule allocateModule =
    ZAF_NULLF(ZafAllocateModule));
```

`AddStaticModule()` is a mechanism for freeing and allocating static data at program runtime. It is used internally by ZAF, and is not intended to be used by the programmer. *freeModule* is a pointer to a function that is to be called during program termination or when internationalization is reset. *allocateModule* is a pointer to a function that is to be called to allocate static data. *category* specifies the class identifier associated with the static data.

`SubtractStaticModule()` removes support for *freeModule* and *allocateModule* after they have been passed to `AddStaticModule()`. The definition of `ZafAllocateModule` and `ZafFreeModule` are shown below:

```
typedef void (*ZafAllocateModule)(void);
typedef void (*ZafFreeModule)(bool);
```

CountryCodeToZafLocale

```
static const ZafIChar *CountryCodeToZafLocale(int
    countryCode);
```

This function maps *countryCode* to a string description of the locale. The country code passed in depends on the operating system that the software is running on. This function is used by `OSI18nName()`.

OSLanguageToZafLanguage

```
static const ZafIChar *OSLanguageToZafLanguage(const
    ZafIChar *osLanguage);
```

On systems that need it, this function maps *osLanguage* to a string describing the language being used. *osLanguage* is a system dependent string. This function is used by `OSI18nName()`.

ZafLocaleToZafLanguage

```
static const ZafIChar *ZafLocaleToZafLanguage(const
    ZafIChar *zafLocale);
```

This function maps *zafLocale* to the appropriate language string. *zafLocale* is a string describing the location the program is running in (e.g. Spain or Germany). Two character ISO country codes are valid input for *zafLocale*.

ZafLanguageTo-ZafLocale

```
static const ZafIChar *ZafLanguageToZafLocale(const
    ZafIChar *zafLanguage);
```

This function maps *zafLanguage* to the appropriate locale string. *zafLanguage* is a string describing the language of the operating system (e.g. Spanish or German). Two character ISO language codes are valid input for *zafLanguage*.

OSI18nName

```
static const ZafIChar *OSI18nName(void);
```

This function determines the internationalization name for the current system. The name will take the form language[_locale[.codeset]]. For instance the string "es_ES.850" means that the language is Spanish, the locale is Spain, and the codeset is 850.

I18nName

```
static const ZafIChar *I18nName(void);
static ZafError SetI18nName(const ZafIChar *i18nName);
static ZafError SetI18nName(const ZafIChar *languageName,
    const ZafIChar *localeName, const ZafIChar
    *codeSetName);
```

I18nName() allows the user to query the program about which internationalization settings it is running with. *SetI18nName* allows the user to set the internationalization components of the program using *i18nName*. *i18nName* should be of the form language[_locale[.codeset]]. These functions use the ISO string form to denote the internationalization settings. For more information on the form of the string used see *OSI18nName*().

The second *SetI18nName*() method initializes the internationalization components. *codeSetName* should be a string describing the codeset that is required. The string pointers are passed to the corresponding initialization routine for each component. If the pointer is NULL the component will initialize with the compiled-in defaults.

I18nAllocate

```
static void I18nAllocate(const ZafIChar *i18nName =
    ZAF_NULLP(ZafIChar));
```

This function allocates the internationalization components. If *i18nName* is NULL then the function queries the system to determine the internationalization name. If a string is passed in, it must be of the form lan-

guage[_locale[.codeset]]. See the [OSI18nName\(\)](#) section for more information.

I18nFree

```
static void I18nFree(bool globalRequest = false);
```

This function frees the global internationalization components, if globalRequest is true.

ResetI18n

```
static const ZafIChar *ResetI18n(const ZafIChar *newName  
    = ZAF_NULLP( ZafIChar ) );
```

This function checks if the *newName* is different than the current internationalization settings. If the settings are different it frees and reallocate the internationalization components using *newName*. If *newName* is NULL then the function queries the system to determine the internationalization name to use. If a string is passed in it should be of the form language[_locale[.codeset]]. See the [OSI18nName\(\)](#) section of this chapter for more information. An example of a typical use for ResetI18n() is shown in the [Constructors](#) section of this chapter.

ZafIcon

	IconData	IconType	SetIconImage
Inheritance	ZafIcon : ZafButton : ZafWindowObject : ZafElement		
Declaration	#include <z_icon1.hpp>		
Description	ZafIcon displays a native icon (miniature bitmap). ZafIcon is used to display icons representing minimized windows, if supported by the environment. Where the environment provides a system default icon image, ZafIcon will use it. Most ZafIcon functionality is inherited from its base class, ZafButton.		
Constructors	All ZafIcon constructors initialize the member variables associated with an instantiated ZafIcon object. The default values set by the ZafIcon 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 ZafIcon. “†” Indicates a blocking function that prevents changes to the attribute in this class.		

Member Initializations

ZafIcon

IconData()	null
IconType()	ZAF_NATIVE_ICON

ZafButton

AllowDefault()	false [†]
AllowToggling()	false [†]
AutoRepeatSelection()	false [†]
BitmapData()	null [†]
ButtonType()	ZAF_FLAT_BUTTON [†]
Depth()	0 (2 in Motif) [†]
HotKeyChar()	0 [†]
HotKeyIndex()	-1 [†]
SelectOnDoubleClick()	true

ZafWindowObject

Bordered()	true (only in Motif)
RegionType()	ZAF_INSIDE_REGION [†]
SupportObject()	true (only if ZAF_MINIMIZE_ICON)

ZafElement

ClassID()	ID_ZAF_ICON
-----------	-------------

Member Initializations

<code>ClassName()</code>	<code>"ZafIcon"</code>
<code>NumberID()</code>	<code>ZAF_NUMID_MIN_ICON (if ZAF_MINIMIZE_ICON)</code>
<code>StringID()</code>	<code>"ZAF_NUMID_MIN_ICON" (if ZAF_MINIMIZE_ICON)</code>

```
ZafIcon(int left, int top, const ZafIChar *text,  
        ZafIconData *iconData = ZAF_NULLP(ZafIconData),  
        ZafIconType iconType = ZAF_NATIVE_ICON);
```

This constructor is useful in straight-code situations, particularly if you wish the `ZafIcon` object to create, maintain and destroy its own `ZafStringData` object automatically. *left* and *top* specify the position where the left and top of the icon will be placed on its parent. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. *text* specifies the string to be displayed under the icon image.

iconData specifies the icon to be displayed on the `ZafIcon` object. *iconType* specifies the type of icon to be created. (See [IconType\(\)](#) for more information.)

```
ZafIcon(int left, int top, ZafStringData *title,  
        ZafIconData *iconData = ZAF_NULLP(ZafIconData),  
        ZafIconType iconType = ZAF_NATIVE_ICON);
```

This constructor is also useful in straight-code situations, particularly if you have already created a `ZafStringData` object to be associated with the icon. (See [ZafStringData](#) for more information.) *title* specifies the string data object to be associated with the icon. Either *title* or *iconData* may be null, to provide an image-only or a string-only icon. Otherwise, this constructor is the same as the first.

iconData may be shared by many `ZafIcons` to conserve system resources or ensure consistency. See [ZafDataManager](#) for information on sharing data objects.

```
ZafIcon(const ZafIcon &copy);
```

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

```
ZafIcon(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 `ZafIcon` object creation techniques.

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

// Add the icons to the window.
extern ZafIconData *iconData;
window1->Add(new ZafIcon(1, 4, ZAF_NULLP(ZafStringData),
        iconData));
// Add a hand icon.
ZafIcon *handIcon = new ZafIcon(20, 4, "Stop",
        ZAF_NULLP(ZafIconData));
handIcon->SetIconImage(ZAF_HAND_ICON);
window1->Add(handIcon);
...
// Create a sample window with a minimize icon.
ZafWindow *window2 = new ZafWindow(10, 10, 40, 10);
extern ZafIconData *windowIconData;
window2->Add(new ZafIcon(1, 4, ZAF_NULLP(ZafStringData),
        windowIconData, ZAF_MINIMIZE_ICON));
```

Destructor

```
virtual ~ZafIcon(void);
```

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

Generally, the programmer will not directly destroy a `ZafIcon` 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.

IconData

```
ZafIconData *IconData(void) const;
```

```
virtual ZafError SetIconData(ZafIconData *iconData);
```

The IconData() object is where the actual image data is stored. The IconData() may be shared among several ZafIcon objects (perhaps to save memory by allowing many icon objects to use the same image data), or it may belong to a single ZafIcon object. If shared among several ZafIcon objects, all the associated ZafIcon objects will be updated when the IconData() changes. SetIconData() may be used to associate an IconData() object with a ZafIcon object. For more information on data sharing in ZAF, see [ZafDataManager](#). SetIconData() will delete the previous IconData() object if it is Destroyable() and no other object uses it.

The return value for IconData() is a pointer to the IconData() object associated with the ZafIcon object. The return value for SetIconData() is normally ZAF_ERROR_NONE.

IconType

```
ZafIconType IconType(void) const;  
virtual ZafIconType SetIconType(ZafIconType iconType);
```

IconType() specifies a ZafIcon object type. The type is used to control platform-specific and context-specific display behavior. The default value of this attribute is ZAF_NATIVE_ICON, but the user may call SetIconType() to change it. Possible icon types include:

IconType()	Description
ZAF_NATIVE_ICON	Creates a native icon object (may look different from environment to environment)
ZAF_MINIMIZE_ICON	Creates a minimize icon object for a window

SetIconImage

```
virtual ZafIconData *SetIconImage(ZafIconImage  
    iconImage);
```

ZAF provides several icon images for use with applications, and where available, ZafIcon may use a system default icon image. The programmer may call SetIconImage() to utilize a default image provided by ZAF or the system. Supplied icon images include:

ZafIconImage	Description
ZAF_APPLICATION_ICON	Default application icon
ZAF_ASTERISK_ICON	Asterisk icon, commonly used for note messages

ZaflconImage	Description
ZAF_EXCLAMATION_ICON	Exclamation icon, commonly used for warning messages
ZAF_HAND_ICON	Hand or stop icon, commonly used for error messages
ZAF_QUESTION_ICON	Question icon, commonly used for help messages

ZafIconData

Array	Height	Width
Clear	SetIcon	operator =

Inheritance	<code>ZafIconData : (ZafImageData : ZafData : (ZafNotification, ZafElement)), ZafIconStruct</code>
Declaration	<code>#include <z_icon.hpp></code>
Description	ZafIconData objects can be used to store and manipulate icon information. ZafIconData is generally used in conjunction with the ZafIcon class.
Constructors	All ZafIconData constructors initialize the member variables associated with an instantiated ZafIconData object. The default values set by the ZafIconData and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafIconData	
<code>Array()</code>	<code>null</code>
ZafElement	
<code>ClassID()</code>	<code>ID_ZAF_ICON_DATA</code>
<code>ClassName()</code>	<code>"ZafIconData"</code>

```
ZafIconData(const ZafImageStruct &data);  
ZafIconData(const ZafIconStruct &data);
```

These constructors allocate a ZafIconData instance and initialize its data to the values in *data*.

```
ZafIconData(const ZafIChar *resourceName, int resourceID  
          = -1);
```

This constructor is useful in straight-code situations when loading a native icon (if supported by the environment). *resourceName* specifies the resource name of the icon in the application file or an open system file. *resourceID* specifies the resource ID number of the image in the application file or an open system file. Usually, either *resourceName* or *resourceID* will be used, but not both. If *resourceName* is null, it is ignored. If *resourceID* is -1, it is ignored.

```
ZafIconData(const ZafIconData &copy);
```

The copy constructor creates a new ZafIconData object and initializes its data from *copy*.

```
ZafIconData(const ZafIChar *name, ZafDataPersistence
             &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 code snippet shows how to create a ZafIconData object:

```
// Create an icon array.
#define blk ZAF_CLR_BLACK
#define wte ZAF_CLR_WHITE
static ZafLogicalColor ZAF_FARDATA plusIconArray[81] =
{
    blk,blk,blk,blk,blk,blk,blk,blk,blk,blk,
    blk,wte,wte,wte,wte,wte,wte,wte,blk,
    blk,wte,wte,wte,blk,wte,wte,wte,blk,
    blk,wte,wte,wte,blk,wte,wte,wte,blk,
    blk,wte,blk,blk,blk,blk,blk,wte,blk,
    blk,wte,wte,wte,blk,wte,wte,wte,blk,
    blk,wte,wte,wte,blk,wte,wte,wte,blk,
    blk,wte,wte,wte,wte,wte,wte,wte,blk,
    blk,blk,blk,blk,blk,blk,blk,blk,blk
};
static ZafIconStruct plusIcon(9, 9, plusIconArray, true);

// Create ZafIconData objects based on the icon array.
ZafIconData icon1(plusIcon);
ZafIconData icon2(icon1);
```

Destructor

```
virtual ~ZafIconData(void);
```

This virtual destructor is used to free the memory associated with an instantiated ZafIconData object.

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.

Array `ZafLogicalColor *Array(void) const;`

`Array()` returns the portable logical color array that the icon data is based on. This array is converted to a native environment icon handle. Each element of the array is a ZAF logical color. See [ZafDisplay](#) for more information.

Clear `virtual void Clear(void);`

`Clear()` destroys the portable `Array()` if `StaticArray()` is false, and it destroys the environment handle if `StaticHandle()` is false. Regardless of `StaticArray()` and `StaticHandle()`, the portable `Array()` and the environment handle are both set to null, effectively clearing the icon data.

Height `int Height(void) const;`

`Height()` returns the height of the icon data.

SetIcon `virtual ZafError SetIcon(int width, int height,
 ZafLogicalColor *array);`
`virtual ZafError SetIcon(const ZafIconStruct &icon);`

These functions copy the data passed in to the `ZafIconData` object. *width* and *height* are the width and height of *array*, respectively. If `StaticArray()` is true, *array* becomes `Array()`; otherwise, a new array is created for `Array()`. In the second function, the data is copied from *icon*. These functions always return `ZAF_ERROR_NONE`.

Width `int Width(void) const;`

`Width()` returns the width of the icon data.

operator = `ZafIconData &operator=(const ZafIconData &icon);`

This operator copies the data from *icon* into this `ZafIconData` object.

ZafIconStruct

[StaticHandle](#)

Inheritance ZafIconStruct : [ZafImageStruct](#)

Declaration `#include <z_dsp.hpp>`

Description ZafIconStruct is used to store icon information. The base ZafImageStruct stores the portable image array using elements of ZafLogicalColors, and ZafIconStruct defines additional members that store environment-specific structures for the icon filled by the ZafDisplay conversion function ZafDisplay::ConvertToOSIcon().

Constructors All ZafIconStruct constructors initialize the member variables associated with an instantiated ZafIconStruct object. The default values set by the ZafIconStruct constructors follow.

Member Initializations

ZafIconStruct

StaticHandle()	false
----------------	-------

ZafImageStruct

array	null
height	0
StaticArray()	false
width	0

ZafIconStruct(void);

This constructor allocates a ZafIconStruct instance and initializes its data to indicate that no icon information has been set.

ZafIconStruct(const ZafImageStruct &data);

This constructor allocates a ZafIconStruct instance and initializes its data to the values in *data*. The environment-specific icon information is initialized to indicate that the icon has not yet been converted.

```
ZafIconStruct(int width, int height, ZafLogicalColor
               *array, bool staticArray);
```

This constructor allocates a `ZafIconStruct` instance and initializes its data to the values passed in. *width* and *height* indicate the size of the image, *array* specifies a pointer to the portable array of `ZafLogicalColors`, and *staticArray* indicates if array is declared static.

The following code snippet shows how to create a `ZafIconStruct` object:

```
// Create an icon structure.
#define blk ZAF_CLR_BLACK
#define wte ZAF_CLR_WHITE
static ZafLogicalColor ZAF_FARDATA plusIconArray[81] =
{
    blk,blk,blk,blk,blk,blk,blk,blk,blk,
    blk,wte,wte,wte,wte,wte,wte,wte,blk,
    blk,wte,wte,wte,blk,wte,wte,wte,blk,
    blk,wte,wte,wte,blk,wte,wte,wte,blk,
    blk,wte,blk,blk,blk,blk,blk,wte,blk,
    blk,wte,wte,wte,blk,wte,wte,wte,blk,
    blk,wte,wte,wte,blk,wte,wte,wte,blk,
    blk,wte,wte,wte,wte,wte,wte,wte,blk,
    blk,blk,blk,blk,blk,blk,blk,blk,blk
};
static ZafIconStruct plusIcon(9, 9, plusIconArray, true);
```

Members

StaticHandle

```
bool StaticHandle(void) const;
bool SetStaticHandle(bool staticHandle);
```

If `StaticHandle()` is true, the environment-specific information (generally known as a handle) of the `ZafIconStruct` is recognized as static, so that it will not be deleted by ZAF, and may be used by several `ZafIconStruct` objects. This attribute is initialized to false, but it may be changed with `SetStaticHandle()`. Both functions return the current value of the `StaticHandle()` attribute.

ZafImage

AutoSize	PathName	Wallpaper
defaultFilterName	Scaled	
PathID	Tiled	

Inheritance ZafImage : ZafWindowObject : ZafElement

Declaration #include <z_image.hpp>

Description The ZafImage 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.

In Motif, a third-party image library is utilized that must be built to support xbm and xpm bitmaps in ZafImage. Information about this process is found in the file readme.mtf in the readme subdirectory.

Constructors All ZafImage constructors initialize the member variables associated with an instantiated ZafImage object. The default values set by the ZafImage 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 ZafImage. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafImage

AutoSize()	false
defaultFilterName[]	(platform-dependent)
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 [†]

Member Initializations

LinkDraggable()	false [†]
MoveDraggable()	false [†]
Noncurrent()	true [†]
OSDraw()	false
TextColor()	ZAF_CLR_NULL [†]

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.

AutoSize

```
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.

defaultFilterName

```
static ZafIChar ZAF_FARDATA defaultFilterName[];
```

defaultFilterName[] is a portable string that specifies a filter that accepts the default image type. For example, "*.bmp" is used in Microsoft Windows. See [ZafFileDialog::Filter\(\)](#) for more information.

PathID

```
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.

PathName

```
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.

Scaled

```
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.

Tiled

```
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.

Wallpaper

```
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.

ZafImageData

Inheritance ZafImageData : ZafData : ZafNotification, ZafElement

Declaration #include <z_idatal.hpp>

Description ZafImageData serves as the base class to all image types including: ZafBitmapData, ZafIconData, and ZafMouseDownData. The common aspect of these derived classes is their derivation and use of the structure ZafImageStruct, defined in z_dsp.hpp. ZafImageData provides protected members that allow derived objects to clear and manipulate the image information associated with their class.

ZafImageData is actually an abstract class! It's abstract nature is not readily apparent, but rather is implied by inheritance, since it does not provide an overload for the pure virtual base ZafData::Clear() function. Thus, derived image classes must resolve the Clear() abstraction inherited from the base ZafData class.

Constructor All ZafImageData constructors initialize the member variables associated with an instantiated ZafImageData object. Default values initialized by ZafImageData or overridden from base class constructors follow:

Member Initializations

ZafElement

ClassID()	ID_ZAF_IMAGE_DATA
ClassName()	"ZafImageData"

ZafImageData(void);

The ZafImageData class constructor, like the ZafData constructor, is protected. The primary purpose of this constructor is to chain the ZafImageData 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.

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

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

Destructor

```
virtual ~ZafImageData(void);
```

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

You can destroy a ZafImageData pointer even though the class definition is abstract. This is done by allocating a derived image object and then by setting the returned object to an image data pointer. The following code shows the correct use of the ZafImageData destructor under these conditions:

```
// Get a persistent bitmap button.  
ZafImageData *image = new ZafBitmapData("help_button",  
    zafDefaultStorage);  
...  
// Free the image.  
delete image;
```

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

ZafImageStruct

<i>array</i>	<i>width</i>
<i>StaticArray</i>	<i>height</i>

Inheritance	Root struct
Declaration	#include <z_dsp.hpp>
Description	ZafImageStruct is used to store portable images as an array of ZafLogicalColors. ZafImageStruct is used a base struct for other image structs in ZAF that have equivalent environment structures, such as ZafIconStruct and ZafMouseStruct.
Members	<div>ZafLogicalColor *array; <i>array</i> is a pointer to the array of ZafLogicalColors associated with the ZafImageStruct object. Since there is no constructor for this struct, this attribute is not initialized at this level, so the programmer must initialize it if explicitly creating a ZafImageStruct object.</div> <div><i>StaticArray</i> bool StaticArray(void) const; bool SetStaticArray(bool staticArray); If StaticArray() is true, array is recognized as static, so that it will not be deleted by ZAF, and may be used by several ZafImageStruct objects. Since there is no constructor for this struct, this attribute is not initialized at this level, so the programmer must initialize it if explicitly creating a ZafImageStruct object. The attribute may be initialized or changed with SetStaticArray(). Both functions return the current value of the StaticArray() attribute.</div> <div><i>width</i> <i>height</i> int width, height; width and height specify the size of array.</div>

ZafInteger

Event	SetInteger
IntegerData	Value

Inheritance	ZafInteger : ZafString : ZafWindowObject : ZafElement														
Declaration	#include <z_intl.hpp>														
Description	<p>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. See ZafString::AllowInvalid() and ZafString::ReportInvalid() for information on these attributes and how they affect validation for this class.</p> <p>All ZafInteger objects refer to data contained in a ZafIntegerData object (refer to this class for additional essential information).</p>														
Formats	<p>ZafString and derived classes parse input and display output based on default or programmer-defined input and output formats. Programmers may use the SetInputFormat() and SetOutputFormat() families of functions to change the default format. These functions are documented in the ZafString reference.</p> <p>Each class derived from ZafFormatData handles a different set of formatting arguments, in addition to those supported by its base classes. ZafIntegerData (and therefore ZafInteger) handles the following arguments:</p> <table> <tr> <th>Format Argument</th><th>Substitution</th></tr> <tr> <td>%d, %D</td><td>Decimal integer</td></tr> <tr> <td>%o, %O</td><td>Octal integer</td></tr> <tr> <td>%x</td><td>Hexadecimal integer using lower-case</td></tr> <tr> <td>%X</td><td>Hexadecimal integer using upper-case</td></tr> <tr> <td>%i</td><td>Decimal, hexadecimal, or octal integer</td></tr> <tr> <td>%u</td><td>Unsigned decimal integer</td></tr> </table>	Format Argument	Substitution	%d, %D	Decimal integer	%o, %O	Octal integer	%x	Hexadecimal integer using lower-case	%X	Hexadecimal integer using upper-case	%i	Decimal, hexadecimal, or octal integer	%u	Unsigned decimal integer
Format Argument	Substitution														
%d, %D	Decimal integer														
%o, %O	Octal integer														
%x	Hexadecimal integer using lower-case														
%X	Hexadecimal integer using upper-case														
%i	Decimal, hexadecimal, or octal integer														
%u	Unsigned decimal integer														
Constructors	<p>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</p>														

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 the `ZafInteger` object is to create, maintain and destroy its own `ZafIntegerData` object automatically. *left* and *top* specify the position where the left and top of the object will be placed on its parent, while *width* 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. *value* is the value that initially appears 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 manually, 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 some `ZafIntegerData` objects and create `ZafInteger` objects using the `ZafIntegerData` objects by passing them into *integerData*. For more information on using `ZafIntegerData` objects, see the chapter on [ZafIntegerData](#). *left*, *top*, and *width* are the same as the previous constructor.

ZafInteger(const `ZafInteger` ©);

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.

Event 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
N_RESET_I18N	causes the object to redisplay its data according to the new internationalization values
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

IntegerData 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](#). SetIntegerData() will delete the previous IntegerData() object if it is Destroyable() and no other object uses it.

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);
```

SetInteger

```
virtual ZafError SetInteger(long value);
```

SetInteger() sets the value of the ZafIntegerData associated with this ZafInteger from *value*.

Value

```
long Value(void);
```

Value() returns the value of the ZafIntegerData associated with this ZafInteger as a long.

ZafIntegerData

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

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

Declaration `#include <z_int.hpp>`

Description	<p>ZafIntegerData objects can be used to store and manipulate 32-bit integers.</p> <p>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.</p> <p>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.</p>
--------------------	---

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 initializes 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 true, a ZafIntegerData object will be destroyed automatically when all ZafInteger objects that refer to it are destroyed.

Members

Clear

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

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

FormattedText

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

FormattedText() fills *buffer* with a string representation of the Zaf-IntegerData 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* characters. 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, 256);
printf("decimal int - %s\n", buffer);
myint.FormattedText(buffer, 256, "%X");
printf("hexadecimal - %s\n", buffer);

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

SetInteger

```
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.

Value
long

```
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
```

operator --

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

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

operator ++

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

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

operator =

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

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

operator +=

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

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

operator -=

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

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

*operator *=*

```
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.

operator /=

```
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.

operator %=

```
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.

ZafKeyboard

AltPressed	Event	ShiftPressed
CtrlPressed	Poll	

Inheritance	ZafKeyboard : ZafDevice : ZafElement
Declaration	#include <z_keybrd.hpp>
Description	ZafKeyboard is the class that defines keyboard device support. A keyboard device accepts events from a keyboard, so keys typed by the end user enter the ZAF system through ZafKeyboard.
Constructors	All ZafKeyboard constructors initialize the member variables associated with an instantiated ZafKeyboard object. Default values set by the ZafKeyboard follow, as well as base class values when overridden by ZafKeyboard.

Member Initializations

ZafDevice	
DeviceType()	E_KEYBOARD
ZafElement	
ClassID()	ID_ZAF_KEYBOARD
ClassName()	"ZafKeyboard"
NumberID()	ID_ZAF_KEYBOARD
StringID()	"ZafKeyboard"

ZafKeyboard(ZafDeviceState state = D_ON);

This constructor is used to instantiate a ZafKeyboard object to be added to a ZafEventManager object. *state* specifies the initial state of the device.

ZafKeyboard(const ZafKeyboard ©);

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

```
// Instantiate the input devices.
ZafEventManager *eventManager = new ZafEventManager;
```

```
eventManager->Add(new ZafKeyboard);  
eventManager->Add(new ZafMouse);  
eventManager->Add(new ZafCursor);
```

Destructor

```
virtual ~ZafKeyboard(void);
```

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

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

Members

AltPressed

```
bool AltPressed(unsigned int shiftState) const;
```

AltPressed() returns true if *shiftState* indicates that the <Alt> (or <Command>) key was pressed; otherwise it returns false. When a keyboard event comes through the system, event.key.shiftState may be directly passed into AltPressed().

CtrlPressed

```
bool CtrlPressed(unsigned int shiftState) const;
```

CtrlPressed() returns true if *shiftState* indicates that the <Control> key was pressed; otherwise it returns false. When a keyboard event comes through the system, event.key.shiftState may be directly passed into CtrlPressed().

Event

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

This overloaded function handles all events sent to the ZafKeyboard object. ZafKeyboard handles the D_STATE message, which causes a device to return its state.

Poll

```
virtual void Poll(void);
```

In some environments, the Poll() function checks the keyboard device for any input events and posts them on the event manager's queue, if the ZafKeyboard's state is not D_OFF. In other environments where keyboard events are handled automatically by the native environment's event queue, Poll() simply blocks keyboard events from coming through ZAF's event manager queue if the ZafKeyboard's state is D_OFF.

ShiftPressed

```
bool ShiftPressed(unsigned int shiftState) const;
```

ShiftPressed() returns true if *shiftState* indicates that the <Shift> key was pressed; otherwise it returns false. When a keyboard event comes through the system, `event.key.shiftState` may be directly passed into **ShiftPressed()**.

ZafKeyStruct

shiftState	value
------------	-------

Inheritance

Root struct

Declaration

#include <z_key.hpp>

Description

ZafKeyStruct is used to store information about a keystroke. The most common use of ZafKeyStruct is by ZafEventStruct's key member. When a keyboard event comes through the system, event.key contains the information about the keystroke.

Members

ZafRawCode **shiftState**;

shiftState specifies the state of the modifier keys when the keystroke occurred. This member may contain any combination of the following bitwise values:

S_ALT	Indicates that the <alt> key was depressed
S_CAPS_LOCK	Indicates that the <caps lock> key was depressed
S_CMD	Indicates that the <command> key was depressed
S_CTRL	Indicates that the <control> key was depressed
S_INSERT	Indicates that the <insert> key was depressed
S_KEYDOWN	Indicates that the key was pressed
S_KEYUP	Indicates that the key was released
S_LEFT_SHIFT	Indicates that the left <shift> key was depressed
S_NUM_LOCK	Indicates that the <num lock> key was depressed
S_OPT	Indicates that the <option> key was depressed
S_RIGHT_SHIFT	Indicates that the right <shift> key was depressed
S_SCROLL_LOCK	Indicates that the <scroll lock> key was depressed
S_SHIFT	Indicates that either <shift> key was depressed

value

ZafIChar **value**;

value specifies the character value of the key that was pressed, if it is mappable. In ISO 8859-1 mode the value is an ISO 8859-1 code, in Unicode mode the value is a Unicode code, and in native mode the value is an ASCII code.

ZafLanguageData

Clear	Lock	SetMessages
GetMessage	Locked	StaticData
GetMessageData	Messages	UnLock
LanguageName	SetMessage	ZafMessageStruct

Inheritance `ZafLanguageData : ZafDataRecord : ZafData :
(ZafNotification, ZafElement), ZafList`

Declaration `#include <z_lang.hpp>`

Description ZafLanguageData maintains message tables so that different languages may easily be supported by a single executable and a data file containing the application messages (see [ZafMessageStruct](#) for more information on the format of the message table entries). Message tables that utilize this functionality for stock library objects (error messages, etc.) are included in the file `i18n.znc`, which must be available for the ZAF application that requires these tables. (Place `i18n.znc` in the directory with the application or the directory specified by `ZAF_BIN`. The `i18n.znc` file is shipped in the `zaf/bin` directory.) Developers may also create their own message tables for customized applications.

Individual ZafLanguageData instances are normally managed by the [ZafLanguageManager](#) for easy access within an application. See [ZafI18nData::ResetI18n\(\)](#) for more information about internationalization of an application.

Many ZAF classes automatically load ZafLanguageData objects of their own via requests to [ZafLanguageManager::Allocate\(\)](#), but the programmer may create ZafLanguageData objects to keep track of tables of strings as needed.

Constructors All ZafLanguageData constructors initialize the member variables associated with an instantiated ZafLanguageData object. The default values set by the ZafLanguageData and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafLanguageData

`StaticData()` `false`

ZafElement

`ClassID()` `ID_ZAF_LANGUAGE_DATA`

`ClassName()` `" ZafLanguageData "`

```
ZafLanguageData(bool staticData = false);
```

This constructor creates a new empty language table that may be loaded using `SetLanguage()`. *staticData* indicates the initial value of `StaticData()`.

```
ZafLanguageData(ZafLanguageStruct *data, bool staticData  
= false);
```

This constructor creates a new language table based on the table specified by *data*. *staticData* specifies the value of `StaticData()`. All the information in *data* is copied to the new language table.

```
ZafLanguageData(const ZafLanguageData &copy);
```

The copy constructor creates a new `ZafLanguageData` object and initializes its data from *copy*.

```
ZafLanguageData(const ZafIChar *name, ZafDataPersistence  
    &persist, const ZafIChar *languageName =  
    ZAF_NULLP(ZafIChar));
```

The final constructor is used for persistence. Different objects may be stored according to language. *languageName* specifies which language to use when loading the object. If *languageName* is null, the current language is used (see [LanguageName\(\)](#) for more information). Refer to `ZafWindow` for more information, since most persistence is done at the `ZafWindow` level.

Destructor

```
virtual ~ZafLanguageData(void);
```

The destructor is used to free the memory associated with a `ZafLanguageData` object. It chains to the `ZafI18nData`, `ZafData`, `ZafNotification`, `ZafElement` destructors.

Members

Clear

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

`Clear()` clears the language table associated with this `ZafLanguageData` object and deletes its language table if `StaticData()` is false.

GetMessage

```
ZafIChar *GetMessage(ZafNumberID numberID, bool  
    useDefault = false, ZafIChar *hotKeyChar =  
    ZAF_NULLP(ZafIChar), int *hotKeyIndex =  
    ZAF_NULLP(int)) const;
```

```
ZafIChar *GetMessage(const ZafIChar *stringID, bool
    useDefault = false, ZafIChar *hotKeyChar =
    ZAF_NULLP(ZafIChar), int *hotKeyIndex =
    ZAF_NULLP(int)) const;
```

GetMessage() finds the message in the language table according to *numberID* or *stringID*. If the message was not found and *useDefault* is true, the first message in the table will be returned; if the message was not found and *useDefault* is false, null is returned. The hot key character associated with the message is returned in *hotKeyChar* if it is not null, and the hot key index is returned in *hotKeyIndex* if it is not null. A pointer to the message is returned if it was found.

GetMessageData

```
ZafMessageData *GetMessageData(const ZafIChar *stringID,
    bool useDefault = false);
ZafMessageData *GetMessageData(ZafNumberID numberID, bool
    useDefault = false);
```

GetMessageData() finds the message in the language table according to *numberID* or *stringID*. If the message was not found and *useDefault* is true, the first message in the table will be returned; if the message was not found and *useDefault* is false, null is returned. A pointer to a ZafMessageData object corresponding to the message is returned if it was found.

LanguageName

```
static const ZafIChar *LanguageName(void);
static ZafError SetLanguageName(const ZafIChar
    *languageName);
```

LanguageName() is the two-character ISO code for a language, such as "en" for English. SetLanguageName() uses the two-character ISO code *languageName* to reset the language information for an application, and is called from ZafI18nData::SetI18nName(). The programmer should normally not call SetLanguageName(). Instead, ZafI18nData::ResetI18n() should be used to change the active language. Refer to the file Readme.i18 for the most current list of languages defined in ZAF.

Lock

Locked

UnLock

```
void Lock(void);
bool Locked(void);
void UnLock(void);
```

If Locked() is true, the message table associated with the ZafLanguageData is being used, and it will not be deleted by the ZafLanguageManager. Lock() may be called to lock the message table, and UnLock() may be called to unlock the message table. These are advanced methods used internally by ZAF and should normally not be used by the programmer.

Messages

```
int Messages(void);
```

Messages() returns the number of entries in the message table associated with the ZafLanguageData.

SetMessage

```
virtual ZafError SetMessage(ZafNumberID numberID,  
    ZafIChar *text, ZafIChar hotKeyChar = 0, int  
    hotKeyIndex = -1);
```

SetMessage() finds the message in the table with *numberID*, and sets its information according to the information passed into the other parameters. *text* specifies the message text, *hotKeyChar* specifies the hot key character, and *hotKeyIndex* specifies the hot key index. If the message was set successfully, ZAF_ERROR_NONE is returned; otherwise, ZAF_ERROR_INVALID_ID is returned if the message was not found or ZAF_ERROR_INVALID_SOURCE is returned if the table was empty.

SetMessages

```
virtual ZafError SetMessages(ZafMessageStruct *value);  
virtual ZafError SetMessages(const ZafLanguageData  
    &data);
```

SetMessages() resets the language table associated with the ZafLanguageData object by copying the information from *value* or *data* and causing the language manager to update all its data objects that rely on the language table entries.

StaticData

```
bool StaticData(void) const;  
virtual bool SetStaticData(bool staticData);
```

If StaticData() is true, the message table associated with the ZafLanguageData object is recognized as static, so that it will not be deleted by ZAF, and may be used by several ZafLanguageData objects. The attribute may be changed with SetStaticArray(). Both functions return the current value of the StaticData() attribute.

ZafMessageStruct

```
operator ZafMessageStruct *();
```

This operator returns a pointer to the language table associated with this ZafLanguageData object.

This constructor creates a new empty language manager. Normally, this constructor will not be called by the programmer, since `ZafApplication` creates the global `zafLanguageManager` member.

Destructor

```
virtual ~ZafLanguageManager(void);
```

The destructor is used to free the memory associated with a `ZafLanguageManager` object.

Allocate

```
static ZafLanguageData *Allocate(const ZafIChar  
    *stringID, ZafMessageStruct *defaultData =  
    ZAF_NULLP(ZafMessageStruct));
```

`Allocate()` returns a pointer to the `ZafLanguageData` object identified by a `stringID` of *stringID*. If it hasn't been loaded yet, `Allocate()` asks `ZafDataManager` to load the object using `ZafDataManager::AllocateData()`. If the object cannot be found, the message table *defaultData* is used to create a new `ZafLanguageData` object to be returned. The `ZafLanguageData` returned is added to the language manager object.

blankString

```
static ZafIChar ZAF_FARDATA blankString[];
```

`blankString` specifies a default string with simply a null terminator in it for use in clearing out any string in an application, since it is public and static.

errorString

```
static ZafIChar ZAF_FARDATA errorString[];
```

`errorString` specifies a default string with the word "Error" in it for the current language being used in the application for use anywhere in an application, since it is public and static.

Free

```
static void Free(ZafLanguageData *messageGroup);
```

`Free()` subtracts *messageGroup* from the language manager object and deletes it when it is no longer in use.

Language

```
const ZafIChar *Language(void);  
ZafError SetLanguage(const ZafIChar *languageName);
```

`Language()` is the two-character ANSI code for a language, such as "en" for English. `SetLanguage()` uses the two-character ANSI code *languageName* to reset the language information for an application, and is called from `ZafI18nData::SetI18nName()`. The programmer should normally not call `Set-`

Language(). Instead, ZafI18nData::ResetI18n() should be used to change the active language. Refer to the file readme.i18 for the most current list of languages defined in ZAF.

LanguageAllocate static void **LanguageAllocate**(const ZafIChar *name = ZAF_NULLP(ZafIChar));

LanguageAllocate() is called by ZafI18nData to allocate the language manager object for an application initially using the language specified by *name*. *name* must be non-null, and is the two-character ANSI code for a language, such as "en" for English. The programmer should normally not call LanguageAllocate(). Instead, ZafI18nData::ResetI18n() should be used to change the active language. Refer to the file readme.i18 for the most current list of languages defined in ZAF. Other languages may be created by the programmer using Zinc Designer.

LanguageFree static void **LanguageFree**(bool globalRequest = false);

LanguageFree() is called by ZafI18nData to delete the language manager object for an application with *globalRequest* being true. The programmer should normally not call LanguageFree().

LanguageName static const ZafIChar ***LanguageName**(void);
static ZafError **SetLanguageName**(const ZafIChar *languageName);

LanguageName() is a static member that may be used instead of calling zafLanguageManager->Language(). SetLanguageName() is a static member that may be used instead of calling zafLanguageManager->SetLanguage(). See [Language\(\)](#) for more information.

ZafList

Add	Find	Sort
CompareFunction	First	Subtract
Count	Get	operator ()
Current	Index	operator +
Destroy	Last	operator -

Inheritance	Root class
Declaration	<code>#include <z_list.hpp></code>
Description	<p>ZafList provides support for linked lists of ZafElement objects. Through multiple inheritance, ZafWindow and other classes derive from ZafList to maintain lists of children.</p> <p>ZafList is also useful as a container class. Any object derived from ZafElement may be kept in a ZafList.</p>
Constructor	ZafList initializes its members to the following default values:

Member Initializations

ZafList

CompareFunction()	user-supplied parameter
Count()	0
Current()	null
First()	null
Last()	null

```
ZafList(ZafCompareFunction compareFunction =
        ZAF_NULLF( ZafCompareFunction ));
```

The ZafList class constructor creates an empty ZafList object. *compareFunction* specifies the function used when elements are compared against each other to determine their order in the list. Below is the definition of ZafCompareFunction. The compare function returns zero if the two parameters are considered equal. If the first should be sorted before the second, then some integer less than zero is returned; otherwise some integer greater than zero is returned.

```
typedef int (*ZafCompareFunction)(ZafElement *, ZafElement *);
```


The following code snippet shows how to create a ZafList object and use it to store a list of objects:

```
// Create an empty list object.
ZafList *list = new ZafList;

// Create several string data objects and store them
// in the list.
for (int i = 0; i < 10; i++)
{
    ZafIChar tempString[16];
    sprintf(tempString, "String %d", i);
    list->Add(new ZafStringData(tempString));
}
```

Destructor

```
virtual ~ZafList(void);
```

This virtual destructor is used to free the memory associated with an instantiated ZafList object. All the elements attached to the list are deleted.

Members

Add

```
ZafElement *Add(ZafElement *element);
ZafElement *Add(ZafElement *element, ZafElement
    *position);
```

operator +

```
ZafList &operator+(ZafElement *element);
```

These functions and operator add *element* to the ZafList object. The Add() functions return a pointer to the object that was added, and the operator returns the ZafList object (useful in multiple uses of the operator in a single statement). *position*, if specified, refers to the element already in the list that *element* is inserted before.

If *position* is not specified, as with the first Add() function and with the operator, the function returned by CompareFunction() is called to determine the position of the new element in the list.

CompareFunction

```
virtual ZafCompareFunction CompareFunction(void) const;
virtual ZafCompareFunction
    SetCompareFunction(ZafCompareFunction
        compareFunction);
```

The function specified by CompareFunction() is used to determine element ordering in the ZafList. If CompareFunction() is null and no position is specified in the Add() function, each new element is added to the end of the list. SetCompareFunction() may be called to set the function used to compare ele-

ments. A pointer to the compare function is returned by both of these functions. See the typedef of `ZafCompareFunction` under the [Constructor](#) section.

If `SetCompareFunction()` is called after the list has been created and populated with elements, `Sort()` must be called to reorder the list using the new `CompareFunction()`.

Count `int Count(void) const;`

`Count()` returns the number of elements in the `ZafList`.

Current `ZafElement *Current(void) const;`
`void SetCurrent(ZafElement *element);`

`Current()` specifies the element in the list considered to be current. `Current()` is not modified by the `ZafList` class, but must be maintained by the class utilizing the `ZafList`. `Current()` will remain null, as it is initialized, unless `SetCurrent()` is called with *element* specifying the element in the list to be considered current.

If the `Current()` element in the list is subtracted from the list, `Current()` will be set to null. The ZAF classes that inherit from `ZafList` call `SetCurrent()` internally (usually within `NotifyFocus()`), and the programmer should normally not call `SetCurrent()`.

Destroy `virtual void Destroy(void);`

`Destroy()` subtracts all the elements from the list and deletes them, in effect making the list empty.

Find `ZafElement *Find(ZafNumberID numberID);`
`ZafElement *Find(const ZafIChar *stringID);`

These functions return a pointer to the element in the list with either the identifier *numberID* or the identifier *stringID*. If an element cannot be found with the identifier provided, null is returned. See [ZafElement](#) for more information on these identifiers.

First `ZafElement *First(void) const;`

`First()` returns a pointer to the first element in the list. If the list is empty, null is returned.

```

Get                                ZafElement *Get(int (*findFunction)(void *element1, void
                                     *matchData), void *matchData);
                                     ZafElement *Get(int index);
operator ()                         ZafElement *operator()(int index);

```

The first Get() function returns a pointer to the element in the list if *findFunction* returns zero based on *matchData*. In other words, Get() calls *findFunction* on each element in the list until one is found that is considered to match *matchData*. If the element is found, a pointer to it is returned; otherwise null is returned.

The second function and the operator return a pointer to the element in the list with the zero- based *index*. If the element does not exist, null is returned.

```

Index                             int Index(ZafElement const *element) const;

```

Index() returns the zero-based index of the *element* in the list. If the element is not in the list, -1 is returned.

```

Last                             ZafElement *Last(void) const;

```

Last() returns a pointer to the last *element* in the list. If the list is empty, null is returned.

```

Sort                             virtual void Sort(void);

```

Sort() causes the elements in the list to be reordered according to CompareFunction(). Add() places each element in the list appropriately, but Sort() must be called when the elements are out of place as a result of either a call to SetCompareFunction() or a call to Add() with the position specified.

```

Subtract                         ZafElement *Subtract(ZafElement *element);
operator -                       ZafList &operator-(ZafElement *element);

```

This function and operator subtract *element* from the ZafList object. Subtract() returns a pointer to the next object in the list (or null if *element* is the last element), and the operator returns the ZafList object (useful in multiple uses of the operator in a single statement). If *element* is Current(), then Current() is set to null.

ZafListBlock

[Full](#)

Inheritance	<code>ZafListBlock</code> : ZafList
Declaration	<pre>#include <z_list.hpp></pre>
Description	<p><code>ZafListBlock</code> is a container object used as a base class by <code>ZafQueueBlock</code> to maintain a list block of <code>ZafElement</code> objects. Only <code>ZafElement</code> objects may be added to a <code>ZafListBlock</code>.</p>
Constructors	<p>All <code>ZafListBlock</code> constructors initialize the member variables associated with an instantiated <code>ZafListBlock</code> object.</p> <pre>ZafListBlock(int noOfElements, ZafCompareFunction compareFunction = ZAF_NULLF(ZafCompareFunction));</pre> <p>This constructor creates a <code>ZafListBlock</code> of <i>noOfElements</i> <code>ZafElements</code>. <i>compareFunction</i> specifies the function used to compare elements when sorting them in the list, and is passed into the <code>ZafList</code> constructor. See ZafList for more information about compare functions.</p>
Destructor	<pre>virtual ~ZafListBlock(void);</pre> <p>The destructor is used to free the memory associated with a <code>ZafListBlock</code> object, including all the <code>ZafElement</code> objects associated with it. It chains to the <code>ZafList</code> destructor.</p>
Members	<pre>bool Full(void) const;</pre> <p><i>Full</i></p> <p><code>Full()</code> returns true if the list block is full, meaning no other elements may be added to it. Otherwise, it returns false.</p>

ZafLocaleData

canonicalLocale	LocaleFree	TimeStamp
Clear	LocaleName	
LocaleAllocate	SetLocale	

Inheritance	<code>ZafLocaleData : ZafI18nData : (ZafData : ZafNotification, ZafElement), ZafLocaleStruct)</code>
Declaration	<code>#include <z_loc.hpp></code>
Description	<p>ZafLocaleData maintains the locale information necessary to internationalize an application, such as month/day/year order, currency symbols, and decimal separators (see ZafLocaleStruct for more information). Locale information is loaded from the operating system at application start-up (ZafApplication calls LocaleAllocate()), and the programmer may change locale information at run-time. The locale tables that enable this functionality are included in the file <code>i18n.znc</code>, which must be available in the same directory as the ZAF application that requires these tables. The <code>i18n.znc</code> file is shipped in the <code>zaf/bin</code> directory. See <code>ZafI18nData::ResetI18n()</code> for more information about internationalization of an application.</p> <p>The programmer will generally not create a <code>ZafLocaleData</code> object, as the global <code>zafLocale</code> is instantiated by the static method <code>LocaleAllocate()</code> at start-up.</p>
Constructors	<p>All <code>ZafLocaleData</code> constructors initialize the member variables associated with an instantiated <code>ZafLocaleData</code> object. The default values set by the <code>ZafLocaleData</code> and its base class constructors follow, if they differ from those set by the base class constructor.</p>

Member Initializations

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_LOCALE_DATA</code>
<code>ClassName()</code>	<code>"ZafLocaleData"</code>

ZafLocaleData(void);

This constructor creates a new locale copied from the canonical locale (see [canonicalLocale](#) below).

```
ZafLocaleData(const ZafLocaleStruct &data);
```

This constructor creates a new locale based on the locale specified in the `ZafLocaleStruct` *data*. All the information in *data* is copied to the new locale.

```
ZafLocaleData(const ZafLocaleData &copy);
```

The copy constructor creates a new `ZafLocaleData` object and initializes its data from *copy*.

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

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

Destructor

```
virtual ~ZafLocaleData(void);
```

The destructor is used to free the memory associated with a `ZafLocaleData` object. It chains to the `ZafI18nData`, `ZafData`, `ZafNotification`, `ZafElement` destructors. The programmer should normally not call this destructor, since the locale is destroyed by `ZafApplication` at program termination. For more information on child object deletion, see [ZafWindow::~ZafWindow\(\)](#).

Members

canonicalLocale

```
static ZafLocaleData *canonicalLocale;
```

The currently active locale in a program may be changed at any time, which presents the need for a canonical, or common form for storing locale information. `canonicalLocale` stores the canonical locale information used for storing data that contains locale information. For example, when a data object containing locale-dependent information is stored, it is converted to the canonical, or common locale, so that when the information gets used again, it can be converted from a known form.

Clear

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

`Clear()` copies the locale data from `canonicalLocale`, in effect clearing the locale information.

SetLocale

```
ZafError SetLocale(const ZafLocaleStruct &value);
```

`SetLocale()` changes the currently active locale by copying the locale information from *value* and causing the data manager to update all its data objects that rely on local information.

LocaleAllocate

```
static void LocaleAllocate(const ZafIChar *name =
    ZAF_NULLP(ZafIChar));
```

LocaleAllocate() is called by ZafI18nData to allocate the locale data object for an application using the locale specified by *name*. *name* is the two-character ISO code for a locale, such as "US" for the United States. If *name* is null, the default locale compiled into the application is used. The programmer should normally not call LocaleAllocate(). Instead, ZafI18nData::ResetI18n() should be used to change the active locale. Refer to the file Readme.i18 for the most current list of locales defined in ZAF. Other locales may be created by the programmer using Zinc Designer.

LocaleFree

```
static void LocaleFree(bool globalRequest = false);
```

LocaleFree() is called by ZafI18nData to delete the locale data object for an application with *globalRequest* being true. The programmer should normally not call LocaleFree().

LocaleName

```
static const ZafIChar *LocaleName(void);
static ZafError SetLocaleName(const ZafIChar
    *localeName);
```

LocaleName() is the two-character ANSI code for a locale, such as "US" for the United States. SetLocaleName() uses the two-character ANSI code *locale-Name* to reset the locale information for an application, and is called from ZafI18nData::SetI18nName(). The programmer should normally not call SetLocaleName(). Instead, ZafI18nData::ResetI18n() should be used to change the active locale. Refer to the file Readme.i18 for the most current list of locales defined in ZAF.

TimeStamp

```
static ZafUInt32 TimeStamp(void);
```

TimeStamp() returns the current date and time in a platform-independent form. The number of seconds since January 1, 1970 is returned.

ZafLocaleStruct

altDigits	eraTableLength	posCurrencyPrecedes
beginGregorian	fractionDigits	positiveSign
creditLeftParen	grouping	posSignPrecedes
creditRightParen	intCurrencySymbol	postSpaceSeparation
currencySymbol	integerStringInputFormat	realStringInputFormat
dateSeparator	integerStringOutputFormat	realStringOutputFormat
dateStringInputFormat	intFractionDigits	skipGregorian
dateStringOutputFormat	monDecimalSeparator	thousandsSeparator
dateTimeStringInputFormat	monGrouping	time12StringOutputFormat
dateTimeStringOutputFormat	monThousandsSeparator	timeSeparator
decimalSeparator	negativeSign	timeStringInputFormat
defDigits	negCurrencyPrecedes	timeStringOutputFormat
eraTable	negSpaceSeparation	

Inheritance Root struct

Declaration `#include <z_loc.hpp>`

Description ZafLocaleStruct is used as a base class by [ZafLocaleData](#), and stores information that may be different for different locales. This information defines a unique locale, and is used by many parts of the ZAF libraries to provide internationalization to an application. See [ZafLocaleData](#) for more information.

Members `ZafIChar *altDigits;`
altDigits altDigits is the array of alternate digit characters to be used with "%ad" in format strings such as a ZafInteger object's OutputFormatData().

beginGregorian `ZafUInt32 beginGregorian;`

beginGregorian is the Julian day that the Gregorian calendar was adopted. For example, the United Kingdom (and thus the United States) adopted the Gregorian calendar on October 4, 1582 (2299160 Julian).

creditLeftParen `ZafIChar *creditLeftParen;`

creditLeftParen is the character used on the left of a negative number when the "@" format character is encountered in format strings such as a ZafBignum object's OutputFormatData().

<i>creditRightParen</i>	<pre>ZafIChar *creditRightParen;</pre> <p>creditRightParen is the character used on the right of a negative number when the "@" format character is encountered in format strings such as a ZafBignum object's <code>OutputFormatData()</code>.</p>
<i>currencySymbol</i>	<pre>ZafIChar currencySymbol[8];</pre> <p>currencySymbol is the symbol used locally to denote currency.</p>
<i>dateSeparator</i>	<pre>ZafIChar dateSeparator[4];</pre> <p>dateSeparator is used to separate the day, month, and year pieces of a date string.</p>
<i>dateStringInput- Format</i>	<pre>ZafIChar *dateStringInputFormat;</pre> <p>dateStringInputFormat is the format string used when inputting dates. See <code>ZafString::InputFormatData()</code> for more information.</p>
<i>dateStringOutput- Format</i>	<pre>ZafIChar *dateStringOutputFormat;</pre> <p>dateStringOutputFormat is the format string used when outputting dates. See <code>ZafString::OutputFormatData()</code> for more information.</p>
<i>dateTimeStringIn- putFormat</i>	<pre>ZafIChar *dateTimeStringInputFormat;</pre> <p>dateTimeStringInputFormat is the format string used when inputting date/time combination strings, such as with <code>ZafUTime</code> objects. See <code>ZafString::InputFormatData()</code> for more information.</p>
<i>dateTimeString- OutputFormat</i>	<pre>ZafIChar *dateTimeStringOutputFormat;</pre> <p>dateTimeStringOutputFormat is the format string used when outputting date/time combination strings, such as with <code>ZafUTime</code> objects. See <code>ZafString::OutputFormatData()</code> for more information.</p>
<i>decimalSeparator</i>	<pre>ZafIChar decimalSeparator[4];</pre> <p>decimalSeparator is the symbol used to separate the whole number portion from the fraction in a non-monetary decimal number, as in <code>ZafReal</code> objects.</p>

<i>defDigits</i>	<pre>ZafIChar *defDigits;</pre> <p>defDigits is the array of default digit characters to be used with "%d" in format strings such as a ZafInteger object's <code>OutputFormatData()</code>.</p>
<i>eraTable</i>	<pre>ZafEraStruct *eraTable;</pre> <p>eraTable is a table of all the different eras for the locale, and is a table of ZafEraStruct objects. See ZafEraStruct for more information.</p>
<i>eraTableLength</i>	<pre>int eraTableLength;</pre> <p>eraTableLength is the number of elements in eraTable.</p>
<i>fractionDigits</i>	<pre>int fractionDigits;</pre> <p>fractionDigits is the number of digits to display after the decimal separator on currency values, as in ZafBignum objects.</p>
<i>grouping</i>	<pre>char grouping[10];</pre> <p>grouping indicates the numbers of digits to be grouped together from right to left in each section of the whole number portion of a non-monetary number. For example, the first element in the array specifies the number of digits in the first group to the left of the decimal separator. An element value of 0 indicates the end of the array, and the previous element's value is used for the remaining digits in the number.</p>
<i>intCurrencySymbol</i>	<pre>ZafIChar intCurrencySymbol[5];</pre> <p>intCurrencySymbol is the international currency symbol followed by a space separator.</p>
<i>integerStringInput- Format</i>	<pre>ZafIChar *integerStringInputFormat;</pre> <p>integerStringInputFormat is the format string used when inputting integers. See <code>ZafString::InputFormatData()</code> for more information.</p>
<i>integerStringOutput- Format</i>	<pre>ZafIChar *integerStringOutputFormat;</pre> <p>integerStringOutputFormat is the format string used when outputting integers. See <code>ZafString::OutputFormatData()</code> for more information.</p>

<i>intFractionDigits</i>	<pre>int intFractionDigits;</pre> <p>intFractionDigits is the number of digits to display after the decimal separator on international currency values, as in ZafBignum objects. This member is used together with intCurrencySymbol.</p>
<i>monDecimal-Separator</i>	<pre>ZafIChar monDecimalSeparator[4];</pre> <p>monDecimalSeparator is the symbol used to separate the whole number portion from the fraction in a monetary decimal number, as in ZafReal objects.</p>
<i>monGrouping</i>	<pre>char monGrouping[10];</pre> <p>monGrouping indicates the numbers of digits to be grouped together from right to left in each section of the whole number portion of a monetary number. For example, the first element in the array specifies the number of digits in the first group to the left of the decimal separator. An element value of 0 indicates the end of the array, and the previous element's value is used for the remaining digits in the number.</p>
<i>monThousands-Separator</i>	<pre>ZafIChar monThousandsSeparator[4];</pre> <p>monThousandsSeparator is the symbol used to separate thousands (every 3 digits on the left of the decimal separator) in a monetary decimal number, as in ZafInteger objects.</p>
<i>negativeSign</i>	<pre>ZafIChar negativeSign[4];</pre> <p>negativeSign is the symbol used to indicate a negative number.</p>
<i>negCurrency-Precedes</i>	<pre>int negCurrencyPrecedes;</pre> <p>negCurrencyPrecedes is non-zero if the currency symbol precedes negative currency values; otherwise the currency symbol follows negative currency values.</p>
<i>negSignPrecedes</i>	<pre>int negSignPrecedes;</pre> <p>negSignPrecedes is non-zero if the negative symbol precedes negative values; otherwise the negative symbol follows negative values.</p>

*negSpace-
Separation*

int **negSpaceSeparation**;

negSpaceSeparation is non-zero if a space separator is placed between the negative symbol and the value; otherwise no space separator is used.

*posCurrency-
Precedes*

int **posCurrencyPrecedes**;

posCurrencyPrecedes is non-zero if the currency symbol precedes positive currency values; otherwise the currency symbol follows positive currency values.

positiveSign

ZafIChar **positiveSign**[4];

positiveSign is the symbol used to explicitly indicate a positive number.

posSignPrecedes

int **posSignPrecedes**;

posSignPrecedes is non-zero if the explicit positive symbol precedes positive values; otherwise the explicit positive symbol follows positive values.

*postSpace-
Separation*

int **posSpaceSeparation**;

posSpaceSeparation is non-zero if a space separator is placed between the explicit positive symbol and the value; otherwise no space separator is used.

*realStringInput-
Format*

ZafIChar ***realStringInputFormat**;

realStringInputFormat is the format string used when inputting real numbers. See ZafString::[InputFormatData\(\)](#) for more information.

*realStringOutput-
Format*

ZafIChar ***realStringOutputFormat**;

realStringOutputFormat is the format string used when outputting real numbers. See ZafString::[OutputFormatData\(\)](#) for more information.

skipGregorian

ZafUInt16 **skipGregorian**;

skipGregorian is the number of days skipped when the Gregorian calendar was adopted. For example, the United Kingdom (and thus the United States) skipped 11 days to October 15, 1582.

<i>thousandsSeparator</i>	<code>ZafIChar thousandsSeparator[4];</code> thousandsSeparator is the symbol used to separate thousands (every 3 digits on the left of the decimal separator) in a non-monetary decimal number, as in ZafInteger objects.
<i>time12StringOutputFormat</i>	<code>ZafIChar *time12StringOutputFormat;</code> time12StringOutputFormat is the format string used when outputting explicitly 12-hour clock times. See ZafString::OutputFormatData() for more information.
<i>timeSeparator</i>	<code>ZafIChar timeSeparator[4];</code> timeSeparator is used to separate the hour, minute, and second pieces of a time string.
<i>timeStringInputFormat</i>	<code>ZafIChar *timeStringInputFormat;</code> timeStringInputFormat is the format string used when inputting times. See ZafString::InputFormatData() for more information.
<i>timeStringOutputFormat</i>	<code>ZafIChar *timeStringOutputFormat;</code> timeStringOutputFormat is the format string used when outputting times. See ZafString::OutputFormatData() for more information.

ZafMaximizeButton

Inheritance	<code>ZafMaximizeButton : ZafButton : ZafWindowObject : ZafElement</code>
Declaration	<code>#include <z_max.hpp></code>
Description	The ZafMaximizeButton object may only be added to a ZafWindow. The ZafMaximizeButton is the maximize/restore button decoration on a ZafWindow, and is generally drawn by the environment. The ZafMaximizeButton object is used to maximize and restore the parent window with a ZafMouse device.
Constructors	All ZafMaximizeButton constructors initialize the member variables associated with an instantiated ZafMaximizeButton object. The default values set by the ZafMaximizeButton 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 ZafMaximizeButton. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafButton

<code>AllowDefault()</code>	<code>false</code> [†]
<code>AllowToggling()</code>	<code>false</code> [†]
<code>AutoRepeatSelection()</code>	<code>false</code> [†]
<code>AutoSize()</code>	<code>true</code> [†]
<code>ButtonType()</code>	<code>ZAF_3D_BUTTON</code> [†]
<code>Depth()</code>	<code>1</code> [†]
<code>HotKeyChar()</code>	<code>0</code> [†]
<code>HotKeyIndex()</code>	<code>-1</code> [†]
<code>HzJustify()</code>	<code>ZAF_HZ_CENTER</code> [†]
<code>SelectOnDoubleClick()</code>	<code>false</code> [†]
<code>SelectOnDownClick()</code>	<code>false</code> [†]
<code>SendMessageText()</code>	<code>null</code> [†]
<code>SendMessageWhenSelected()</code>	<code>true</code> [†]
<code>Value()</code>	Used internally by ZAF [†]
<code>VtJustify()</code>	<code>ZAF_VT_CENTER</code> [†]

ZafWindowObject

<code>AcceptDrop()</code>	<code>false</code> [†]
<code>Bordered()</code>	<code>false</code> [†]
<code>CopyDraggable()</code>	<code>false</code> [†]

Member Initializations

Disabled()	false [†]
Focus()	false [†]
HelpContext()	null [†]
HelpObjectTip()	null [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
Noncurrent()	true [†]
ParentDrawBorder()	false [†]
ParentDrawFocus()	false [†]
ParentPalette()	false [†]
QuickTip()	null [†]
RegionType()	ZAF_AVAILABLE_REGION [†]
Selected()	false [†]
SupportObject()	true [†]
SystemObject()	false
UserFunction()	null [†]

ZafElement

ClassID()	ID_ZAF_MAXIMIZE_BUTTON
ClassName()	"ZafMaximizeButton"
NumberID()	ZAF_NUMID_MAXIMIZE
StringID()	"ZAF_NUMID_MAXIMIZE"

ZafMaximizeButton(void);

This constructor is useful in straight-code situations to create a ZafMaximizeButton object.

ZafMaximizeButton(const ZafMaximizeButton ©);

The copy constructor creates a new ZafMaximizeButton object and initializes its data from *copy*.

**ZafMaximizeButton(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 maximize button follows:

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

Destructor

```
virtual ~ZafMaximizeButton(void);
```

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

Member Initializations

ZafMDIWindow

MDIType()	ZAF_MDI_PARENT
-----------	----------------

ZafWindow

SelectionType()	ZAF_SINGLE_SELECTION ^{††}
Temporary()	false [†]

ZafWindowObject

AcceptDrop()	false ^{††}
Bordered()	false [†]
Disabled()	false [†]
Noncurrent()	false [†]
ParentPalette()	false [†]
RegionType()	ZAF_INSIDE_REGION [†]

ZafElement

ClassID()	ID_ZAF_MDI_WINDOW
ClassName()	"ZafMDIWindow"

```
ZafMDIWindow(int left, int top, int width, int height,  
               ZafMDIType type = ZAF_MDI_PARENT);
```

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 or on the window manager. *width* and *height* specify the width and height of the client region of the object. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. *type* specifies the type of MDI window and may be either ZAF_MDI_PARENT or ZAF_MDI_CHILD. See [MDIType](#) for important information about MDI window types.

```
ZafMDIWindow(const ZafMDIWindow &copy);
```

The copy constructor creates a new ZafMDIWindow object and initializes its data from *copy*.

```
ZafMDIWindow(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 an MDI window follows:

```
// Create a parent MDI window with a menu bar.
ZafMDIWindow *parentWindow = new ZafMDIWindow(1, 1, 60, 16,
    ZAF_MDI_PARENT);
parentWindow->AddGenericObjects(new ZafStringData("MDI
    Parent"));
ZafPullDownMenu *menuBar = new ZafPullDownMenu;
ZafPullDownItem *fileMenu = new ZafPullDownItem("File");
fileMenu->Add(new ZafPopUpItem("", ZAF_EXIT_OPTION));
menuBar->Add(fileMenu);
parentWindow->Add(menuBar);

// Create 2 child MDI windows.
ZafMDIWindow *child1 = new ZafMDIWindow(1, 1, 40, 10,
    ZAF_MDI_CHILD);
child1->AddGenericObjects(new ZafStringData("MDI Child 1"));
ZafMDIWindow *child2 = new ZafMDIWindow(2, 2, 40, 10,
    ZAF_MDI_CHILD);
child2->AddGenericObjects(new ZafStringData("MDI Child 2"));

// Add the children to the parent.
parentWindow->Add(child1);
parentWindow->Add(child2);

// Add the parent to the window manager.
zafWindowManager->Add(parentWindow);
```

Destructor

```
virtual ~ZafMDIWindow(void);
```

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

MDIType`ZafMDIType MDIType(void) const;`*SetMDIType*`ZafMDIType SetMDIType(ZafMDIType mdiType);`

`MDIType()` specifies whether an MDI window is an MDI parent window or an MDI child window. It returns `ZAF_MDI_PARENT` for MDI parent windows and `ZAF_MDI_CHILD` for MDI child windows. This attribute may not be modified after the object is on the screen, after the object has been added to another MDI window, or after another MDI window has been added to the object. Otherwise, it may be changed by calling `SetMDIType()`.

When the object gets the `S_INITIALIZE` message, this attribute is changed to appropriately indicate whether it is a parent or child. For example, if several MDI windows are stored in a data file as `ZAF_MDI_PARENT`, they may dynamically be added to another MDI parent window and during the `S_INITIALIZE` message handling, their `MDIType()` will automatically be changed to `ZAF_MDI_CHILD`. For this reason, care must be taken to not add a parent MDI window with child MDI windows to another parent MDI window (in which case there would be three levels of windows).

ZafMessageData

HotKeyChar

HotKeyIndex

Inheritance	ZafMessageData : ZafStringData : ZafFormatData : ZafData : ZafNotification, ZafElement
Declaration	#include <z_lang.hpp>
Description	ZafMessageData maintains an entry of a ZafLanguageData message table so that different languages may easily be supported by a single executable and a data file containing the application messages. ZafMessageData objects enable user interface level objects to be notified when a message table is reset (such as when the application's default language changes). See ZafI18nData::ResetI18n() for more information about internationalization of an application.
Constructors	The ZafMessageData constructor initializes the member variables associated with an instantiated ZafMessageData object. The default values set by the ZafMessageData constructor follows.

Member Initializations

ZafMessageData

HotKeyChar()	0
HotKeyIndex()	-1

ZafMessageData(const ZafIChar *value, ZafIChar hotKeyChar = 0, int index = -1);

This constructor creates a new message table entry with the string *value*, the hot key character *hotKeyChar* (converted to upper-case), and the hot key index *index*.

Members

HotKeyChar

HotKeyIndex

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

HotKeyChar() specifies the character associated with the message to be used as a hot key, and HotKeyIndex() specifies the zero-based index into the message of the character to be used as a hot key. See ZafButton::HotKeyChar() for more information on hot key characters, and see ZafButton::HotKeyIndex() for more information on hot key indices. SetHotKey() may be called to change the hot key information for a message table entry.

ZafMessageStruct

hotKeyChar	numberID	text
hotKeyIndex	stringID	

Inheritance Root struct

Declaration `#include <z_lang.hpp>`

Description ZafMessageStruct objects are used by [ZafLanguageData](#) in message tables to allow different languages to be supported by a single application and a data file containing the message tables. The last entry in a table of ZafMessageStruct entries should specify null in the text member to terminate a table search. See [ZafLanguageData::GetMessage\(\)](#) for more information.

Members

<i>text</i>	<p><code>ZafIChar *text;</code></p> <p>text specifies the message itself associated with the table entry. This member should be null for the last entry in a table.</p>
<i>numberID</i>	<p><code>ZafNumberID numberID;</code></p> <p>numberID is a unique numeric identification constant that identifies each message in the message table, and may be used to search for a message.</p>
<i>stringID</i>	<p><code>ZafIChar *stringID;</code></p> <p>stringID is a unique string identification constant that identifies each message in the message table, and is used when searching for a message.</p>
<i>hotKeyChar</i>	<p><code>ZafIChar hotKeyChar;</code></p> <p>hotKeyChar specifies the character associated with the message to be used as a hot key. See ZafButton::HotKeyChar() for more information on hot key characters.</p>
<i>hotKeyIndex</i>	<p><code>int hotKeyIndex;</code></p> <p>hotKeyIndex specifies the zero-based index into the message of the character to be used as a hot key. See ZafButton::HotKeyIndex() for more information on hot key indices.</p>

ZafMessageWindow

ClearMessageFlags	IconImage	MessageFlags
DefaultMessageFlag	Message	

Inheritance ZafMessageWindow : [ZafDialogWindow](#) : [ZafWindow](#) :
 ([ZafWindowObject](#) : [ZafElement](#)), [ZafList](#)

Declaration `#include <z_msgwin.hpp>`

Description ZafMessageWindow provides support for easily presenting a message dialog and returning a standard reply from the end user. ZafMessageWindows can automatically include icons and buttons based on “option flags” specified by the programmer. For example, a message window could be used to present an error message and allow the user to select “OK” or “Cancel”. ZafMessageWindow is derived from ZafDialogWindow, and is therefore a true dialog window class.

A message window is not moveable or sizeable by nature, since its border indicates a dialog window rather than movability and sizability and since a message window’s contents should promote understandability. A message window is also by its nature not a temporary window, since the end user is expected to dismiss it after acknowledging the message.

Constructors All ZafMessageWindow constructors initialize the member variables associated with an instantiated ZafMessageWindow object. The default values set by the ZafMessageWindow 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 ZafMessageWindow. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafMessageWindow

<code>DefaultMessageFlag()</code>	user-supplied parameter
<code>IconImage()</code>	user-supplied parameter
<code>Message()</code>	user-supplied parameter
<code>MessageFlags()</code>	user-supplied parameter

ZafWindow

<code>Destroyable()</code>	false [†]
<code>Locked()</code>	false [†]
<code>Maximized()</code>	false [†]
<code>Minimized()</code>	false [†]

Member Initializations

<code>Moveable()</code>	<code>true</code> [†]
<code>SelectionType()</code>	<code>ZAF_SINGLE_SELECTION</code> [†]
<code>Sizeable()</code>	<code>false</code> [†]
<code>Temporary()</code>	<code>false</code> [†]

ZafWindowObject

<code>AcceptDrop()</code>	<code>false</code> [†]
<code>AutomaticUpdate()</code>	<code>true</code> [†]
<code>Bordered()</code>	<code>false</code> [†]
<code>Disabled()</code>	<code>false</code> [†]
<code>HelpObjectTip()</code>	<code>null</code> [†]
<code>RegionType()</code>	<code>ZAF_INSIDE_REGION</code> [†]

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_MESSAGE_WINDOW</code>
<code>ClassName()</code>	<code>"ZafMessageWindow"</code>

```
ZafMessageWindow(const ZafIChar *title, ZafIconImage
    iconImage, ZafDialogFlags msgFlags, ZafDialogFlags
    defaultFlag, const ZafIChar *format, ...);
```

This constructor is useful in straight-code situations, particularly if the `ZafTitle` object is to create, maintain and destroy its own `ZafStringData` object automatically. The message window is sized appropriately, so that all the information fits in the window, and the message window is centered on the main screen when added to the window manager via the `Control()` method provided by the base `ZafDialogWindow` class.

title specifies the text to appear in the title bar.

iconImage specifies the icon that is placed in the message window (see [IconImage](#)() for more information).

msgFlags specifies the buttons that are placed in the message window (see [MessageFlags](#)() for more information).

defaultFlag specifies the default button (see [DefaultMessageFlag](#)() for more information).

format, along with the variable arguments that follow, specify the formatted textual message to be added to the message window using format options similar to those used by the `printf` functions. Refer to standard library documentation for detailed information on `printf` functions and conversion characters.

Note: The maximum length of the message text is 1024 characters when using this constructor.

```
ZafMessageWindow(ZafStringData *title, ZafIconImage  
    iconImage, ZafDialogFlags msgFlags, ZafDialogFlags  
    defaultFlag, const ZafIChar *format, ...);  
ZafMessageWindow(ZafStringData *title, ZafIconImage  
    iconImage, ZafDialogFlags msgFlags, ZafDialogFlags  
    defaultFlag, ZafUInt16 bufferSize, const ZafIChar  
    *format, ...);
```

These constructors are useful in straight-code situations, particularly if a `ZafStringData` object has already been created for use by the `ZafTitle` object. *title* specifies the `ZafStringData` containing the text to appear in the title bar. The first allocates a temporary 1024 character buffer for the entire message, and the second allocates a temporary buffer of *bufferSize* characters for the entire message. See the previous constructor for a description of the other parameters.

```
ZafMessageWindow(const ZafMessageWindow &copy);
```

The copy constructor creates a new `ZafMessageWindow` object and initializes its data from *copy*.

```
ZafMessageWindow(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 message window follows:

```
// Create a message window.  
ZafMessageWindow *window = new ZafMessageWindow("Error",  
    ZAF_EXCLAMATION_ICON, ZAF_DIALOG_OK, ZAF_DIALOG_OK, "You are  
    about to commit a grievous error!");  
  
// Add the message window to the window manager.  
// Use Control() only if Modal() is true.  
// ZafMessageWindows should always be Modal().  
// Control() returns only when the user has made a selection.  
window->Control();
```

Destructor

```
virtual ~ZafMessageWindow(void);
```

The destructor is used to free the memory associated with a ZafMessageWindow object. It chains to the ZafDialogWindow, ZafWindow, ZafWindowObject, ZafList, and ZafElement destructors. For more information on child object deletion, see [ZafWindow::~~ZafWindow\(\)](#).

Members

ClearMessageFlags

```
ZafDialogFlags ClearMessageFlags(ZafDialogFlags
    msgFlags);
```

ClearMessageFlags() clears only the flags specified by *msgFlags* out of the MessageFlags() attribute. Flags other than those specified in *msgFlags* are left untouched. See [MessageFlags\(\)](#) for more information about these flags. ClearMessageFlags() may be called to clear flags any time before the message window appears on the window manager, but has no effect on a visible MessageWindow.

DefaultMessage-Flag

```
ZafDialogFlags DefaultMessageFlag(void) const;
ZafDialogFlags SetDefaultMessageFlag(ZafDialogFlags
    msgFlags);
```

DefaultMessageFlag() specifies the default button on the message window. *msgFlags* specifies the button the same way the MessageFlags() specifies a button. SetDefaultMessageFlag() may be called to change the DefaultMessageFlag() any time before the message window appears on the window manager.

IconImage

```
ZafIconImage IconImage(void) const;
ZafIconImage SetIconImage(ZafIconImage iconImage);
```

IconImage() specifies the icon that appears on the message window. SetIconImage() may be called to change IconImage() any time before the message window appears on the window manager. The following images are defined by ZAF:

IconImage()	Description
ZAF_APPLICATION_ICON	Provides a default application icon
ZAF_ASTERISK_ICON	Provides an asterisk icon
ZAF_EXCLAMATION_ICON	Provides an exclamation icon
ZAF_HAND_ICON	Provides a hand icon
ZAF_QUESTION_ICON	Provides a question icon

Message

```
const ZafIChar *Message(void) const;
void SetMessage(const ZafIChar *format, ...);
void SetMessage(ZafUInt16 bufferSize, const ZafIChar
    *format, ...);
```

`Message()` is the textual message that appears on the message window. `SetMessage()` may be called to change the message any time before the message window appears on the window manager.

format, along with the variable arguments that follow, specify the formatted textual message to be added to the message window using format options similar to those used by the `printf` functions. Refer to standard library documentation for detailed information on `printf` functions and conversion characters. The first `SetMessage()` allocates a temporary 1024 character buffer for the entire message, and the second allocates a temporary buffer of *bufferSize* characters for the entire message.

MessageFlags

```
ZafDialogFlags MessageFlags(void) const;
ZafDialogFlags SetMessageFlags(ZafDialogFlags msgFlags);
```

`MessageFlags()` specifies the buttons that are placed on the message window, according to the bit flags described below OR'd together. `SetMessageFlags()` sets the flags specified in *msgFlags*, and does not modify other flags previously set (see [ClearMessageFlags\(\)](#) for information on how to clear flags). `SetMessageFlags()` may be called to add flags any time before the message window appears on the window manager. The following `MessageFlags()` are defined by ZAF:

MessageFlags()	Description
ZAF_DIALOG_OK	Provides an "OK" button
ZAF_DIALOG_CANCEL	Provides a "Cancel" button
ZAF_DIALOG_YES	Provides a "Yes" button
ZAF_DIALOG_NO	Provides a "No" button
ZAF_DIALOG_ABORT	Provides an "Abort" button
ZAF_DIALOG_RETRY	Provides a "Retry" button
ZAF_DIALOG_IGNORE	Provides an "Ignore" button
ZAF_DIALOG_HELP	Provides a "Help" button

Following is a code snippet showing how to make an OK/Cancel message window:

```
// Create a message window with OK and Cancel buttons.
// The OK button is the default button.
ZafMessageWindow *window = new ZafMessageWindow("Error",
    ZAF_EXCLAMATION_ICON, ZAF_DIALOG_OK | ZAF_DIALOG_CANCEL,
    ZAF_DIALOG_OK, "You have committed a grievous error! By
    selecting the OK button, you admit your guilt...");

// Add the message window to the window manager.
// Use Control() only if Modal() is true.
// ZafMessageWindows should always be Modal().
// Control() returns only when the user has made a selection.
if (window->Control() == S_DLG_CANCEL)
    break;
```

ZafMinimizeButton

Inheritance	<code>ZafMinimizeButton : ZafButton : ZafWindowObject : ZafElement</code>
Declaration	<code>#include <z_min.hpp></code>
Description	The ZafMinimizeButton object may only be added to a ZafWindow. The ZafMinimizeButton is the minimize button decoration on a ZafWindow, and is generally drawn by the environment. The ZafMinimizeButton object is used to minimize the parent window with a ZafMouse device.
Constructors	All ZafMinimizeButton constructors initialize the member variables associated with an instantiated ZafMinimizeButton object. The default values set by the ZafMinimizeButton 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 ZafMinimizeButton. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafButton

<code>AllowDefault()</code>	<code>false</code> [†]
<code>AllowToggling()</code>	<code>false</code> [†]
<code>AutoRepeatSelection()</code>	<code>false</code> [†]
<code>AutoSize()</code>	<code>true</code> [†]
<code>ButtonType()</code>	<code>ZAF_3D_BUTTON</code> [†]
<code>Depth()</code>	<code>1</code> [†]
<code>HotKeyChar()</code>	<code>0</code> [†]
<code>HotKeyIndex()</code>	<code>-1</code> [†]
<code>HzJustify()</code>	<code>ZAF_HZ_CENTER</code> [†]
<code>SelectOnDoubleClick()</code>	<code>false</code> [†]
<code>SelectOnDownClick()</code>	<code>false</code> [†]
<code>SendMessageText()</code>	<code>null</code> [†]
<code>SendMessageWhenSelected()</code>	<code>true</code> [†]
<code>Value()</code>	Used internally by ZAF [†]
<code>VtJustify()</code>	<code>ZAF_VT_CENTER</code> [†]

ZafWindowObject

<code>AcceptDrop()</code>	<code>false</code> [†]
<code>Bordered()</code>	<code>false</code> [†]
<code>CopyDraggable()</code>	<code>false</code> [†]

Member Initializations

Disabled()	false [†]
Focus()	false [†]
HelpContext()	null [†]
HelpObjectTip()	null [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
Noncurrent()	true [†]
ParentDrawBorder()	false [†]
ParentDrawFocus()	false [†]
ParentPalette()	false [†]
QuickTip()	null [†]
RegionType()	ZAF_AVAILABLE_REGION [†]
Selected()	false [†]
SupportObject()	true [†]
SystemObject()	false
UserFunction()	null [†]

ZafElement

ClassID()	ID_ZAF_MINIMIZE_BUTTON
ClassName()	"ZafMinimizeButton"
NumberID()	ZAF_NUMID_MINIMIZE
StringID()	"ZAF_NUMID_MINIMIZE"

ZafMinimizeButton(void);

This constructor is useful in straight-code situations to create a ZafMinimize-Button object.

ZafMinimizeButton(const ZafMinimizeButton ©);

The copy constructor creates a new ZafMinimizeButton object and initializes its data from *copy*.

**ZafMinimizeButton(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 minimize button follows:

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

Destructor

```
virtual ~ZafMinimizeButton(void);
```

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

ZafMouse

	Event	ImageType	Poll
Inheritance	ZafMouse : ZafDevice : ZafElement		
Declaration	#include <z_mouse2.hpp>		
Description	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.		
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 "	
NumberID()		ID_ZAF_MOUSE	
StringID()		" 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. *state* specifies the initial state of the device, and *imageType* specifies the initial image type displayed by the device (see ImageType() for more information).

```
ZafMouse(const ZafMouse &copy);
```

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.

Event

```
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.

ImageType

```
ZafDeviceImage ImageType(void) const;  
virtual ZafDeviceImage SetImageType(ZafDeviceImage  
    imageType);
```

ImageType() 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_DRAG	causes the mouse to show the generic drag image
DM_DRAG_COPY	causes the mouse to show the copy-drag image
DM_DRAG_COPY_MULTIPLE	causes the mouse to show the copy-drag multiple image
DM_DRAG_LINK	causes the mouse to show the link-drag image
DM_DRAG_LINK_MULTIPLE	causes the mouse to show the link-drag multiple image
DM_DRAG_MOVE	causes the mouse to show the move-drag image
DM_DRAG_MOVE_MULTIPLE	causes the mouse to show the move-drag multiple 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

ImageType()	Description
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

Poll

```
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, Poll() simply blocks mouse events from coming through ZAF's event manager queue if the ZafMouse's state is D_OFF.

ZafMouseData

Array	HotSpotX	Width
Clear	HotSpotY	operator =
Height	SetMouse	

Inheritance	ZafMouseData : (ZafImageData : ZafData : (ZafNotification, ZafElement)), ZafMouseStruct
Declaration	#include <z_mouse1.hpp>
Description	ZafMouseData objects can be used to store and manipulate mouse information. ZafMouseData is most often used in conjunction with the ZafMouse device object.
Constructors	All ZafMouseData constructors initialize the member variables associated with an instantiated ZafMouseData object. The default values set by the ZafMouseData and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafMouseData

Array()	null
HotSpotX()	0
HotSpotY()	0

ZafElement

ClassID()	ID_ZAF_MOUSE_DATA
ClassName()	" ZafMouseData "

```
ZafMouseData(const ZafImageStruct &data);  
ZafMouseData(const ZafMouseStruct &data);
```

These constructors allocate a ZafMouseData instance and initialize its data to the values in *data*.

```
ZafMouseData(const ZafMouseData &copy);
```

The copy constructor creates a new ZafMouseData object and initializes its data from *copy*.

```
ZafMouseData(const ZafIChar *name, ZafDataPersistence
&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 code snippet shows how to create a `ZafMouseData` object:

```
// Create a mouse image array.
#define gnd ZAF_CLR_BACKGROUND
#define blk ZAF_CLR_BLACK
#define wte ZAF_CLR_WHITE
static ZafLogicalColor ZAF_FARDATA mouseArray[160] =
{
    blk,blk,gnd,gnd,gnd,gnd,gnd,gnd,gnd,gnd,
    blk,wte,blk,gnd,gnd,gnd,gnd,gnd,gnd,gnd,
    blk,wte,wte,blk,gnd,gnd,gnd,gnd,gnd,gnd,
    blk,wte,wte,wte,blk,gnd,gnd,gnd,gnd,gnd,
    blk,wte,wte,wte,wte,blk,gnd,gnd,gnd,gnd,
    blk,wte,wte,wte,wte,wte,blk,gnd,gnd,gnd,
    blk,wte,wte,wte,wte,wte,wte,blk,gnd,gnd,
    blk,wte,wte,wte,wte,wte,wte,wte,blk,gnd,
    blk,wte,wte,wte,wte,wte,wte,wte,blk,blk,
    blk,wte,wte,wte,wte,wte,blk,blk,blk,gnd,
    blk,wte,wte,blk,wte,wte,blk,gnd,gnd,gnd,
    blk,blk,blk,blk,blk,wte,wte,blk,gnd,gnd,
    gnd,gnd,gnd,gnd,blk,wte,wte,blk,gnd,gnd,
    gnd,gnd,gnd,gnd,gnd,blk,wte,wte,blk,gnd,
    gnd,gnd,gnd,gnd,gnd,blk,wte,wte,blk,gnd,
    gnd,gnd,gnd,gnd,gnd,gnd,blk,blk,gnd,gnd
};
static ZafMouseStruct mouseStruct(10, 16, mouseArray, 0, 0,
    true);

// Create ZafMouseData objects based on the mouse image array.
ZafMouseData mouse1(mouseStruct);
ZafMouseData mouse2(mouse1);
```

Destructor

```
virtual ~ZafMouseData(void);
```

This virtual destructor is used to free the memory associated with an instantiated `ZafMouseData` object.

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.

Array `ZafLogicalColor *Array(void) const;`

`Array()` returns the portable logical color array that the mouse data is based on. This array is converted to a native environment mouse image. Each element of the array is a ZAF logical color. See [ZafDisplay](#) for more information.

Clear `virtual void Clear(void);`

`Clear()` destroys the portable `Array()` if `StaticArray()` is false, and it destroys the environment handle if `StaticHandle()` is false. Regardless of `StaticArray()` and `StaticHandle()`, the portable `Array()` and the environment handle are both set to null, effectively clearing the mouse data.

Height `int Height(void) const;`

`Height()` returns the height of the mouse data.

HotSpotX `int HotSpotX(void) const;`

HotSpotY `int HotSpotY(void) const;`

`virtual ZafError SetHotSpot(int hotSpotX, int hotSpotY);`

The hot spot is the pixel in the mouse data where a mouse click occurs, and is relative to the top left corner of the mouse data. `HotSpotX()` returns the zero-based position of the hot spot along the x-axis, and `HotSpotY()` returns the zero-based position of the hot spot along the y-axis. The hot spot may be changed by calling `SetHotSpot()`. `SetHotSpot()` always returns `ZAF_ERROR_NONE`.

SetMouse `virtual ZafError SetMouse(int width, int height,
 ZafLogicalColor *array);`

`virtual ZafError SetMouse(const ZafMouseStruct &mouse);`

These functions copy the data passed in to the `ZafMouseData` object. *width* and *height* are the width and height of *array*, respectively. If `StaticArray()` is true, *array* becomes `Array()`; otherwise, a new array is created for `Array()`. In the second function, the data is copied from *mouse*. These functions always return `ZAF_ERROR_NONE`.

Width

```
int width(void) const;
```

Width() returns the width of the mouse data.

operator =

```
ZafMouseData &operator=(const ZafMouseData &mouse);
```

This operator copies the data from mouse into this ZafMouseData object.

ZafMouseStruct

	<div>hotSpotXhotSpotYStaticHandle</div>
Inheritance	ZafMouseStruct : ZafImageStruct
Declaration	#include <z_dsp.hpp>
Description	ZafMouseStruct is used to store mouse image information. The base ZafImageStruct stores the portable image array using elements of ZafLogicalColors, and ZafMouseStruct defines additional members that store environment-specific structures for the mouse image filled by the ZafDisplay conversion function ZafDisplay::ConvertToOSMouse().
Constructors	All ZafMouseStruct constructors initialize the member variables associated with an instantiated ZafMouseStruct object. The default values set by the ZafMouseStruct constructors follow.

Member Initializations

ZafMouseStruct

hotSpotX	0
hotSpotY	0
StaticHandle()	false

ZafImageStruct

array	null
height	0
StaticArray()	false
width	0

ZafMouseStruct(void);

This constructor allocates a ZafMouseStruct instance and initializes its data to indicate that no mouse image information has been set.

ZafMouseStruct(const ZafImageStruct &data);

This constructor allocates a ZafMouseStruct instance and initializes its data to the values in data. The environment-specific mouse image information is initialized to indicate that the mouse image has not yet been converted.

```
ZafMouseStruct(int width, int height, ZafLogicalColor
    *array, int hotSpotX, int hotSpotY, bool staticArray);
```

This constructor allocates a `ZafMouseStruct` instance and initializes its data to the values passed in. *width* and *height* indicate the size of the image, *array* specifies a pointer to the portable array of `ZafLogicalColors`, *hotSpotX* and *hotSpotY* indicate the position of the hot spot within the image (see the explanation of the members `hotSpotX` and `hotSpotY` below), and *staticArray* indicates if array is declared static.

The following code snippet shows how to create a `ZafMouseStruct` object:

```
// Create a mouse image structure.
#define gnd ZAF_CLR_BACKGROUND
#define blk ZAF_CLR_BLACK
#define wte ZAF_CLR_WHITE
static ZafLogicalColor ZAF_FARDATA mouseArray[160] =
{
    blk,blk,gnd,gnd,gnd,gnd,gnd,gnd,gnd,gnd,
    blk,wte,blk,gnd,gnd,gnd,gnd,gnd,gnd,gnd,
    blk,wte,wte,blk,gnd,gnd,gnd,gnd,gnd,gnd,
    blk,wte,wte,wte,blk,gnd,gnd,gnd,gnd,gnd,
    blk,wte,wte,wte,wte,blk,gnd,gnd,gnd,gnd,
    blk,wte,wte,wte,wte,wte,blk,gnd,gnd,gnd,
    blk,wte,wte,wte,wte,wte,wte,blk,gnd,gnd,
    blk,wte,wte,wte,wte,wte,wte,wte,blk,gnd,
    blk,wte,wte,wte,wte,wte,wte,wte,blk,blk,
    blk,wte,wte,wte,wte,wte,blk,blk,blk,gnd,
    blk,wte,wte,blk,wte,wte,blk,gnd,gnd,gnd,
    blk,blk,blk,blk,blk,wte,wte,blk,gnd,gnd,
    gnd,gnd,gnd,gnd,blk,wte,wte,blk,gnd,gnd,
    gnd,gnd,gnd,gnd,gnd,blk,wte,wte,blk,gnd,
    gnd,gnd,gnd,gnd,gnd,blk,wte,wte,blk,gnd,
    gnd,gnd,gnd,gnd,gnd,gnd,blk,blk,gnd,gnd
};
static ZafMouseStruct mouseStruct(10, 16, mouseArray, 0, 0,
    true);
```

Members

hotSpotX

int **hotSpotX**, **hotSpotY**;

hotSpotY

The hot spot is the pixel in the mouse image where a mouse click occurs, and is relative to the top left corner of the mouse image. `hotSpotX` specifies the zero-based position of the hot spot along the x-axis, and `hotSpotY` specifies the zero-based position of the hot spot along the y-axis.

StaticHandle

```
bool StaticHandle(void) const;  
bool SetStaticHandle(bool staticHandle);
```

If `StaticHandle()` is true, the environment-specific information (generally known as a handle) of the `ZafMouseStruct` is recognized as static, so that it will not be deleted by ZAF, and may be used by several `ZafMouseStruct` objects. This attribute is initialized to false, but it may be changed with `SetStaticHandle()`. Both functions return the current value of the `StaticHandle()` attribute.

ZafMSWindowsApp

available3D	dragStartPosition	MDIFrameJumpProc
commonControlsAvailable	dragTest	mouseTimerID
convertText	dropDownCombo	native3D
CreateSubclassedWindow	hInstance	ObjectFromHandle
ctl3dModule	HotKeyText	objectLowWord
Ctl3dRegister	InitializeWrappers	objectHighWord
Ctl3dEnabled	JumpProc	windowsPlatform
Ctl3dAutoSubclass	FrameJumpProc	windowsVersion
Ctl3dUnregister	FrameProc	
Ctl3dSubclassCtl	MDIChildJumpProc	

Inheritance	Root class
Declaration	<code>#include <w_app.hpp></code>
Description	<p>ZafMSWindowsApp is an advanced, internal class used by the Microsoft Windows versions of ZAF 5 and encapsulates various Microsoft Windows application-specific information. Most of the members of ZafMSWindowsApp are public statics and may be accessed from anywhere in the application, but most of the members are "advanced" and should normally not be accessed by the programmer. ZafMSWindows is provided for advanced Microsoft Windows programmers who may need access to this information for custom, non-portable development. A ZafMSWindowsApp instance is created by ZafApplication and destroyed on exit.</p>
Constructor	<p><code>ZafMSWindowsApp(HINSTANCE hInstance);</code></p> <p>The ZafMSWindowsApp constructor initializes the information associated with an instantiated ZafMSWindowsApp object. <i>hInstance</i> should be the HINSTANCE passed into WinMain().</p>
Destructor	<p><code>virtual ~ZafMSWindowsApp(void);</code></p> <p>The destructor is used to free the memory associated with a ZafMSWindowsApp object. Generally, the programmer will not directly destroy a ZafMSWindowsApp object, since it is automatically destroyed when the application closes down.</p>

Members

available3D

static bool **available3D**;

This member is only available when the macro ZAF_MSWINDOWS_3D is defined, and indicates if 3D control support is available (either built-into the system, or through a DLL).

*commonControls-
Available*

static bool **commonControlsAvailable**;

If commonControlsAvailable is true, the system contains support for common controls (such as in Microsoft Windows 95); otherwise common controls are not supported. This member is only available in the 32-bit version of the ZAF libraries for Microsoft Windows.

convertText

static bool **convertText**;

If convertText is false, the native system string encoding matches the ZAF library string encoding (either ISO or Unicode), so no conversion is necessary between ZafIChar strings and native strings; otherwise the conversion routines found in ZafCodeSetData are used to convert strings.

*CreateSubclassed-
Window*

```
static HWND CreateSubclassedWindow(ZafWindowObject
    *object, const ZafIChar *lpClassName, DWORD dwStyle,
    const ZafIChar *text = NULL, HWND hWndParent = 0,
    DWORD dwExStyle = WS_EX_NOPARENTNOTIFY, HINSTANCE
    hInstance = 0, LPVOID lpParam = 0);
```

CreateSubclassedWindow() is the ZAF substitute for the Microsoft Windows API functions CreateWindow() and CreateWindowEx(). CreateSubclassedWindow() creates a Microsoft Windows "window" and then subclasses from it so that Microsoft Windows messages are routed through the object's Event() function. *object* is a pointer to the ZAF window object to be associated with the new "window". *text* is the ZafIChar version of the text to be associated with the new "window", and is converted to the native Microsoft Windows string format before being passed to CreateWindowEx(). *lpClassName*, *dwStyle*, *hWndParent*, *dwExStyle*, *hInstance*, and *lpParam* are the equivalent parameters passed to CreateWindowEx(). The HWND to be associated with the new "window" is returned, and is normally assigned to the ZAF object's screenID.

ctl3dModule
Ctl3dRegister
Ctl3dEnabled

```
static HMODULE ctl3dModule;  
static BOOL (WINAPI *Ctl3dRegister)(HINSTANCE);  
static BOOL (WINAPI *Ctl3dEnabled)(void);
```

```
Ctl3dAutoSubclass     static BOOL (WINAPI *Ctl3dAutoSubclass)(HINSTANCE);  
Ctl3dUnregister       static BOOL (WINAPI *Ctl3dUnregister)(HINSTANCE);  
Ctl3dSubclassCtl      static BOOL (WINAPI *Ctl3dSubclassCtl)(HWND);
```

These members are pointers to the CTL3D modules and functions for non-native 3D support, and are resolved by loading `ctl3d32.dll` (for 32-bit) or `ctl3dv2` (for 16-bit). They are for internal use only, and are not intended to be used by the programmer.

```
dragStartPosition     static POINT ZAF_FARDATA dragStartPosition;  
                       static POINTS dragStartPosition;
```

These members are used internally by the ZAF libraries to support drag and drop under Microsoft Windows, and should not be accessed by the programmer.

```
dragTest              static bool dragTest;
```

This member is used internally by the ZAF libraries to support drag-and-drop under Microsoft Windows, and should not be accessed by the programmer.

```
dropDownCombo        static HWND dropDownCombo;
```

This member is used internally by the ZAF libraries to support combo-boxes under Microsoft Windows, and should not be accessed by the programmer. It is used to keep a handle to the combo-box that is currently open.

```
hInstance             static HINSTANCE hInstance;
```

This member stores a handle to the application instance, which is passed-into `WinMain()` at application start-up.

```
HotKeyText            static ZafIChar *HotKeyText(const ZafIChar *text, int  
                           hotKeyIndex);
```

`HotKeyText()` returns a newly-created string that contains special embedded characters required by the Microsoft Windows API to denote a hot key. *text* is the original ZAF string to be converted. *hotKeyIndex* is the zero-based index of the character in *text* to be displayed as a hot key. The programmer must delete the return value when it is no longer needed.

InitializeWrappers

```
void InitializeWrappers(void);
```

When running in Unicode character mode, `InitializeWrappers()` initializes function callback pointers used by the ZAF library for Microsoft Windows. This is an advanced routine usually called in the constructor, and should normally not be called by the programmer.

JumpProc

```
static LRESULT CALLBACK JumpProc(HWND hwnd, UINT wMsg,  
    WPARAM wParam, LPARAM lParam);
```

`JumpProc()` is the callback function (the Microsoft Windows “window procedure”) called by Windows and used to send native messages to the appropriate ZAF `Event()` function for message handling. The parameters are normal `WindowProc` parameters as documented in the Microsoft Windows API documentation. `JumpProc()` is an advanced routine, and should normally not be used by the programmer.

FrameJumpProc

```
static LRESULT CALLBACK FrameJumpProc(HWND hwnd, UINT  
    wMsg, WPARAM wParam, LPARAM lParam);
```

`FrameJumpProc()` is the callback function (window procedure for frame windows) called by Microsoft Windows and used to pass the appropriate default function to `FrameProc()` for normal frame message handling. The parameters are normal `WindowProc` parameters as documented in the Microsoft Windows API documentation. `FrameJumpProc()` is an advanced routine, and should normally not be used by the programmer.

FrameProc

```
static LRESULT FrameProc(LRESULT (DefaultProc)(HWND,  
    UINT, WPARAM, LPARAM), HWND hwnd, UINT wMsg, WPARAM  
    wParam, LPARAM lParam);
```

`FrameProc()` is the common code for all frame procedures and either routes them to the appropriate ZAF objects for processing, or calls the default callback routine for base class processing by the Microsoft Windows API. The first parameter is a pointer to the default callback routine (`DefWindowProc`, `DefFrameProc` or `DefMDIChildProc`). `hwnd`, `wMsg`, `wParam`, and `lParam` are normal `WindowProc` parameters as documented in the Microsoft Windows API documentation.

MDIChildJumpProc

```
static LRESULT CALLBACK MDIChildJumpProc(HWND hwnd, UINT  
    wMsg, WPARAM wParam, LPARAM lParam);
```

`MDIChildJumpProc()` is the callback function (window procedure for MDI child frames) called by Microsoft Windows and used to pass the appropriate

default function to `FrameProc()` for MDI child frame message handling. The parameters are normal `WindowProc` parameters as documented in the Microsoft Windows API documentation. `MDIChildJumpProc()` is an advanced routine, and should normally not be used by the programmer.

*MDIFrameJump-
Proc*

```
static LRESULT CALLBACK MDIFrameJumpProc(HWND hwnd, UINT  
wMsg, WPARAM wParam, LPARAM lParam);
```

`MDIFrameJumpProc()` is the callback function (window procedure for MDI frame windows) called by Microsoft Windows and used to pass the appropriate default function to `FrameProc()` for MDI parent frame message handling. The parameters are normal `WindowProc` parameters as documented in the Microsoft Windows API documentation. `MDIFrameJumpProc()` is an advanced routine, and should normally not be used by the programmer.

mouseTimerID

```
static UINT mouseTimerID;
```

This member is used internally by the ZAF libraries to support the `N_MOUSE_ENTER` and `N_MOUSE_LEAVE` events under Microsoft Windows, and should not be accessed by the programmer.

native3D

```
static bool native3D;
```

This member is only available when the macro `ZAF_MSWINDOWS_3D` is defined, and indicates whether 3-D is supported natively.

ObjectFromHandle

```
static ZafWindowObject *ObjectFromHandle(HWND hwnd);
```

`ObjectFromHandle()` returns a pointer to the ZAF window object that corresponds to the Microsoft Windows handle `hwnd` (`hwnd` is normally equivalent to the object's `screenID`).

*objectLowWord
objectHighWord*

```
static ATOM objectLowWord;  
static ATOM objectHighWord;
```

These members are used internally by the ZAF libraries for all compilers except Watcom to get pointers to ZAF objects from a window's properties under 16-bit versions of Microsoft Windows, and should not be accessed by the programmer.

```
static char *objectLowWord;  
static char *objectHighWord;
```


These members are used internally by the ZAF libraries for the Watcom compiler to get pointers to ZAF objects from a window's properties under 16-bit versions of Microsoft Windows, and should not be accessed by the programmer. (The Watcom-specific differences should disappear in future versions of the Watcom compiler.)

windowsPlatform

```
static ZafWindowsPlatform windowsPlatform;
```

`windowsPlatform` specifies which flavor of Microsoft Windows the application is running under. The possible values for this member are as follows:

ZafWindowsPlatform	Description
ZAF_WIN16	Microsoft Windows (16-bit under Windows 3.x)
ZAF_WIN32S	Microsoft Windows (32-bit under Windows 3.x)
ZAF_WIN32_WINDOWS	Microsoft Windows (32-bit, such as Windows 95)
ZAF_WIN32_NT	Microsoft Windows NT (32-bit)

windowsVersion

```
static int windowsVersion;
```

`windowsVersion` specifies which version of Microsoft Windows the application is running under. For example, `windowsVersion` would be 400 under Microsoft Windows NT version 4.0, or it would be 310 for Microsoft Windows version 3.1.

ZafNotebook

CurrentPage	SetTabHotKey	TabHotKeyIndex
FirstPage	TabHeight	TabText
LastPage	TabHotKeyChar	TabWidth

Inheritance ZafNotebook : ZafWindow : (ZafWindowObject : ZafElement),
 ZafList

Declaration #include <z_notebk.hpp>

Description ZafNotebook is a container object used to logically group other objects into tabbed pages. A ZafWindow object with a ZafTitle is added to a ZafNotebook to create a page. The text on the ZafTitle is copied to the corresponding tab. ZafNotebook tabs are scrollable, so more tabs may be on a ZafNotebook than are visible in the available space.

This class is not available in the Personal version of ZAF, but is included with the Registered and Professional versions.

Constructors All ZafNotebook constructors initialize the member variables associated with an instantiated ZafNotebook object. The default values set by the ZafNotebook 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 ZafNotebook. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafNotebook

CurrentPage()	-1
TabHeight()	0
TabHotKeyChar()	'\0'
TabHotKeyIndex()	-1
TabWidth()	0

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Modal()	false [†]
Moveable()	false [†]

Member Initializations

RegionType()	ZAF_AVAILABLE_REGION
SelectionType()	ZAF_SINGLE_SELECTION [†]
Sizeable()	false [†]
Temporary()	false [†]

ZafWindowObject

AcceptDrop()	false [†]
--------------	--------------------

ZafElement

ClassID()	ID_ZAF_NOTEBOOK
ClassName()	"ZafNotebook"

ZafNotebook(int left, int top, int width, int height);

This constructor is useful in straight-code situations. *left*, *top*, *width*, and *height* specify the position and size on a parent window. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired. See [ZafWindowObject::CoordinateType\(\)](#) for more information.

ZafNotebook(const ZafNotebook ©);

The copy constructor creates a new ZafNotebook object and initializes its data from *copy*.

ZafNotebook(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 code shows how to create a ZafNotebook object.

```
// Create a sample window with a notebook.
ZafWindow *window = new ZafWindow(10, 10, 40, 10);
window->AddGenericObjects(new ZafStringData("Notebook
    Window"));
ZafNotebook *notebook = new ZafNotebook(0, 0, 40, 10);

// Create Page 1 with a string and add the page to the notebook.
ZafWindow *page = new ZafWindow(0, 0, 40, 4);
page->Add(new ZafTitle("Page 1"));
```

```
page->Add(new ZafString(2, 0, 20, "string1", 100));
notebook->Add(page);

// Create Page 2 with a string and add the page to the notebook.
page = new ZafWindow(0, 0, 40, 4);
page->Add(new ZafTitle("Page 2"));
page->Add(new ZafString(4, 1, 20, "string2", 100));
notebook->Add(page);

// Finally, add the notebook to the window.
window->Add(notebook);
```

Destructor

```
virtual ~ZafNotebook(void);
```

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

CurrentPage

```
int CurrentPage(void) const;
virtual int SetCurrentPage(int currentPage);
virtual int SetCurrentPage(ZafWindowObject *currentPage);
```

CurrentPage() returns the zero-based index of the current page for the ZafNotebook selected either by the end user, or programmatically. SetCurrentPage() may be called to change the current page programmatically by passing as currentPage either the zero-based index or a pointer to the ZafWindow page object for the page to be made current.

The ZafNotebook object receives an N_CHANGE_PAGE event whenever the page is changed either programmatically, or the end user intervention.

FirstPage

```
int FirstPage(void) const;
```

FirstPage() returns the zero-based index of the first page on the ZafNotebook. If there are no pages, it returns -1; otherwise, it returns 0.

LastPage

```
int LastPage(void) const;
```

LastPage() returns the zero-based index of the last page on the ZafNotebook. If there are no pages, it returns -1.

TabHeight

```
int TabHeight(void);
virtual int SetTabHeight(int tabHeight);
```

TabHeight() specifies in pixel coordinates the height of the notebook tabs. By default this attribute is 0, which means that the default height for each environment is used, but SetTabHeight() may be used to change it.

TabHotKeyChar *TabHotKeyIndex* *SetTabHotKey*

```
ZafIChar TabHotKeyChar(int page) const;
int TabHotKeyIndex(int page) const;
virtual ZafIChar SetTabHotKey(int page, ZafIChar
    hotKeyChar, int index = -1);
```

ZafNotebook's tab objects are implemented with the ZafButton class, and thus may utilize hot keys. These functions provide access to the tab objects' hot key information. *page* is the zero-based index of the tab whose hot key information is being accessed. The other parameters to these functions are similar to those passed into ZafButton::HotKeyChar(), ZafButton::HotKeyIndex(), and ZafButton::SetHotKey(). See [ZafButton](#) for more information.

TabText

```
virtual const ZafIChar *TabText(int page);
virtual ZafError SetTabText(const ZafIChar *text, int
    page);
```

TabText() returns the text of the tab corresponding to the notebook page with zero-based offset *page* by getting the text on the corresponding ZafTitle object. SetTabText() may be called to change the text of the tab on the zero-based *page* to what is passed into *text*. The corresponding ZafTitle object's text is modified to *text*.

TabWidth

```
int TabWidth(void);
virtual int SetTabWidth(int tabWidth);
```

TabWidth() specifies in pixel coordinates the width of the notebook tabs. By default this attribute is 0, which means that the tabs are variable-width, each tab calculating its width based on its text, but SetTabWidth() may be used to specify a fixed width for all the tabs.

ZafNotification

AddNotification	SubtractNotifications	UpdateObjects
ClearNotifications	Update	
NotifyCount	UpdateData	

Inheritance Root class

Declaration `#include <z_notify.hpp>`

Description 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.

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

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.

AddNotification

ClearNotifications

SubtractNotifica-tions

```
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. AddNotifiction() 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.

NotifyCount

```
int NotifyCount(void) const;
```

`NotifyCount()` returns the number of elements that have been added to the internal notification array.

Update

```
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.

UpdateData

```
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.

UpdateObjects

```
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.

ZafObjectPersistence

AddCompareFunction	GetClassName	Merge
AddObjectConstructor	GetCompareFunction	SetObjectConstructors
AddUserCallback	GetCompareFunctionName	SetUserCallbacks
AddUserObject	GetObjectConstructor	SetUserObjects
ClearCompareFunctions	GetUserCallback	SubtractCompareFunction
ClearObjectConstructors	GetUserCallbackName	SubtractObjectConstructor
ClearUserCallbacks	GetUserObject	SubtractUserCallback
ClearUserObjects	GetUserObjectName	SubtractUserObject

Inheritance ZafObjectPersistence : [ZafDataPersistence](#)

Declaration #include <z_win.hpp>

Description ZafObjectPersistence allows objects derived from [ZafWindowObject](#) to be written to, and/or read from a storage object—usually a persistent data file.

Using ZafObjectPersistence, a ZafWindowObject can be constructed directly from information stored in the data file, or stored there for later use. A table of persistent objects is maintained by this class and is used to reference each object’s Read() function during construction. A table of ZafWindowObject classes used in application is generated by Zinc Designer, or may be supplied by the programmer. Derived objects may also be supported using a generated table or a programmer-supplied table. See [AddObjectConstructor\(\)](#).

ZafObjectPersistence uses ZafFileSystem and ZafFile to maintain and manipulate one or more persistent object data files. See [ZafStorage](#) for more information.

Constructors **ZafObjectPersistence**(ZafFileSystem *fileSystem,
 DataConstructor *dataConstructor, ObjectConstructor
 *objectConstructor, UserFunction *userFunction =
 ZAF_NULLP(UserFunction), CompareFunction
 *compareFunction = ZAF_NULLP(CompareFunction),
 UserObject *userObject = ZAF_NULLP(UserObject));

The ZafObjectPersistence class constructor creates a ZafObjectPersistence object based on the open file *fileSystem*, the persistent data constructor table *dataConstructor*, and the persistent window object constructor table *objectConstructor*.

Optionally, the programmer may create tables of other application information to be passed into the constructor as follows: *userFunction* is a table of user functions, *compareFunction* is a table of compare functions (used when sort-

ing), and *userObject* is a table of user-defined objects. These tables must have a terminating entry in which the name field is null.

Following are the definitions of `ZafObjectPersistence::ObjectConstructor`, `ZafObjectPersistence::UserFunction`, `ZafObjectPersistence::CompareFunction`, and `ZafObjectPersistence::UserObject`. See [ZafDataPersistence](#) for definition of `ZafDataPersistence::AddDataConstructor`.

```
struct ObjectConstructor
{
    ZafClassID classID;
    ZafClassName className;
    ZafObjectConstructor constructor;
};
```

classID is the numeric class identification constant for the window object class, *className* is the string class identification constant for the window object class, and *constructor* is a pointer to the function used to construct an instance of the window object class. Internally, ZAF uses the *className* for its lookups. These identifiers are used by `ZafObjectPersistence` to locate the correct persistent constructor for a window object class.

The definition of `ZafObjectConstructor` is as follows:

```
typedef ZafElement *(*ZafObjectConstructor)(const ZafIChar *,
    ZafObjectPersistence &);
```

```
struct UserFunction
{
    ZafDataName name;
    ZafUserFunction function;
};
```

name is the unique string name of the user function and *function* is a pointer to the user function. These identifiers are used by `ZafObjectPersistence` to locate a user function in the user function table. The definitions of `ZafDataName` and `ZafUserFunction` are as follows:

```
typedef const ZafIChar *ZafDataName;
typedef ZafEventType (*ZafUserFunction)(ZafWindowObject *,
    ZafEventStruct &, ZafEventType);
```

The definition of `ZafObjectConstructor` is as follows:

```
struct CompareFunction
{
    ZafDataName name;
    ZafCompareFunction function;
```

```
};
```

name is the unique string name of the compare *function* and *function* is a pointer to the compare function. These identifiers are used by ZafObjectPersistence to locate a compare function in the compare function table. See [ZafList](#) for an explanation of [CompareFunction](#).

The definition of ZafObjectConstructor is as follows:

```
struct UserObject
{
    ZafDataName name;
    void *object;
};
```

name is the unique string name of the user object and *object* is a pointer to the user object. Since *object* is a void pointer, it may refer to any type. These identifiers are used by ZafObjectPersistence to locate a user object in the user object table.

```
ZafObjectPersistence(const ZafObjectPersistence &copy);
```

The copy constructor creates a new ZafObjectPersistence object and initializes its data from *copy*.

The following code snippet shows how to create a ZafObjectPersistence object and use it to load a window object from storage:

```
// Create a storage object, which opens the data file.
ZafStorage *storage = new ZafStorage("test.znc", ZAF_FILE_READ);

// Create a persistence object for loading the window object.
// Use the global data and object constructor tables defined
// either in the ZAF persistence library or in the file
// generated by Zinc Designer.
zafObjectPersistence = new ZafObjectPersistence(storage,
    zafDefaultDataConstructor, zafDefaultObjectConstructor);

// Load the window object from the data file.
ZafWindow *window = new ZafWindow("MyWindow",
    *zafObjectPersistence);
```

The following code snippet shows how to derive a class that stores more information than just what the base class stores:

```

// Define the static members of MyStringObject.
ZafClassID MyStringObject::classID = 3600;
ZafClassNameChar MyStringObject::className[] = "MyStringObject";

// Persistent constructor of MyStringObject.
// Call PushLevel() before reading the base class information.
MyStringObject::MyStringObject(const ZafIChar *name,
    ZafObjectPersistence &persist) : ZafString(name,
    persist.PushLevel(className, classID,
    ZAF_PERSIST_DIRECTORY))
{
    if (persist.Error() == ZAF_ERROR_NONE)
    {
        // Read the MyStringObject class information.
        ZafFile *file = persist.CurrentFile();
        *file >> MyStringObject::member;
    }

    // Call PopLevel() after the class information has been read.
    persist.PopLevel();

    // Save the error into the object if the data couldn't be read.
    if (persist.Error() != ZAF_ERROR_NONE)
        SetError(persist.Error());
}

// Define Read(), since ZafWindowObject must add it to the
// ZafObjectPersistence object constructor table.
ZafElement *MyStringObject::Read(const ZafIChar *name,
    ZafObjectPersistence &persist)
{
    return (new MyStringObject(name, persist));
}

// Write() stores the class information in the data file.
void MyStringObject::Write(ZafObjectPersistence &persist)
{
    // Write the base class information.
    // Call PushLevel() before writing the base class information.
    // AllocateFile() is called internally to prepare the file.
    ZafString::Write(persist.PushLevel(className, classID,
    ZAF_PERSIST_DIRECTORY));

    if (persist.Error() == ZAF_ERROR_NONE)
    {
        // Write the MyStringObject class information.
        ZafFile *file = persist.CurrentFile();
        *file << MyStringObject::member;
    }
}

```

```
// Call PopLevel() after writing the class information.
persist.PopLevel();

// Save the error into the object if the data couldn't be
// written.
if (persist.Error() != ZAF_ERROR_NONE)
    SetError(persist.Error());
}
```

Destructor

```
virtual ~ZafObjectPersistence(void);
```

This virtual destructor is used to free the memory associated with an instantiated ZafObjectPersistence object. Generally, the programmer will not directly destroy a ZafObjectPersistence object, since it is destroyed when the application is closed.

Members

AddCompare-Function

```
virtual CompareFunction *AddCompareFunction(ZafDataName
name, ZafCompareFunction function);
```

AddCompareFunction() adds a compare function to ZafObjectPersistence's table of compare functions. *name* is the unique string name of the compare function and *function* is a pointer to the compare function (available only at run-time). If an entry identified by *name* is already present in the table, its compare function is updated to *function*. A pointer to the CompareFunction element in the table is returned.

AddObject-Constructor

```
virtual ObjectConstructor
*AddObjectConstructor(ZafClassName className,
ZafClassID classID, ZafObjectConstructor constructor);
```

AddObjectConstructor() adds a window object class constructor to ZafObjectPersistence's table of constructors. *className* is the string class identification constant for the window object class, *classID* is the numeric class identification constant for the window object class, and *constructor* is a pointer to the function used to construct an instance of the window object class. If an entry with the same class identification constants is already present in the table, its constructor is updated to *constructor*. A pointer to the ObjectConstructor element in the table is returned.

AddUserCallback

```
virtual UserFunction *AddUserCallback(ZafDataName name,
ZafUserFunction function);
```

AddUserCallback() adds a user function to ZafObjectPersistence's table of user functions. *name* is the unique string name of the user function and *function* is a

pointer to the user function (available only at run-time). If an entry identified by *name* is already present in the table, its user function is updated to *function*. A pointer to the UserFunction element in the table is returned.

AddUserObject

```
virtual UserObject *AddUserObject(ZafDataName name, void
    *object);
```

AddUserObject() adds a user object to ZafObjectPersistence's table of user objects. *name* is the unique string name of the user object and *object* is a pointer to the user object. If an entry identified by *name* is already present in the table, its user object is updated to *object*. Since *object* is a void pointer, it may refer to any type. A pointer to the UserObject element in the table is returned.

ClearCompare- Functions

```
virtual void ClearCompareFunctions(void);
```

ClearCompareFunctions() clears the table of compare functions and destroys the memory associated with the table, so that no compare function is associated with the ZafObjectPersistence object.

ClearObject- Constructors

```
virtual void ClearObjectConstructors(void);
```

ClearObjectConstructors() clears the table of window object constructors and destroys the memory associated with the table, so that no window object constructor is associated with the ZafObjectPersistence object.

ClearUserCallbacks

```
virtual void ClearUserCallbacks(void);
```

ClearUserCallbacks() clears the table of user functions and destroys the memory associated with the table, so that no user function is associated with the ZafObjectPersistence object.

ClearUserObjects

```
virtual void ClearUserObjects(void);
```

ClearUserObjects() clears the table of user objects and destroys the memory associated with the table, so that no user object is associated with the ZafObjectPersistence object.

GetClassName

```
virtual ZafDataName GetClassName(ZafClassID classID);
```

GetClassName() searches the tables of constructors in ZafObjectPersistence, then in ZafDataPersistence for a class with a numeric class identification con-

stant of *classID*. If a matching entry is found, its class name is returned; otherwise, null is returned.

*GetCompare-
Function*

```
virtual ZafCompareFunction GetCompareFunction(ZafDataName  
name);
```

GetCompareFunction() finds and returns a compare function in ZafObjectPersistence's table of compare functions according to the unique string name of the compare function name. If no matching compare function is found, null is returned.

*GetCompare-
FunctionName*

```
virtual ZafDataName  
GetCompareFunctionName(ZafCompareFunction function);
```

GetCompareFunctionName() finds and returns the unique string name corresponding to the compare function *function*. If no matching compare function is found, null is returned.

*GetObject-
Constructor*

```
virtual ZafObjectConstructor  
GetObjectConstructor(ZafClassID classID, ZafClassName  
className = 0);
```

GetObjectConstructor() finds and returns a window object class constructor in ZafObjectPersistence's table of constructors according to the class identification constants. *classID* is the numeric class identification constant for the window object class and *className* is the string class identification constant for the window object class. If no matching class is found, null is returned.

GetUserCallback

```
virtual ZafUserFunction GetUserCallback(ZafDataName  
name);
```

GetUserCallback() finds and returns a user function in ZafObjectPersistence's table of user functions according to the unique string name of the user function name. If no matching user function is found, null is returned.

*GetUserCallback-
Name*

```
virtual ZafDataName GetUserCallbackName(ZafUserFunction  
function);
```

GetUserCallbackName() finds and returns the unique string name corresponding to the user function *function*. If no matching user function is found, null is returned.

-
- GetObject* `virtual void *GetObject(ZafDataName name);`
- GetObject() finds and returns a user object in ZafObjectPersistence's table of user objects according to the unique string name of the user object name. If no matching user object is found, null is returned.
-
- GetObject-Name* `virtual ZafDataName GetObjectName(void *userObject);`
- GetObjectName() finds and returns the unique string name corresponding to the user object *userObject*. If no matching user object is found, null is returned.
-
- Merge* `bool Merge(ZafObjectPersistence ©);`
- Merge() merges *copy* and all its data (including tables and file systems) into this ZafObjectPersistence object and returns true. The merged object's tables and file systems are removed from *copy* and added to this ZafObjectPersistence object, so *copy* is in effect empty after calling this function.
-
- SetCompare-Functions* `virtual bool SetCompareFunctions(CompareFunction *compareFunction);`
- SetCompareFunctions() clears ZafObjectPersistence's table of compare functions, then creates a new table of compare functions based on the pre-built *compareFunction* table. SetCompareFunctions() is an advanced routine, and should normally not be called by the programmer.
-
- SetObject-Constructors* `virtual bool SetObjectConstructors(ObjectConstructor *objectConstructor);`
- SetObjectConstructors() clears ZafObjectPersistence's table of constructors, then creates a new table of constructors based on the pre-built *objectConstructor* table. SetObjectConstructors() is an advanced routine, and should normally not be called by the programmer.
-
- SetUserCallbacks* `virtual bool SetUserCallbacks(UserFunction *userFunction);`
- SetUserCallbacks() clears ZafObjectPersistence's table of user functions, then creates a new table of user functions based on the pre-built *userFunction* table. SetUserCallbacks() is an advanced routine, and should normally not be called by the programmer.

- SetUserObjects* `virtual bool SetUserObjects(UserObject *userObject);`
- SetUserObjects() clears ZafObjectPersistence's table of user objects, then creates a new table of user objects based on the pre-built *userObject* table. SetUserObjects() is an advanced routine, and should normally not be called by the programmer.
- Subtract-CompareFunction* `virtual bool SubtractCompareFunction(ZafDataName name);`
- SubtractCompareFunction() removes a compare function from ZafObjectPersistence's table of compare functions. *name* is the unique string name of the compare function. If the entry was successfully removed, true is returned; otherwise, false is returned.
- SubtractObject-Constructor* `virtual bool SubtractObjectConstructor(ZafClassID
 classID, ZafClassName className = 0);`
- SubtractObjectConstructor() removes a window object class constructor from ZafObjectPersistence's table of constructors. *classID* is the numeric class identification constant for the window object class and *className* is the string class identification constant for the window object class. If the entry was successfully removed, true is returned; otherwise, false is returned.
- SubtractUser-Callback* `virtual bool SubtractUserCallback(ZafDataName name);`
- SubtractUserCallback() removes a user function from ZafObjectPersistence's table of user functions. *name* is the unique string name of the user function. If the entry was successfully removed, true is returned; otherwise, false is returned.
- SubtractUserObject* `virtual bool SubtractUserObject(ZafDataName name);`
- SubtractUserObject() removes a user object from ZafObjectPersistence's table of user objects. *name* is the unique string name of the user object. If the entry was successfully removed, true is returned; otherwise, false is returned.

ZafPaletteData

AddPalette	GetPalette	StaticData
Clear	MapTable	operator ()

Inheritance ZafPaletteData : [ZafData](#) : [ZafNotification](#), [ZafElement](#)

Declaration #include <z_pall.hpp>

Description ZafPaletteData is used to maintain and persist ZafPaletteMap tables, commonly used by ZAF objects derived from ZafWindowObject. See ZafWindowObject::UserPaletteData() for more information.

Constructors All ZafPaletteData constructors initialize the member variables associated with an instantiated ZafPaletteData object. The default values set by the ZafPaletteData and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafPaletteData

MapTable()	null
StaticData()	false

ZafElement

ClassID()	ID_ZAF_PALETTE_DATA
ClassName()	"ZafPaletteData"

```
ZafPaletteData(bool staticData = false);
```

```
ZafPaletteData(ZafPaletteMap *mapTable, bool staticData = false);
```

These constructors allocate a ZafPaletteData instance and initialize StaticData() to *staticData*. *mapTable* specifies the ZafPaletteMap table to use for MapTable(). See [ZafPaletteMap](#) for more information.

```
ZafPaletteData(const ZafPaletteData &copy);
```

The copy constructor creates a new ZafPaletteData object and initializes its data from *copy*.

```
ZafPaletteData(const ZafIChar *name, ZafDataPersistence
&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 code snippet shows how to create a `ZafPaletteData` object:

```
// Create a ZafPaletteData object.
ZafPaletteMap myPaletteMap[] =
{
    { ZAF_PM_BACKGROUND, ZAF_PM_ANY_STATE, { ZAF_LINE_NULL,
        ZAF_PTN_SOLID_FILL, ZAF_CLR_NULL, ZAF_CLR_WHITE,
        ZAF_MONO_NULL, ZAF_MONO_WHITE, ZAF_FNT_NULL, 0 } },
    { ZAF_PM_TEXT, ZAF_PM_ANY_STATE, { ZAF_LINE_NULL,
        ZAF_PTN_SOLID_FILL, ZAF_CLR_DARKGRAY, ZAF_CLR_WHITE,
        ZAF_MONO_BLACK, ZAF_MONO_WHITE, ZAF_FNT_DIALOG, 0 } },
    { ZAF_PM_ANY_TYPE, ZAF_PM_ANY_STATE, { ZAF_LINE_SOLID,
        ZAF_PTN_SOLID_FILL, ZAF_CLR_BLACK, ZAF_CLR_WHITE,
        ZAF_MONO_BLACK, ZAF_MONO_WHITE, ZAF_FNT_DIALOG, 0 } }
};
ZafPaletteData paletteData(myPaletteMap, false);
```

Destructor

```
virtual ~ZafPaletteData(void);
```

This virtual destructor is used to free the memory associated with an instantiated `ZafPaletteData` object.

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.

AddPalette

```
ZafError AddPalette(ZafPaletteType type, ZafPaletteState
state, ZafPaletteStruct &palette);
```

`AddPalette()` inserts an entry into the `MapTable()` and initializes the new entry's fields to *type*, *state*, and *palette*. `AddPalette()` usually returns `ZAF_ERROR_NONE`. See [ZafPaletteMap](#) for more information.

Clear

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

`Clear()` destroys the `MapTable()` if `StaticData()` is false. Regardless of `StaticData()`, `MapTable()` is set to null, effectively clearing the palette map data.

GetPalette

```
ZafPaletteStruct GetPalette(ZafPaletteType type,
    ZafPaletteState state);
```

GetPalette() finds an entry in the MapTable() corresponding to *type* and *state* and returns the ZafPaletteStruct portion of the entry. If no entry was found corresponding to *type* and *state*, an empty ZafPaletteStruct object is returned. See [ZafPaletteMap](#) for more information.

MapTable

```
const ZafPaletteMap *MapTable(void);
virtual ZafError SetMapTable(ZafPaletteMap *paletteMap);
```

MapTable() returns the palette map table associated with the ZafPaletteData object. This attribute defaults to null, but may be changed with SetMapTable(). SetMapTable() usually returns ZAF_ERROR_NONE.

StaticData

```
bool StaticData(void) const;
virtual bool SetStaticData(bool staticData);
```

If StaticData() is true, then MapTable() is considered static, and will not be destroyed by the ZafPaletteData class neither in the destructor, nor in SetMapTable(). This attribute defaults to false, but may be changed with SetStaticData().

operator ()

```
operator const ZafPaletteMap *();
```

This operator returns the MapTable() attribute.

ZafPaletteMap

	type	state	palette
Inheritance	Root struct		
Declaration	#include <z_pall.hpp>		
Description	The ZafPaletteMap struct is normally used in tables, and is used to map ZafPaletteStruct values to ZAF objects according to their current drawing type and state. See ZafWindowObject::UserPaletteData() for more information.		
Members	ZafPaletteType type ;		
<i>type</i>	type specifies the type of drawing being done for the object. Some examples are ZAF_PM_BACKGROUND (when drawing the background portion of the object) and ZAF_PM_TEXT (when drawing the textual information of the object). See ZafWindowObject::UserPaletteData() for more information.		
<i>state</i>	ZafPaletteState state ;		
	state specifies the object's current state during the draw operation. Some examples are ZAF_PM_CURRENT (when the object is current) and ZAF_PM_DISABLED (when the object is disabled). See ZafWindowObject::UserPaletteData() for more information.		
<i>palette</i>	ZafPaletteStruct palette ;		
	palette specifies the palette to be used for the draw operation, if type and state are appropriate for the draw operation. See ZafPaletteStruct for more information.		

ZafPaletteStruct

colorBackground	fillPattern	monoForeground
colorForeground	font	osPalette
lineStyle	monoBackground	

Inheritance	Root struct
Declaration	<code>#include <z_pal.hpp></code>
Description	ZafPaletteStruct is used to store palette information to be used during draw operations. A common use of ZafPaletteStruct is by ZafPaletteMap's palette member. A draw operation looks into a ZafPaletteStruct object to find the color or style to use. See ZafPaletteMap and ZafWindowObject::UserPaletteData() for more information.

Members

<i>colorBackground</i>	ZafLogicalColor colorBackground;
<i>colorForeground</i>	ZafLogicalColor colorForeground;
	colorBackground specifies the color to use when drawing a background on a color or grayscale screen, and colorForeground specifies the color to use when drawing a foreground on a color or grayscale screen. The possible values of these members are as follows:

ZafLogicalColor	Description
ZAF_CLR_BACKGROUND	Causes the color to be transparent, so that whatever is underneath the object shows through
ZAF_CLR_BLACK	Causes black to be used
ZAF_CLR_BLUE	Causes blue to be used
ZAF_CLR_BROWN	Causes brown to be used
ZAF_CLR_CYAN	Causes cyan to be used
ZAF_CLR_DARKGRAY	Causes dark gray to be used
ZAF_CLR_DEFAULT	Causes the object to use the environment's default color
ZAF_CLR_GREEN	Causes green to be used
ZAF_CLR_LIGHTBLUE	Causes light blue to be used
ZAF_CLR_LIGHTCYAN	Causes light cyan to be used
ZAF_CLR_LIGHTGRAY	Causes light gray to be used
ZAF_CLR_LIGHTGREEN	Causes light green to be used

ZafLogicalColor	Description
ZAF_CLR_LIGHTMAGENTA	Causes light magenta to be used
ZAF_CLR_LIGHTRED	Causes light red to be used
ZAF_CLR_MAGENTA	Causes magenta to be used
ZAF_CLR_NULL	Causes the object to use the class default color
ZAF_CLR_PARENT	Causes the object to use its parent's color
ZAF_CLR_RED	Causes red to be used
ZAF_CLR_WHITE	Causes white to be used
ZAF_CLR_YELLOW	Causes yellow to be used

lineStyle

ZafLogicalLineStyle **lineStyle**;

lineStyle specifies the style to use when drawing a line. The possible values of this member are as follows:

ZafLogicalLineStyle	Description
ZAF_LINE_DEFAULT	Causes the object to use the environment's default line style
ZAF_LINE_DOTTED	Causes a dotted line to be drawn, if supported on the environment
ZAF_LINE_NULL	Causes the object to use the class default line style
ZAF_LINE_PARENT	Causes the object to use it's parent's line style
ZAF_LINE_SOLID	Causes a solid line to be drawn

fillPattern

ZafLogicalFillPattern **fillPattern**;

fillPattern specifies the pattern to use when filling a shape. The possible values of this member are as follows:

ZafLogicalFillPattern	Description
ZAF_PTN_DEFAULT	Causes the object to use the environment's default fill pattern
ZAF_PTN_INTERLEAVE_FILL	Causes a 50% interleave fill pattern to be used
ZAF_PTN_NULL	Causes the object to use the class default fill pattern

ZafLogicalFillPattern	Description
ZAF_PTN_PARENT	Causes the object to use it's parent's fill pattern
ZAF_PTN_SOLID_FILL	Causes a solid fill pattern to be used

*font*ZafLogicalFont **font**;

font specifies the font to use when drawing text. The possible values of this member are as follows:

ZafLogicalFont	Description
ZAF_FNT_APPLICATION	Causes the application font to be used (as with multi-line text fields)
ZAF_FNT_DEFAULT	Causes the object to use the environment's default font
ZAF_FNT_DIALOG	Causes the dialog font to be used (as on dialog windows)
ZAF_FNT_FIXED	Causes the fixed-width font to be used (such as Courier)
ZAF_FNT_NULL	Causes the object to use the class default font
ZAF_FNT_PARENT	Causes the object to use it's parent's font
ZAF_FNT_SMALL	Causes the small font to be used (as with icons)
ZAF_FNT_SYSTEM	Causes the system font to be used (as on window title bars)

*monoBackground*ZafLogicalColor **monoBackground**;*monoForeground*ZafLogicalColor **monoForeground**;

monoBackground specifies the color to use when drawing a background on a black and white screen, and monoForeground specifies the color to use when drawing a foreground on a black and white screen. The possible values of these members are as follows:

ZafLogicalColor	Description
ZAF_MONO_BACKGROUND	Causes the color to be transparent, so that whatever is underneath the object shows through
ZAF_MONO_BLACK	Causes black to be used
ZAF_MONO_DEFAULT	Causes the object to use the environment's default color

ZafLogicalColor	Description
ZAF_MONO_DIM	Causes dim intensity to be used (text mode only)
ZAF_MONO_HIGH	Causes high intensity to be used (text mode only)
ZAF_MONO_NORMAL	Causes normal intensity to be used (text mode only)
ZAF_MONO_NULL	Causes the object to use the class default color
ZAF_MONO_PARENT	Causes the object to use it's parent's color
ZAF_MONO_WHITE	Causes white to be used

osPalette

OSPaletteID **osPalette**;

osPalette is used internally by ZAF, and specifies environment-specific palette information, if necessary. This is an advanced member, and the programmer should normally not access this member.

ZafPath

[Load](#)

Inheritance ZafPath : [ZafList](#)

Declaration `#include <z_file.hpp>`

Description A search path may be built by adding ZafPathElement objects to a ZafPath object. The global zafSearchPath is used internally by ZAF when trying to open a data file. The path elements are searched in the order that they were added to the ZafPath object. See [ZafPathElement](#) for more information.

Constructor **ZafPath**(const ZafIChar *pwd, bool getCwd);

This constructor is useful in straight-code situations, and creates a ZafPath object. If *pwd* is not null, it is a path specification to be added as a ZafPathElement to the new ZafPath object. If *getCwd* is true, the current directory is added as a ZafPathElement to the new ZafPath object as the first child of the ZafPath object (before *pwd* if it was added). Lastly, this constructor attempts to load the "ZAF_PATH" environment variable's contents (see [Load\(\)](#) for more information).

An example of how to create a search path follows:

```
// Create the global search path, including the current
// directory.
zafSearchPath = new ZafPath(ZAF_NULLP(ZafIChar), true);

// Add a data file subdirectory to the search path.
extern ZafIChar dataFilePath[];
zafSearchPath->Add(new ZafPathElement(dataFilePath));
```

Members bool **Load**(const ZafIChar *envname);

Load Load() finds the *envname* environment variable if it exists in the environment, and loads a new ZafPathElement object for each path specifier denoted by the environment variable into the ZafPath object. The ZafPath constructor automatically calls Load() to attempt to load the "ZAF_PATH" environment variable's contents.

path specifies the environment- specific search pathname to be associated with this ZafPathElement object. *length* specifies the maximum number of ZafI-Chars to be copied from *path*. If *length* is -1, the entire *path* buffer is copied into the ZafPathElement object.

ZafPopUpItem

Add	FocusObject	menu
Count	Get	Sort
Current	GetObject	Subtract
Destroy	Index	operator +
Event	ItemType	operator -
First	Last	

Inheritance	ZafPopUpItem : ZafButton : ZafWindowObject : ZafElement
Declaration	#include <z_popup.hpp>
Description	<p>ZafPopUpItem objects are the items in a menu. A hierarchical sub-menu may be created by simply adding ZafPopUpItem objects to a ZafPopUpItem object. A ZafPopUpItem object with its Selected() attribute set to true displays a checkmark. It is important to note that ZafPopUpItem objects may only be added to a ZafPullDownItem object, a ZafPopupMenu object, or another ZafPopUpItem object.</p>
Constructors	<p>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.</p>

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 [†]

Member Initializations

HzJustify()	ZAF_HZ_LEFT [†]
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.

Add

```
virtual ZafWindowObject *Add(ZafWindowObject *object,  
                              ZafWindowObject *position =  
                              ZAF_NULLP(ZafWindowObject));
```

operator +

```
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.

Count `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.

Current `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.

Destroy `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.

Event `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

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

The `First()` function returns the first `ZafPopUpItem` object in the sub-menu, if any. See `ZafList::First()` for more information.

```
FocusObject 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.

```
Get                ZafWindowObject *Get(int index);
```

Get() returns the ZafPopUpItem object at *position* index in the sub-menu, if there is one. *index* is zero-based, meaning the first ZafPopUpItem object in the sub-menu is at position 0. See ZafList::Get() for more information.

```
GetObject      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.

```
Index      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.

```
ItemType ZafPopUpItemType ItemType(void) const;
```

```
virtual ZafPopUpItemType SetItemType(ZafPopUpItemType
    itemType);
```

ItemType() specifies the menu item type of a ZafPopUpItem 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:

ZafPopUpItemType	Description
ZAF_NORMAL_ITEM	Creates a normal menu item

ZafPopUpItemType	Description
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

Last

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

Last() returns the last ZafPopUpItem object in the sub-menu, if any. See ZafList::Last() for more information.

menu `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.

Sort `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.

Subtract `virtual ZafWindowObject *Subtract(ZafWindowObject
 *object);`

operator - `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

```
Inheritance      ZafPopupMenu : ZafWindow : ((ZafWindowObject :
                    ZafElement), ZafList)
```

```
Declaration      #include <z_popup.hpp>
```

Description	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.

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 [†]
Modal()	false [†]
Moveable()	false [†]
NormalHotKeys()	true [†]
Sizeable()	false [†]
Temporary()	true [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	true [†]

Member Initializations

Disabled()	false [†]
Noncurrent()	false [†]
ParentPalette()	false [†]
RegionType()	ZAF_INSIDE_REGION [†]

ZafElement

ClassID()	ID_ZAF_POP_UP_MENU
ClassName()	"ZafPopupMenu"

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.
ZafPopupMenu *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\(\)](#).

ZafPositionStruct

Assign	Export	Position
column	HzShift	VtShift
ConvertCoordinates	Import	operator ==
coordinateType	line	operator !=

Inheritance Root structure

Declaration `#include <z_pos.hpp>`

Description ZafPositionStruct provides support for storing a position in 2-dimensional space. Various different coordinate systems may be specified for the position (see [ZafCoordinateType](#) for more information).

Members `void Assign(int column, int line, ZafCoordinateType type = ZAF_COORD_DEFAULT);`

Assign Assign() sets the position to *column* along the x-axis, and to *line* along the y-axis. If *type* is ZAF_COORD_DEFAULT, coordinateType is unchanged; otherwise, coordinateType is set to *type*.

column `int column, line;`
line *column* and *line* store the position's values along the x-axis and y-axis, respectively.

ConvertCoordinates `void ConvertCoordinates(ZafCoordinateType newType, ZafDisplay *display = ZAF_NULLP(ZafDisplay));`

ConvertCoordinates() converts the position to the *newType* coordinate system for use on the *display* device.

coordinateType `ZafCoordinateType coordinateType;`

coordinateType stores the coordinate system used by the position. See [ZafCoordinateType](#) for more information on coordinate systems in ZAF.

-
- Export** `void Export(POINT &point) const;`
`void Export(Point &point) const;`
- `Export()` exports the position to *point* in the native environment's position type. For example, in Microsoft Windows, the built-in type `POINT` is used, and on the Macintosh, the built-in type `Point` is used.
- HzShift** `void HzShift(int shift);`
- `HzShift()` modifies the position along the x-axis by the amount specified in *shift*. If *shift* is positive, the position moves in the positive direction. If *shift* is negative, the position moves in the negative direction.
- Import** `void Import(const POINT &point);`
`void Import(const Point &point);`
- `Import()` imports the position from *point* in the native environment's position type. For example, in Microsoft Windows, the built-in type `POINT` is used, and on the Macintosh, the built-in type `Point` is used.
- Position** `ZafPositionStruct Position(ZafCoordinateType type);`
- `Position()` returns the position converted to the *type* coordinate system. The original position is not modified.
- VtShift** `void VtShift(int shift);`
- `VtShift()` modifies the position along the y-axis by the amount specified in *shift*. If *shift* is positive, the position moves in the positive direction. If *shift* is negative, the position moves in the negative direction.
- operator ==** `bool operator==(const ZafPositionStruct &position) const;`
- This operator returns true if *position* is equal to this position object; otherwise it returns false. If the coordinate systems are different, a converted version of *position* is used in the comparison.
- operator !=** `bool operator!=(const ZafPositionStruct &position) const;`
- This operator returns true if *position* is not equal to this position object; otherwise it returns false. If the coordinate systems are different, a converted version of *position* is used in the comparison.

ZafPrintDialog

	AllowModifyCollate	AllowModifyCopies	AllowModifyRange
Inheritance	ZafPrintDialog : ZafDialogWindow		
Declaration	#include <z_print.hpp>		
Description	ZafPrintDialog presents the end user with a dialog with which the ZafPrintJobStruct members (provided to the constructor) may be modified. The available SetAllowModify*() accessor functions change the appearance of the ZafPrintDialog window where appropriate. ZafPrintDialog invokes a native dialog on environments that have native printing support and a Zinc supplied dialog on others.		
Constructor	The ZafPrintDialog constructor initializes the member variables associated with an instantiated ZafPrintDialog object. The default values set by the ZafPrintDialog constructor follow.		

Member Initializations

ZafPrintDialog

AllowModifyCollate()	platform-specific
AllowModifyCopies()	true
AllowModifyRange()	true

ZafPrintDialog(int left, int top, ZafPrinter *printer, ZafPrintJobStruct *printJob);

The constructor is useful in straight-code situations. *left* and *top* specify the position of the dialog on the screen, but may be ignored by some environments that position the dialog themselves. *printer* is a pointer to the ZafPrinter object corresponding to the print dialog. *printJob* is a pointer to the print job structure to be modified if the end user successfully modifies the information presented in the print dialog. If the end user cancels the dialog, *printJob* is left untouched. This constructor (followed by the Control() method) is usually not called by the programmer, since it is called by ZafPrinter::JobSetup(). The following code snippet shows an example of using this class:

```
ZafPrinter printer;
ZafPrintJobStruct printJob;
printJob.collate = false;
printJob.copies = 1;
```

```

printJob.startPage = printJob.endPage = 1;
printJob.minPage = printJob.maxPage = 1;

// Do the same thing printer.JobSetup(printJob) would do.
ZafPrintDialog printDialog(0, 0, printer, &printJob);
return ((printDialog.Control() == S_DLG_OK) ? true : false);

```

Members

AllowModifyCollate

```

bool AllowModifyCollate(void) const;
bool SetAllowModifyCollate(bool allowModifyCollate);

```

If `AllowModifyCollate()` is true, the end user may modify the *collate* member of the `ZafPrintJobStruct` passed into the constructor. `SetAllowModifyCollate()` may be called to allow or disallow end user modification of *collate* in the print dialog.

AllowModifyCopies

```

bool AllowModifyCopies(void) const;
bool SetAllowModifyCopies(bool allowModifyCopies);

```

If `AllowModifyCopies()` is true, the end user may modify the *copies* member of the `ZafPrintJobStruct` passed into the constructor. `SetAllowModifyCopies()` may be called to allow or disallow end user modification of *copies* in the print dialog.

AllowModifyRange

```

bool AllowModifyRange(void) const;
bool SetAllowModifyRange(bool allowModifyRange);

```

If `AllowModifyRange()` is true, the end user may modify the *startPage* and *endPage* members of the `ZafPrintJobStruct` passed into the constructor. `SetAllowModifyRange()` may be called to allow or disallow end user modification of *startPage* and *endPage* in the print dialog.

ZafPrinter

AbortJob	Margins	PrinterType
BeginJob	PageHeight	PrintSetup
BeginPage	PageWidth	TextBlock
EndJob	PaperHeight	TextLine
EndPage	PaperOrientation	
JobSetup	PaperWidth	

Inheritance ZafPrinter : [ZafDisplay](#)

Declaration #include <z_print.hpp>

Description ZafPrinter defines the basic functionality necessary to output to a printer. ZafPrinter provides the same set of graphic display primitives as those used for displaying on the screen, plus additional printer-specific primitives and printer interface methods such as [BeginJob\(\)](#) and [EndJob\(\)](#). See the base class [ZafDisplay](#) for complete descriptions of the inherited “display” functions provided by ZafPrinter.

Constructor The ZafPrinter constructor initializes the member variables associated with an instantiated ZafPrinter object. The default values set by the ZafPrinter constructor and its base class constructor follow, if they differ from those set by the base class constructor.

Member Initializations

ZafPrinter

<code>colorTable[]</code>	display-dependent
<code>fontTable[]</code>	display-dependent
<code>lineTable[]</code>	display-dependent
<code>Margins()</code>	(0.5", 0.5", 0.5", 0.5")
<code>modeTable[]</code>	display-dependent
<code>PageHeight()</code>	<code>PaperHeight()</code> - top margin - bottom margin
<code>PageWidth()</code>	<code>PaperWidth()</code> - left margin - right margin
<code>PaperHeight()</code>	11" (North America) or A4
<code>PaperOrientation()</code>	ZAF_PORTRAIT_ORIENTATION
<code>PaperWidth()</code>	8.5" (North America)_or A4
<code>patternTable[]</code>	display-dependent
<code>PrinterType()</code>	printer-dependent

Member Initializations

ZafDisplay

cellHeight	font-dependent
cellWidth	font-dependent
columns	printer-dependent
coordinateType	printer-dependent
DisplayType()	"ZafPrinter"
lines	printer-dependent
pixelsPerInchX	printer-dependent
pixelsPerInchY	printer-dependent
postSpace	environment-specific
preSpace	environment-specific

ZafPrinter(void);

The constructor is useful in straight-code situations. This constructor calls the base ZafDisplay constructor. The following code snippet shows an example of using this class:

```
// Set up the printer (often in a "Page setup" menu item).
ZafPrinter printer;
printer.PrintSetup();

// Set up the print job.
ZafPrintJobStruct printJob;
printJob.collate = false;
printJob.copies = 1;
printJob.startPage = printJob.endPage = 1;
printJob.minPage = printJob.maxPage = 1;
if (JobSetup(printJob))
{
    BeginJob();
    BeginPage();
    printer.Text(0, 0, printer.PageWidth(), printer.PageHeight(),
        "Hello, Printer!", -1, ZAF_HZ_CENTER, ZAF_VT_CENTER);
    EndPage();
    EndJob();
}
```

Destructor

```
virtual ~ZafPrinter(void);
```

The destructor is used to free the memory associated with a ZafPrinter object. This destructor calls the base ZafDisplay destructor.

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.

Many of the methods described here return an error code of one of the following codes:

ZafError	Description
ZAF_ERROR_NOT_RESPONDING	The printer appears to not be responding, such as when its power is off, when a cable is disconnected, or the network connection is down.
ZAF_ERROR_OUT_OF_PAPER	The printer is out of paper.
ZAF_ERROR_LOW_MEMORY	Not enough memory is available to complete the operation.
ZAF_ERROR_OUTPUT_FAILED	The print output did not complete successfully.
ZAF_ERROR_UNKNOWN	Some other error exists which prevents the completion of the printing operation.

AbortJob

ZafError **AbortJob**(void);

AbortJob() tells the printer to ignore any data currently in its buffer and end the print job. It is not necessary to call EndJob() if AbortJob() is called. An error code is returned if the job could not be aborted.

BeginJob

ZafError **BeginJob**(ZafIChar *jobName =
ZAF_NULLP(ZafIChar));

BeginJob() must be called to begin a print job, and possibly communicates with the native printer driver. If the native printer driver can use a job name for the native queue, *jobName* is used. An error code is returned if the job could not be started. If an error code is returned, the error condition must be corrected and BeginJob() must be called again.

BeginPage

ZafError **BeginPage**(void);

BeginPage() must be called to begin a print page, and performs any necessary internal setup for beginning a print page. An error code is returned if the page

could not be printed due to printer error conditions. If an error code is returned, the error condition must be corrected and `BeginPage()` must be called again.

EndJob `ZafError EndJob(void);`

`EndJob()` must be called to end a print job, and does any necessary cleanup following printing. An error code is returned if the job could not be completed.

EndPage `ZafError EndPage(void);`

`EndPage()` must be called to end a print page, and performs any necessary internal cleanup after printing a page, including sending a form feed command. An error code is returned if the page could not be printed due to printer error conditions. If an error code is returned the page must be printed again to ensure intended output.

JobSetup `virtual bool JobSetup(ZafPrintJobStruct &printJob);`

`JobSetup()` presents the end user with a `ZafPrintDialog` window, allowing the end user to modify the data in *printJob* using a native or Zinc-supplied dialog. (When using a native dialog only those fields available on that platform may be changed). A `ZafPrintDialog` should be created by the programmer when fields other than the platform defaults must be allowed or disallowed (see [ZafPrintDialog](#)).

`JobSetup()` returns true if the end user chose to continue printing based on the values in *printJob*. Otherwise, *printJob* was not modified, and the end user chose not to continue printing. See [ZafPrintJobStruct](#) for more information.

`JobSetup()` should generally be called before beginning a print job, but should not be called between calls to `BeginJob()` and `EndJob()`. If the end user cancels the dialog (indicated by a return code of false), the print job data was not changed, and the job should not be printed. Otherwise, the print job may be started by calling `BeginJob()`.

Margins

```
virtual void Margins(ZafCoordinate &leftMargin,  
    ZafCoordinate &topMargin, ZafCoordinate &rightMargin,  
    ZafCoordinate &bottomMargin) const;  
virtual void SetMargins(ZafCoordinate leftMargin,  
    ZafCoordinate topMargin, ZafCoordinate rightMargin,  
    ZafCoordinate bottomMargin);
```

Margins() specifies the logical print page size (printable area) and the size of the unprintable border that remains blank. When the margins are modified, PageHeight() and PageWidth() are automatically modified to reflect the change. The default values of this attribute are the equivalent of (0.5", 0.5", 0.5", 0.5") using the printer's CoordinateType(), but the user may call SetMargins() to make changes.

PageHeight

```
virtual ZafCoordinate PageHeight(void) const;  
virtual ZafCoordinate SetPageHeight(ZafCoordinate  
    paperHeight);
```

PageWidth

```
virtual ZafCoordinate PageWidth(void) const;  
virtual ZafCoordinate SetPageWidth(ZafCoordinate  
    paperWidth);
```

PageHeight() and PageWidth() specify the dimensions of the logical print page using the printer's [CoordinateType\(\)](#). The logical print page is the printable area within the margins. If SetPageHeight() or SetPageWidth() are called, the print margins will be adjusted to correlate to the logical print page. If the specified size is too large for the paper size, the logical size will be reduced and the corresponding margin(s) set to zero. Left and top margins are not adjusted by SetPage*().

PaperHeight

```
virtual ZafCoordinate PaperHeight(void) const;  
virtual ZafCoordinate SetPaperHeight(ZafCoordinate  
    paperHeight);
```

PaperWidth

```
virtual ZafCoordinate PaperWidth(void) const;  
virtual ZafCoordinate SetPaperWidth(ZafCoordinate  
    paperWidth);
```

Initialized to letter size (North America) or A4 in the constructor, PaperHeight() and PaperWidth() specify the size of the physical paper being used. The paper size may include unprintable areas such as printer "dead zones" or margins. This information may be used to communicate the paper size to the printer. This member is normally set by the native print job setup dialog invoked using PrintSetup().

PaperOrientation virtual ZafPaperOrientation **PaperOrientation**(void) const;
 virtual ZafPaperOrientation
 SetPaperOrientation(ZafPaperOrientation
 paperOrientation);

Initialized to ZAF_PORTRAIT_ORIENTATION in the constructor, PaperOrientation() specifies the orientation of the physical paper. PaperOrientation() may either be ZAF_PORTRAIT_ORIENTATION or ZAF_LANDSCAPE_ORIENTATION. This member is normally set by the native print job setup dialog invoked using PrintSetup().

PrinterType virtual ZafPrinterType **PrinterType**(void) const;
 virtual ZafPrinterType **SetPrinterType**(ZafPrinterType
 printerType);

PrinterType() specifies the type of printer being used, such as PostScript (level 1 or 2), PCL, or native printer driver. The printer types currently supported by ZAF are as follows:

Printer Type	Description
ZAF_NATIVE_PRINTER	A native printer driver is being used. This is the default for environments with native support for printers.
ZAF_POSTSCRIPT_1_PRINTER	The printer is a PostScript (Level 1) printer.
ZAF_POSTSCRIPT_2_PRINTER	The printer is a PostScript (Level 2) printer. This is the default for environments without native support for printers (e.g. Unix, MS-DOS.)
ZAF_PCL_PRINTER	The printer supports HP's PCL (not yet supported by ZAF).

PrintSetup virtual bool **PrintSetup**(void);

PrintSetup() presents the end user with a print setup dialog window (commonly called page setup on environments that have native printer driver support), allowing the end user to modify the data members of the printer object, plus others as necessary (resolution, for example) and to communicate with the native printer driver on platforms that provide them. PrintSetup() should not be called between calls to BeginJob() and EndJob(), but the printer object's data members should be initialized by the programmer in case the end user doesn't request the print setup dialog.

PrintSetup() returns true if the end user successfully modified the information in the print setup dialog, and the printer's data members were modified accordingly; otherwise, the end user canceled the dialog and the printer's data members were not modified.

TextBlock

```
virtual ZafCoordinate TextBlock(ZafCoordinate left,  
    ZafCoordinate top, ZafCoordinate right, ZafCoordinate  
    bottom, int &charactersPrinted, const ZafIChar *text,  
    int length = -1, ZafHzJustify hzJustify = ZAF_HZ_LEFT,  
    ZafVtJustify vtJustify = ZAF_VT_TOP, bool fill =  
    false);
```

TextBlock() prints *text* in the region specified by *left*, *top*, *right*, and *bottom*, utilizing word wrapping, and stopping when it can no longer fit any more text within the region provided. TextBlock() returns the vertical position it would expect the next line of text to use. This is useful when positioning sequential rows of data on a report. TextBlock() returns in *charactersPrinted* the number of characters printed from the supplied text. This is useful when the programmer does not know whether the text to be printed will fit on a page since TextBlock() will allow a page in progress to be ejected (via a call to EndPage()) and the remaining text to be printed on subsequent pages. If *length* is -1, TextBlock() will attempt to print all the text; otherwise, only up to *length* characters will be printed. The text will be justified according to *hzJustify* and *vtJustify*, and the region will be filled with the background color if *fill* is true. See ZafDisplay::Text() for more information on the parameters to TextBlock().

TextLine

```
virtual ZafCoordinate TextLine(ZafCoordinate left,  
    ZafCoordinate top, ZafCoordinate right, const ZafIChar  
    *text, int length = -1, ZafHzJustify hzJustify =  
    ZAF_HZ_LEFT, bool fill = false);
```

TextLine() prints a single line of *text* at the position specified by *left*, *top*, and *right*. TextLine() returns the vertical position it would expect the next line of text to use. This is useful when positioning sequential rows of data on a report. If *length* is -1, TextLine() will attempt to print all the text on the line; otherwise, only up to *length* characters will be printed on the line. The text will be justified according to *hzJustify*, and the line will be filled with the background color if *fill* is true. See ZafDisplay::Text() for more information on the parameters to TextLine().

ZafPrintJobStruct

Inheritance Root structure

Declaration `#include <z_print.hpp>`

Description ZafPrintJobStruct contains information about a print job. Since this structure has no constructor, the programmer is responsible for initializing each of the members. ZafPrintJobStruct is used by [ZafPrintDialog](#).

The ZafPrintJobStruct structure is defined as follows:

```
struct ZAF_EXPORT ZafPrintJobStruct
{
    // --- Data Memebers ---
    bool collate;
    int copies, startPage, endPage, minPage, maxPage;
};
```

collate If *collate* is true, the application is responsible for collating the print job, meaning that all pages are printed before printing another copy. For example, two copies of a 2-page document would be printed as follows: page 1, page 2, page 1, page 2. If the printer handles collation, the application does not need to, so *collate* is false in this case. *collate* defaults to false.

copies *copies* specifies the number of copies of the document that should be printed. If the printer can print multiple copies the application does not need to, so *copies* is 1 in this case. *copies* defaults to 1.

startPage
endPage
minPage
maxPage

startPage and *endPage* specify the first and last pages of the document to be printed; and *minPage* and *maxPage* specify the first and last pages of the document. (Some native print drivers require *minPage* and *maxPage* for internal validation.) *startPage* should not be less than *minPage* and *endPage* should not be greater than *maxPage*.

For example, a document whose first page is 1 and whose last page is 10 would print the last page if *startPage* and *endPage* were both 10. On the other hand, a document that represents a chapter of a book, whose first page is 6 and whose last page is 15 would print the last page of the document (or chapter) if *startPage* and *endPage* were both 15.

ZafProgressBar

Current	Maximum	ProgressType
Decrement	Minimum	Step
Delta	ProgressData	TextStyle
Increment	ProgressStyle	

Inheritance ZafProgressBar : ZafWindowObject : ZafElement

Declaration #include <z_prgrss.hpp>

Description 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.

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 [†]
Focus()	false [†]
HelpContext()	null [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
Noncurrent()	true [†]
OSDraw()	false
ParentDrawBorder()	false [†]
ParentDrawFocus()	false [†]

Member Initializations

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.

Current

```
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()`.

Decrement

```
long Decrement(long value);
```

`Decrement()` decrements the progress bar by the amount `Delta()` by passing *value* to `ProgressData()->Decrement()`. The current value of the progress bar is returned.

Delta

```
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()`.

Increment

```
long Increment(long value);
```

`Increment()` increments the progress bar by the amount `Delta()` by passing *value* to `ProgressData()->Increment()`. The current value of the progress bar is returned.

Maximum

```
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()`.

Minimum

```
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()`.

ProgressData

```
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](#). SetProgressData() will delete the previous ProgressData() object if it is Destroyable() and no other object uses it.

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.

ProgressStyle

```
ZafProgressBar ProgressStyle(void) const;  
ZafProgressBar SetProgressStyle(ZafProgressBar  
    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
ZAF_PROGRESS_NATIVE	Appears the same as a progress bar on the native environment
ZAF_PROGRESS_INDENTED	Appears bevelled into the plane of its parent
ZAF_PROGRESS_FLAT	Appears at the same level as the plane of its parent
ZAF_PROGRESS_RAISED	Appears raised above the plane of its parent

ProgressType

```
ZafProgressBar ProgressType(void) const;  
ZafProgressBar SetProgressType(ZafProgressBar  
    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.

Step

```
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.

TextStyle

```
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	HzJustify	TransparentBackground
HotKeyChar	SetHotKey	VtJustify
HotKeyIndex	StringData	

Inheritance ZafPrompt : ZafWindowObject : ZafElement

Declaration #include <z_prompt.hpp>

Description 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.

Note that ZafPrompt utilizes the Bordered() attribute differently than other window objects. With ZafPrompt, the Bordered() attribute means the prompt is to be aligned as if it were a Bordered() object, so that the prompt's text is aligned with an adjoining Bordered() field's text. The prompt object, however, does not display a border. For additional information see ZafWindowObject::Bordered().

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 [†]

Member Initializations

Focus()	false [†]
HelpContext()	null [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
Noncurrent()	true [†]
RegionType()	ZAF_INSIDE_REGION [†]
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 [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 informa-

tion. 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 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.

AutoSize

```
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.

HotKeyChar
HotKeyIndex
SetHotKey

```
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));
```

HzJustify

```
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.

StringData

```
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](#). SetStringData() will delete the previous StringData() object if it is Destroyable() and no other object uses it.

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.

*Transparent-
Background*

```
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.

VtJustify

```
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	FocusObject	Sort
Count	Get	Subtract
Current	GetObject	operator +
Destroy	Index	operator -
Event	Last	
First	menu	

Inheritance	ZafPullDownItem : ZafButton : ZafWindowObject : ZafElement
Declaration	#include <z_plldn.hpp>
Description	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.
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

menu	ZafPopUpMenu(0, 0)
menu.Destroyable()	false
menu.parent	this
menu.StringID()	"menu"
menu.Temporary()	true

ZafButton

AllowDefault()	false [†]
AllowToggling()	false [†]
AutoRepeatSelection()	false [†]
AutoSize()	false [†]
BitmapData()	ZAF_NULLP(ZafBitmapData) [†]
ButtonType()	ZAF_FLAT_BUTTON [†]
Depressed()	false [†]
Depth()	0 [†]

Member Initializations

HzJustify()	ZAF_HZ_CENTER [†]
SelectOnDoubleClick()	false [†]
SelectOnDownClick()	false [†]
SendMessageText()	null [†]
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 data-

base 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 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

Add

```
virtual ZafWindowObject *Add(ZafWindowObject *object,  
                               ZafWindowObject *position =  
                               ZAF_NULLP(ZafWindowObject));
```

operator +

```
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.

Count int **Count**(void) const;

Count() returns the number of ZafPopUpItem objects in the menu. See ZafList::Count() for more information.

Current 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.

Destroy 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.

Event 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
S_ADD_OBJECT	Causes event.windowObject to be added to the menu
S_SUBTRACT_OBJECT	Causes event.windowObject to be subtracted from the menu

First ZafWindowObject ***First**(void) const;

First() returns the first ZafPopUpItem object in the menu, if any. See ZafList::First() for more information.

FocusObject `virtual ZafWindowObject *FocusObject(void) const;`

FocusObject() returns the ZafPopUpItem 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.

Get `ZafWindowObject *Get(int index);`

Get() returns the ZafPopUpItem object at position *index* in the menu, if there is one. The index parameter is zero-based, meaning the first ZafPopUpItem object is at position 0. See ZafList::Get() for more information.

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

GetObject() returns the ZafPopUpItem object with either the numberID of *numberID*, or the stringID of *stringID*, if it is found. See ZafWindowObject::GetObject() for more information.

Index `int Index(ZafWindowObject const *element);`

Index() returns the zero-based index of the ZafPopUpItem object *element* in the menu. If *element* is not found in the menu, Index() returns -1. See ZafList::Index() for more information.

Last `ZafWindowObject *Last(void) const;`

Last() returns the last ZafPopUpItem object in the menu, if there is one. See ZafList::Last() for more information.

menu `ZafPopupMenu menu;`

The menu member, used internally by the ZAF libraries to maintain the ZafPopUpItem objects added to the ZafPullDownItem object, should normally not be accessed by the programmer. See ZafPopupMenu for more information.

Sort `virtual void Sort(void);`

`Sort()` causes the `ZafPopUpItem` objects in the menu to be sorted by the function returned by `menu.CompareFunction()`. See `ZafList::CompareFunction()` for more information.

Subtract `virtual ZafWindowObject *Subtract(ZafWindowObject
 *object);`

operator - `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

```
Inheritance      ZafPullDownMenu : ZafWindow : ((ZafWindowObject :
                    ZafElement), ZafList)
```

```
Declaration      #include <z_plldn.hpp>
```

Description	The ZafPullDownMenu object is a menu bar object that supports multiple-leveled 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.

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 [†]
Modal()	false [†]
Moveable()	false [†]
NormalHotKeys()	true [†]
SelectionType()	ZAF_SINGLE_SELECTION [†]
Sizeable()	false [†]
Temporary()	false [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	true [†]
Disabled()	false [†]

Member Initializations

<code>Noncurrent()</code>	<code>false</code> [†]
<code>ParentPalette()</code>	<code>false</code> [†]
<code>RegionType()</code>	<code>ZAF_AVAILABLE_REGION</code> [†]
<code>SupportObject()</code>	<code>true</code> [†]

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_PULL_DOWN_MENU</code>
<code>ClassName()</code>	<code>"ZafPullDownMenu"</code>
<code>SetNumberID()</code>	<code>ZAF_NUMID_PULL_DOWN_MENU</code>
<code>SetStringID()</code>	<code>"ZAF_NUMID_PULL_DOWN_MENU"</code>

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().

ZafQueueBlock

Inheritance `ZafQueueBlock` : [ZafListBlock](#) : [ZafList](#)

Declaration `#include <z_evtmgr.hpp>`

Description `ZafQueueBlock` is a container object used by `ZafEventManager` to maintain a queue block of `ZafEventStruct` objects. `ZafQueueElement` is used to contain the actual `ZafEventStruct` objects. Only `ZafQueueElement` objects may be added to a `ZafQueueBlock`, and it will generally not be used by the programmer, as `ZafEventManager` maintains a `ZafQueueBlock` internally.

Constructors All `ZafQueueBlock` constructors initialize the member variables associated with an instantiated `ZafQueueBlock` object.

```
ZafQueueBlock(int noOfElements);
```

This constructor creates a `ZafQueueBlock` of *noOfElements* `ZafQueueElements`. It is used only by `ZafEventManager`.

Destructor `virtual ~ZafQueueBlock(void);`

The destructor is used to free the memory associated with a `ZafQueueBlock` object, including all the `ZafQueueElement` objects associated with it. It chains to the `ZafListBlock` and `ZafList` destructors.

ZafQueueElement

[event](#)

Inheritance	ZafQueueElement : ZafElement
Declaration	<code>#include <z_evtmgr.hpp></code>
Description	ZafQueueElement is a container object used by ZafQueueBlock and ZafEventManager to maintain a queue of ZafEventStruct objects. A ZafQueueElement may only be added to a ZafQueueBlock, and generally will not be used by the programmer, as ZafEventManager maintains a ZafQueueBlock internally.
Constructors	<p>All ZafQueueElement constructors initialize the member variables associated with an instantiated ZafQueueElement object.</p> <pre>ZafQueueElement(void);</pre> <p>This constructor creates a ZafQueueElement, and is used only by ZafQueueBlock.</p>
Destructor	<pre>virtual ~ZafQueueElement(void);</pre> <p>The destructor is used to free the memory associated with a ZafQueueElement object. It chains to the ZafElement destructor.</p>
Members	<pre>ZafEventStruct event;</pre> <p><i>event</i></p> <p>The event member stores the actual event associated with the ZafQueueElement object. It is initialized by ZafEventManager::Get(), and retrieved by ZafEventManager::Put().</p>

ZafReal

Event	SetReal
RealData	Value

Inheritance ZafReal : ZafString : ZafWindowObject : ZafElement

Declaration #include <z_real1.hpp>

Description 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. See ZafString::AllowInvalid() and ZafString::ReportInvalid() for information on these attributes and how they affect validation for this class.

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

Formats ZafString and derived classes parse input and display output based on default or programmer-defined input and output formats. Programmers may use the SetInputFormat() and SetOutputFormat() families of functions to change the default format. These functions are documented in the ZafString reference.

Each class derived from ZafFormatData handles a different set of formatting arguments, in addition to those supported by its base classes. ZafRealData (and therefore ZafReal) handles the following arguments in addition to those used by ZafInteger and ZafString:

Format Argument	Substitution
%[ww.dd]A	Optional arguments in “[]”: <i>ww</i> indicates the minimum width of the output (left-padded with spaces if necessary). <i>dd</i> indicates the mandatory number of digits after the decimal (right-padded with zeros if necessary). <i>A</i> is a proxy for any of the arguments below. For example, a complete format argument might be: “%10.2f”
%f, %F	Signed floating point value
%e	Signed floating point value with lower-case exponent
%E	Signed floating point value with upper-case exponent
%g	Signed floating point value using the most compact of either %f or %e format
%G	Signed floating point value using the most compact of either %F or %E format

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 the ZafReal object is 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 that initially appears 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 ZafRealData object, rather than having the ZafReal class create and maintain the data object automatically. For more information on using ZafRealData objects, see [ZafRealData](#). *left*, *top*, and *width* 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*Event*

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
N_RESET_I18N	causes the object to redisplay its data according to the new internationalization values
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

RealData

```
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](#). SetRealData() will delete the previous RealData() object if it is Destroyable() and no other object uses it.

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 [Constructors](#) code snippet for an example using ZafRealData objects with ZafReal.

SetReal

```
virtual ZafError SetReal(double value);
```

SetReal() sets the value of the ZafRealData associated with this ZafReal from *value*.

Value

```
double Value(void);
```

Value() returns the value of the ZafRealData associated with this ZafReal as a double.

ZafRealData

Clear	operator ++	operator *=
double	operator --	operator /=
FormattedText	operator =	operator %=
SetReal	operator +=	
Value	operator -=	

Inheritance

ZafRealData : ZafFormatData : ZafData : ZafElement, ZafNotification

Declaration

#include <z_real.hpp>

Description

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.

Constructors

ZafRealData constructors allocate space to hold the instance data, and initialize the member variables associated with a new ZafRealData object.

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

Member Initializations

ZafRealData	
Value()	(varies with constructor)
ZafElement	
ClassID()	ID_ZAF_REAL_DATA
ClassName()	"ZafRealData"

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 real1(value);
ZafRealData copyReal = real1;
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

Clear

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

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

double operator **double**();
See [Value\(\)](#).

FormattedText 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* characters. 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, 256);
printf("real - %s\n", buffer);
myreal.FormattedText(buffer, 256, "%+8.3g");
printf("real - %s\n", buffer);

=====
real - 1234.5600
real - +1234.560
```

SetReal 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.

Value double **Value**(void) const;
double 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.

operator ++ ZafRealData **operator++**(void);
ZafRealData **operator++**(int);

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

operator -- ZafRealData **operator--**(void);
ZafRealData **operator--**(int);

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

operator = ZafRealData &**operator**=(double value);

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

operator += ZafRealData &**operator**+=(double value);
operator -= ZafRealData &**operator**-(double value);

These operators increments or decrement the ZafRealData object's value by the input *value*.

*operator *=* 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.

operator /= 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.

operator %= 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.

ZafRegionStruct

Assign	ImportPixel	Width
bottom	ImportPoint	operator ==
ConvertCoordinates	left	operator !=
coordinateType	Overlap	operator ++
Encompassed	Region	operator --
ExportPixel	right	operator +=
ExportPoint	top	operator -=
Height	Touching	
HzShift	VtShift	

Inheritance Root structure

Declaration `#include <z_region.hpp>`

Description ZafRegionStruct provides support for storing a two-dimensional rectangular region. Various different coordinate systems may be specified for the region (see [ZafCoordinateType](#) for more information).

Members

Assign `void Assign(int left, int top, int width, int height, ZafCoordinateType type = ZAF_COORD_DEFAULT);`

Assign() sets the region's boundaries to *left*, *top*, *width*, and *height*. If *type* is ZAF_COORD_DEFAULT, coordinateType is unchanged; otherwise, coordinateType is set to *type*.

ConvertCoordinates `void ConvertCoordinates(ZafCoordinateType newType, ZafDisplay *display = ZAF_NULLP(ZafDisplay));`

ConvertCoordinates() converts the region to the *newType* coordinate system for use on the *display* device.

coordinateType `ZafCoordinateType coordinateType;`

coordinateType stores the coordinate system used by the region. See [ZafCoordinateType](#) for more information on coordinate systems in ZAF.

Encompassed `bool Encompassed(const ZafRegionStruct ®ion) const;`

Encompassed() returns true if *region* is encompassed by this region; otherwise it returns false. A region is encompassed within another if each of the four

boundaries is either inside or equal to the encompassing region's boundaries. If the coordinate systems are different, a converted version of *region* is used in the comparison.

```

ExportPixel    void ExportPixel(OSRegion &rect) const;
ExportPoint    void ExportPoint(OSRegion &rect) const;
ImportPixel    void ImportPixel(const OSRegion &rect);
ImportPoint    void ImportPoint(const OSRegion &rect);

```

On environments whose coordinate system is based on infinitely small points (rather than pixels, which have finite size on the screen) such as Macintosh and Microsoft Windows, conversion is necessary when passing a region between ZAF and the native API.

Since ZAF's coordinate systems are based on pixels, `ExportPixel()` exports the region to *rect* as a pixel-based rectangle in the native environment's rectangle type without any conversion and `ImportPixel()` imports the region from *rect* as a pixel-based rectangle in the native environment's rectangle type without any conversion. `ExportPoint()` exports the region to *rect* as a point-based rectangle in the native environment's rectangle type by adding one to the right and bottom boundaries of the `ZafRegionStruct` object and `ImportPoint()` imports the region from *rect* as a point-based rectangle in the native environment's rectangle type by subtracting one from the right and bottom boundaries of the native rectangle.

```

Height        int Height(void) const;

```

`Height()` returns the height of the region.

```

HzShift       void HxShift(int shift);

```

`HxShift()` modifies the region along the x-axis by the amount specified in *shift*. If *shift* is positive, the region moves in the positive direction. If *shift* is negative, the region moves in the negative direction.

```

left
top
right
bottom       int left, top, right, bottom;

```

left, *top*, *right*, and *bottom* store the boundaries of the region.

Overlap

```
bool Overlap(const ZafRegionStruct &region) const;  
bool Overlap(const ZafPositionStruct &position) const;  
bool Overlap(const ZafPositionStruct &position, int  
    columnShift, int lineShift) const;  
bool Overlap(const ZafRegionStruct &region,  
    ZafRegionStruct &result) const;
```

Overlap() returns true if the arguments specified overlap the region; otherwise it returns false. If the coordinate systems are different, a converted version of *region* or *position* is used in the comparison. *region* and *position* specify the object to be compared. *columnShift* and *lineShift* specify the amount using the same coordinate system as this region that *position* is shifted before comparing. The original *region* or *position* is not modified. In the fourth method, the intersection of the two regions are returned in result if the regions overlap.

Region

```
ZafRegionStruct Region(ZafCoordinateType type);
```

Region() returns the region converted to the *type* coordinate system. The original region is not modified.

Touching

```
bool Touching(const ZafPositionStruct &position) const;
```

Touching() returns true if *position* lies exactly on one of the four boundaries of this region; otherwise it returns false. If the coordinate systems are different, a converted version of *position* is used in the comparison.

VtShift

```
void VtShift(int shift);
```

VtShift() modifies the region along the y-axis by the amount specified in *shift*. If *shift* is positive, the region moves in the positive direction. If *shift* is negative, the region moves in the negative direction.

Width

```
int Width(void) const;
```

Width() returns the width of the region.

operator ==

```
bool operator==(const ZafRegionStruct &region) const;
```

This operator returns true if *region* is equal to this region object; otherwise it returns false. If the coordinate systems are different, a converted version of *region* is used in the comparison.

operator != `bool operator!=(const ZafRegionStruct ®ion) const;`

This operator returns true if *region* is not equal to this region object; otherwise it returns false. If the coordinate systems are different, a converted version of *region* is used in the comparison.

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

The prefix and postfix versions of this operator enlarge the region by subtracting one from the left and top boundaries and by adding one to the right and bottom boundaries.

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

The prefix and postfix versions of this operator shrink the region by adding one to the left and top boundaries and by subtracting one from the right and bottom boundaries.

operator += `ZafRegionStruct &operator+=(int offset);`

This operator enlarges the region by subtracting *offset* from the left and top boundaries and by adding *offset* to the right and bottom boundaries.

operator -= `ZafRegionStruct &operator-=(int offset);`

This operator shrinks the region by adding *offset* to the left and top boundaries and by subtracting *offset* from the right and bottom boundaries.

ZafRelativeConstraint

Center Event	OppositeSide Ratio	Stretch Type
-----------------	-----------------------	-----------------

Inheritance	ZafRelativeConstraint : ZafConstraint : ZafElement
Declaration	#include <z_gmgr.hpp>
Description	ZafRelativeConstraint allows a window object to be placed on its parent window object relative to a ratio of the parent’s available region (see Ratio() for more information). ZafRelativeConstraint objects must be added to the ZafGeometryManager object that has been added to the managed object’s parent (see ZafGeometryManager for more information).
Constructors	All ZafRelativeConstraint constructors initialize the member variables associated with an instantiated ZafRelativeConstraint object. The default values set by the ZafRelativeConstraint and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafRelativeConstraint	
Center()	false
OppositeSide()	false
Ratio()	0
Stretch()	false
Type()	user-supplied parameter
ZafElement	
ClassID()	ID_ZAF_RELATIVE_CONSTRAINT
ClassName()	"ZafRelativeConstraint"

```
ZafRelativeConstraint(ZafWindowObject *object,  
                      ZafRelativeConstraintType type);
```

This constructor is useful in straight-code situations to create a ZafRelativeConstraint object. *object* specifies the window object the constraint applies to, and *type* specifies the type of relative constraint. See [ZafConstraint::Object\(\)](#) and [ZafRelativeConstraint::Type\(\)](#) for more information.

```
ZafRelativeConstraint(const ZafRelativeConstraint &copy);
```

The copy constructor creates a new ZafRelativeConstraint object and initializes its data from *copy*.

```
ZafRelativeConstraint(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 relative constraint follows:

```
// Create a window with a geometry-managed child.
ZafWindow *win1 = new ZafWindow(1, 1, 60, 8);
ZafGeometryManager *geo = new ZafGeometryManager;

// The button will occupy the bottom right third of the window.
ZafButton *bottomRight = new ZafButton(15, 2, 15, 1, new
    ZafStringData("bottom right"));

//Attach directly to right side.
ZafAttachment *attach = new ZafAttachment(bottomRight,
    ZAF_ATCF_RIGHT);
attach->SetOffset(0);
geo->Add(attach);

//Attach directly to bottom.
attach = new ZafAttachment(bottomRight, ZAF_ATCF_BOTTOM);
attach->SetOffset(0);
geo->Add(attach);

//Attach left to imaginary 66% vertical divider line in window.
ZafRelativeConstraint *rel = new
    ZafRelativeConstraint(bottomRight, ZAF_RLCF_LEFT);
rel->SetRatio(66);
rel->SetStretch(true);
geo->Add(rel);

//Attach top to imaginary 66% horizontal divider line in window.
rel = new ZafRelativeConstraint(bottomRight, ZAF_RLCF_TOP);
rel->SetRatio(66);
rel->SetStretch(true);
geo->Add(rel);

// Add the button and geometry manager to the window.
win1->Add(bottomRight);
```

```
win1->Add(geo);
```

Destructor

```
virtual ~ZafRelativeConstraint(void);
```

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

Generally, the programmer will not directly destroy a ZafRelativeConstraint object, since it is automatically destroyed when its parent geometry manager 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.

Center

```
bool Center(void) const;
int SetCenter(bool center);
```

If Center() is true, then Object() is centered at Ratio(). For example, if Type() is ZAF_RLCF_LEFT, Ratio() is 50, and Center() is true, then Object() will be centered horizontally on its parent. If Center() is false, then the left edge of Object() would be horizontally at the center of its parent. This attribute defaults to false, but the programmer may change it with SetCenter().

Event

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

This overloaded function handles all events sent to the ZafRelativeConstraint object. The events handled by ZafRelativeConstraint are as follows:

ZafEventType	Description
S_COMPUTE_SIZE	causes the constraint to compute and modify the size or position of its window object
S_INITIALIZE	causes the constraint to initialize its numberID, stringID, and Object()

OppositeSide

```
bool OppositeSide(void) const;
bool SetOppositeSide(bool oppositeSide);
```

If `OppositeSide()` is true, then the opposite side of the parent object is used when computing the position of `Object()`. For example, if `Type()` is `ZAF_RLCF_LEFT`, `Ratio()` is 33, and `OppositeSide()` is true, then the left edge of `Object()` will be placed a third of the way from the right side of the parent. If `OppositeSide()` is false, then the left edge of `Object()` would be placed a third of the way from the left side of the parent. This attribute defaults to false, but the programmer may change it with `SetOppositeSide()`.

Ratio

```
int Ratio(void) const;
int SetRatio(int ratio);
```

`Ratio()` is the percentage used when calculating the relative offset of `Object()`. `Ratio()` is specified as a percentage, so 50% is specified as `SetRatio(50)`. For example, if `Type()` is `ZAF_RLCF_LEFT`, `Ratio()` is 50, and `OppositeSide()` is false, then the left edge of `Object()` will be centered horizontally on its parent. As another example, if `Type()` is `ZAF_RLCF_RIGHT`, `Ratio()` is 50, and `OppositeSide()` is false, then the right edge of `Object()` will be centered horizontally on its parent. This attribute defaults to 0, but the programmer may change it with `SetRatio()`.

Stretch

```
bool Stretch(void) const;
bool SetStretch(bool stretch);
```

If `Stretch()` is true, then the opposite side of `Object()` than that specified by `Type()` is not modified. For example, if `Type()` is `ZAF_RLCF_RIGHT`, `Ratio()` is 66, and `Stretch()` is true, then the right edge of `Object()` will be placed a third of the way from the right edge of its parent and the left edge of `Object()` will remain where it is, in effect making it possible to stretch `Object()`. If `Stretch()` is false, then the left edge of `Object()` would be modified so that the width of `Object()` would remain the same. This attribute defaults to false, but the programmer may change it with `SetStretch()`.

Type

```
ZafRelativeConstraintType Type(void) const;
ZafRelativeConstraintType
    SetType(ZafRelativeConstraintType type);
```

`Type()` is the relative constraint's type, which specifies the side of `Object()` used when calculating its position. If `OppositeSide()` is false, then `Type()` also specifies the side of the parent object used when calculating the position of `Object()`. If `OppositeSide()` is true, then the opposite side of the parent object

is used when calculating the position of Object(). The programmer may use SetType() to change this attribute. The possible values for Type() follow:

Type()	Description
ZAF_RLCF_BOTTOM	causes the constraint to affect the bottom side of Object()
ZAF_RLCF_LEFT	causes the constraint to affect the left side of Object()
ZAF_RLCF_RIGHT	causes the constraint to affect the right side of Object()
ZAF_RLCF_TOP	causes the constraint to affect the top side of Object()

ZafScreenDisplay

colorTable	lineTable	monoTable
fontTable	modeTable	patternTable

Inheritance ZafScreenDisplay : [ZafDisplay](#)

Declaration `#include <z_scrdsp.hpp>`

Description ZafScreenDisplay defines the basic functionality necessary to interface with the screen. ZafScreenDisplay 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. See the base class [ZafDisplay](#) for complete descriptions of the display functions provided by ZafScreenDisplay.

ZafScreenDisplay is important to displayable ZAF classes because it provides the interface necessary to interact with the screen. A ZafScreenDisplay object is instantiated by the ZafApplication class, and should not be instantiated by the programmer. When needed, the method [ZafWindowObject::Display\(\)](#) should be used to access the ZafScreenDisplay object instantiated automatically by the ZafApplication class.

Constructor The ZafScreenDisplay constructor initializes the member variables associated with an instantiated ZafScreenDisplay object. The default values set by the ZafScreenDisplay constructor and its base class constructor follow, if they differ from those set by the base class constructor.

Member Initializations

ZafScreenDisplay

<code>colorTable[]</code>	display-dependent
<code>fontTable[]</code>	display-dependent
<code>lineTable[]</code>	display-dependent
<code>modeTable[]</code>	display-dependent
<code>monoTable[]</code>	display-dependent
<code>patternTable[]</code>	display-dependent

ZafDisplay

<code>cellHeight</code>	font-dependent
<code>cellWidth</code>	font-dependent
<code>columns</code>	display-dependent
<code>coordinateType</code>	display-dependent

Member Initializations

<code>DisplayType()</code>	<code>"ZafScreenDisplay"</code>
<code>lines</code>	<code>display-dependent</code>
<code>pixelsPerInchX</code>	<code>display-dependent</code>
<code>pixelsPerInchY</code>	<code>display-dependent</code>
<code>postSpace</code>	<code>environment-specific</code>
<code>preSpace</code>	<code>environment-specific</code>

```
ZafScreenDisplay(int &argc, char **argv);
```

The constructor is useful in straight-code situations. *argc* and *argv* are simply the *argc* and *argv* parameters passed into the application, and are used by some environments to find the application's name to be registered with the system and/or to allow the user to specify a display mode (see the *readme.** files in the *zaf/readme* directory for more information). This constructor will not normally be called by the programmer, since it is automatically called by *ZafApplication*.

Destructor

```
virtual ~ZafScreenDisplay(void);
```

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

Members

colorTable

```
OSColor colorTable[ZAF_MAXCOLORS];
```

The *colorTable* array 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 *ZafDisplay::AddColor()* for a description of adding support for RGB colors.

fontTable

```
OSFont fontTable[ZAF_MAXFONTS];
```

The *fontTable* array 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 *ZafDisplay::AddFont()* for a description of adding support for available fonts.

lineTable `OSLineStyle lineTable[ZAF_MAXLINES];`

The `lineTable` array 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.

modeTable `OSMode modeTable[ZAF_MAXMODES];`

The `modeTable` array 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.

monoTable `OSMono monoTable[ZAF_MAXCOLORS];`

The `monoTable` array 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.

patternTable `OSFillPattern patternTable[ZAF_MAXPATTERNS];`

The `patternTable` array 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.

ZafScrollBar

	AutoSize	ScrollData	ScrollType
Inheritance	ZafScrollBar : ZafWindow : (ZafWindowObject : ZafElement), ZafList		
Declaration	#include <z_scrll2.hpp>		
Description	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.		
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†
Modal()	false†
Moveable()	false†
NormalHotKeys()	false†
SelectionType()	ZAF_SINGLE_SELECTION†
Sizeable()	false†
Temporary()	false†
ZafWindowObject	
AcceptDrop()	false†
Bordered()	false†

Member Initializations

ParentPalette()	false [†]
SupportObject()	true (if not a slider object)

ZafElement

ClassID()	ID_ZAF_SCROLL_BAR
ClassName()	"ZafScrollBar"

ZafScrollBar(ZafScrollBarType scrollType);

This constructor is useful in straight-code situations, particularly if the ZafScrollBar object is to be a support object for its parent. In this case, the ZafScrollBar object's data is initialized by the parent object, so no data values are necessary.

scrollType specifies the type of ZafScrollBar object to be created. See [ScrollType\(\)](#) for more information.

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 [ScrollData\(\)](#) 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.

AutoSize

```
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.

ScrollData

```
ZafScrollData *ScrollData(void) const;
virtual ZafError SetScrollData(ZafScrollData *scroll);
```

The `ScrollData()` object contains the actual data used by the `ZafScrollBar`. `ZafScrollData` 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. `SetScrollData()` will delete the previous `ScrollData()` object if it is `Destroyable()` and no other object uses it.

`ScrollData()` returns a pointer to the `ScrollData()` object associated with the `ZafScrollBar`. `SetScrollData()` normally returns `ZAF_ERROR_NONE`.

ScrollType

```
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
<code>ZAF_CORNER_SCROLL</code>	Corner scroll bar (typically used for spacing when both horizontal and vertical scroll bars are present)
<code>ZAF_HORIZONTAL_SCROLL</code>	Horizontal scroll bar
<code>ZAF_VERTICAL_SCROLL</code>	Vertical scroll bar
<code>ZAF_HORIZONTAL_SLIDER</code>	Horizontal slider control
<code>ZAF_VERTICAL_SLIDER</code>	Vertical slider control
<code>ZAF_SIZEGRIP_SCROLL</code>	Corner scroll bar with size-grip appearance (typically used on <code>ZafWindow</code> objects for a Windows95 look)

ZafScrollData

Clear	Increment	Showing
Current	Maximum	Step
Decrement	Minimum	operator =
Delta	SetScroll	

Inheritance `ZafScrollData : (ZafData : (ZafNotification, ZafElement)), ZafScrollStruct`

Declaration `#include <z_scroll11.hpp>`

Description ZafScrollData objects can be used to store and manipulate scrolling information.

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

Constructors All ZafScrollData constructors initialize the member variables associated with an instantiated ZafScrollData object. The default values set by the ZafScrollData and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafScrollData

<code>Current()</code>	user-supplied parameter
<code>Delta()</code>	user-supplied parameter
<code>Maximum()</code>	user-supplied parameter
<code>Minimum()</code>	user-supplied parameter
<code>Showing()</code>	user-supplied parameter
<code>Step()</code>	same attribute as <code>Delta()</code>

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_SCROLL_DATA</code>
<code>ClassName()</code>	<code>"ZafScrollData"</code>

zafscrollData(long minimum = 0, long maximum = 0, long current = 0, long delta = 1, long showing = 1);

This constructor allocates a ZafScrollData instance and initializes its contents according to the values passed in. *minimum* is the minimum value the scroll

data can be, *maximum* is the maximum value, *current* is the current value, *delta* is the amount the scroll data changes for every scroll, and *showing* is the amount the scroll data changes when paging (page-up or page-down).

```
ZafScrollData(ZafScrollStruct &data);
```

This constructor allocates a ZafScrollData instance and initializes its data to the values in *data*.

```
ZafScrollData(const ZafScrollData &copy);
```

The copy constructor creates a new ZafScrollData object and initializes its data from *copy*.

```
ZafScrollData(const ZafIChar *name, ZafDataPersistence  
               &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 code snippet shows how to create a ZafScrollData object:

```
// Create some ZafScrollData objects.  
ZafScrollData scroll1;  
ZafScrollData scroll2 = scroll1;  
ZafScrollData scroll3(1, 100, 1, 1, 10);
```

Destructor

```
virtual ~ZafScrollData(void);
```

This virtual destructor is used to free the memory associated with an instantiated ZafScrollData object. Unless StaticData() is true, a ZafScrollData object will be destroyed automatically when all ZafScrollBar objects that refer to it are 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.

Clear virtual void **Clear**(void);

Clear() sets the scroll data values as follows: current is set to 0, delta is set to 1, maximum is set to 0, minimum is set to 0, and showing is set to 1.

Current long **Current**(void) const;
virtual long **SetCurrent**(long current);

Current() is the current scroll value. It may be changed by calling SetCurrent().

Decrement long **Decrement**(long value);

Decrement() subtracts Delta() from Current(). If the result would be less than Minimum(), then Current() becomes Minimum().

Delta long **Delta**(void) const;
virtual long **SetDelta**(long delta);

Delta() is the amount that Current() changes during a scroll operation. SetDelta() may be called to change it.

Increment long **Increment**(long value);

Increment() adds Delta() to Current(). If the result would be greater than Maximum(), then Current() becomes Maximum().

Maximum long **Maximum**(void) const;
virtual long **SetMaximum**(long maximum);

Maximum() is the maximum value that Current() may be. SetMaximum() may be called to change it.

Minimum long **Minimum**(void) const;
virtual long **SetMinimum**(long minimum);

Minimum() is the minimum value that Current() may be. SetMinimum() may be called to change it.

SetScroll

```
virtual ZafError SetScroll(long minimum, long maximum,
    long current, long delta, long showing);
virtual ZafError SetScroll(const ZafScrollStruct
    &scroll);
```

SetScroll() may be called to modify all of the scrolling values at once. *minimum* is the minimum value the scroll data can be, *maximum* is the maximum value, *current* is the current value, *delta* is the amount the scroll data changes for every scroll, and *showing* is the amount the scroll data changes when paging (page-up or page-down). Since all these values are stored in the ZafScrollStruct, *data* refers to a ZafScrollStruct whose values are to be copied into the ZafScrollData object.

Showing

```
long Showing(void) const;
virtual long SetShowing(long showing);
```

Showing() specifies the amount the scroll data changes when paging (page-up or page-down). SetShowing() may be called to change it.

Step

```
long Step(void) const;
virtual long SetStep(long step);
```

Step() is identical to Delta(), and SetStep() is identical to SetDelta().

operator =

```
ZafScrollData &operator=(const ZafScrollData &scroll);
ZafScrollData &operator=(const ZafScrollStruct &scroll);
```

These operators set the scroll values of the ZafScrollData object to be the same as the *scroll* parameters.

ZafScrollStruct

current	minimum	operator !=
delta	showing	
maximum	operator ==	

Inheritance	Root struct
Declaration	#include <z_scrll.hpp>
Description	ZafScrollStruct objects can be used to store and manipulate scrolling information. ZafScrollStruct is most often used in conjunction with the ZafScrollData class but may be used as a stand-alone object if desired.
Members	
<i>current</i>	<div>long current;</div> <div>current is the current scroll value, and should always be a value between minimum and maximum. current is analogous to the thumb button on a scroll bar.</div>
<i>delta</i>	<div>long delta;</div> <div>delta is the amount that current changes during a scroll operation. delta may be thought of as the amount the thumb button on a scroll bar moves when one of the arrows is selected.</div>
<i>maximum</i>	<div>long maximum;</div> <div>maximum is the maximum value that current may be, and must be greater than or equal to minimum.</div>
<i>minimum</i>	<div>long minimum;</div> <div>minimum is the minimum value that current may be, and must be less than or equal to maximum.</div>
<i>showing</i>	<div>long showing;</div> <div>showing is the amount that current changes when paging (page-up or page-down).</div>

operator ==

```
bool operator==(const ZafScrollStruct &scroll) const;
```

This operator checks equality of two ZafScrollStruct objects and returns true if all the members are equal.

operator !=

```
bool operator!=(const ZafScrollStruct &scroll) const;
```

This operator checks inequality of two ZafScrollStruct objects and returns true if any one of the members are not equal.

ZafScrolledWindow

HzScrollPos	ScrollWidth
ScrollHeight	VtScrollPos

Inheritance	ZafScrolledWindow : ZafWindow : (ZafWindowObject : ZafElement), ZafList
Declaration	#include <z_sclwin.hpp>
Description	ZafScrolledWindow provides support for scrolling the client area of a window with ZafScrollBar support objects. All the functionality of the base ZafWindow class is retained. The programmer must specify the size of the window's scrollable area which may not be more than 32K pixels in either dimension..
Constructors	All ZafScrolledWindow constructors initialize the member variables associated with an instantiated ZafScrolledWindow object. The default values set by the ZafScrolledWindow and its base class constructors follow, if they differ from those set by the base class constructor.

Member Initializations

ZafScrolledWindow

HzScrollPos()	0
ScrollHeight()	user-supplied parameter
ScrollWidth()	user-supplied parameter
VtScrollPos()	0

ZafElement

ClassID()	ID_ZAF_SCROLLED_WINDOW
ClassName()	"ZafScrolledWindow"

```
ZafScrolledWindow(int left, int top, int width, int height, int scrollWidth, int scrollHeight, int hzScrollPos = 0, int vtScrollPos = 0);
```

This constructor is useful in straight-code situations. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired.

left and *top* specify the position where the left and top of the object will be placed on the window manager or its parent. *width* and *height* specify the width and height of the client region of the object.

scrollWidth and *scrollHeight* specify the width and height of the scrollable region of the window. Note neither the *scrollWidth* nor *scrollHeight* can be more than 32K pixels. Since these parameters are often specified using another coordinate system (e.g. ZAF_CELL) the programmer must be careful not to specify a size that will convert to more than 32K pixels.

hzScrollPos and *vtScrollPos* specify the initial position of the viewable area of the window.

```
ZafScrolledWindow(const ZafScrolledWindow &copy);
```

The copy constructor creates a new **ZafScrolledWindow** object and initializes its data from *copy*.

```
ZafScrolledWindow(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 scrolled window follows:

```
// Create a scrolled window.  
ZafScrolledWindow *window = new ZafScrolledWindow(1, 1, 50, 10,  
    100, 100);  
window->AddGenericObjects(new ZafStringData("Scrolled  
    Window"));  
  
// Add the scroll bars to the window.  
window->Add(new ZafScrollBar(0, 0, 0, 0,  
    ZAF_NULLP(ZafScrollData), ZAF_CORNER_SCROLL));  
window->Add(new ZafScrollBar(0, 0, 0, 0,  
    ZAF_NULLP(ZafScrollData), ZAF_VERTICAL_SCROLL));  
window->Add(new ZafScrollBar(0, 0, 0, 0,  
    ZAF_NULLP(ZafScrollData), ZAF_HORIZONTAL_SCROLL));  
  
// Add a string to the window.  
window->Add(new ZafString(60, 15, 10, new  
    ZafStringData("String")));
```

Destructor

```
virtual ~ZafScrolledWindow(void);
```

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

ZafElement destructors. 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.

ScrollHeight

```
int ScrollHeight(void);  
void SetScrollHeight(int height);
```

ScrollHeight() specifies the height of the scrollable area in the same coordinate system as Region(). When ScrollHeight() is larger than Region().Height(), the window's contents may be scrolled vertically. When it is smaller, the vertical scroll bar becomes disabled, since all the window's contents are visible within its vertical client area. SetScrollHeight() may be called to modify the window's scrollable height.

Note the scrollHeight cannot be more than 32K pixels. Since this parameter may be specified using another coordinate system (e.g. ZAF_CELL) the programmer must be careful not to specify a size that will convert to more than 32K pixels.

ScrollWidth

```
int ScrollWidth(void);  
void SetScrollWidth(int width);
```

ScrollWidth() specifies the width of the scrollable area in the same coordinate system as Region(). When ScrollWidth() is larger than Region().Width(), the window's contents may be scrolled horizontally. When it is smaller, the horizontal scroll bar becomes disabled, since all the window's contents are visible within its horizontal client area. SetScrollWidth() may be called to modify the window's scrollable width.

Note the scrollWidth cannot be more than 32K pixels. Since this parameter may be specified using another coordinate system (e.g. ZAF_CELL) the programmer must be careful not to specify a size that will convert to more than 32K pixels.

HzScrollPos

```
int HzScrollPos(void);  
void SetHzScrollPos(int pos);
```

HzScrollPos() specifies how much the window has scrolled horizontally. A value of zero means that the left-most contents of the window are at the left of

the window's client region. A positive value means that the left-most contents of the window are to the left of the window's client region, and thus are not visible. `SetHzScrollPos()` may be called to scroll the window's contents horizontally.

VtScrollPos

```
int VtScrollPos(void);  
void SetVtScrollPos(int pos);
```

`VtScrollPos()` specifies how much the window has scrolled vertically. A value of zero means that the top-most contents of the window are at the top of the window's client region. A positive value means that the top-most contents of the window are above the window's client region, and thus are not visible. `SetVtScrollPos()` may be called to scroll the window's contents vertically.

ZafSpinControl

	Delta	WrappedData
Inheritance	ZafSpinControl : ZafWindow : (ZafWindowObject : ZafElement), ZafList	
Declaration	#include <z_spin.hpp>	
Description	<p>ZafSpinControl allows an end-user to automatically increment and decrement another “value” object. Value objects derive from ZafString and define Increment() and Decrement() functions. They include ZafBignum, ZafDate, ZafTime, ZafUTime, ZafInteger, and ZafReal. ZafSpinControl utilizes native spin control APIs, if available.</p> <p>The user manipulates a spin control by clicking on increment or decrement buttons, or by using the keyboard. Each time the ZafSpinControl is changed it adds or subtracts a “delta” value specified by the programmer until an optional maximum or minimum value is reached. Values may “wrap” at maximum or minimum if desired.</p>	
Constructors	<p>All ZafSpinControl constructors initialize the member variables associated with an instantiated ZafSpinControl object. The default values set by the ZafSpinControl 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 ZafSpinControl. “†” Indicates a blocking function that prevents changes to the attribute in this class.</p>	

Member Initializations

ZafSpinControl

Delta()	user-supplied parameter
WrappedData()	true

ZafWindow

Destroyable()	false†
Locked()	false†
Maximized()	false†
Minimized()	false†
Modal()	false†
Moveable()	false†
NormalHotKeys()	false†
SelectionType()	ZAF_SINGLE_SELECTION†

Member Initializations

Sizeable()	false [†]
Temporary()	false [†]

ZafElement

ClassID()	ID_ZAF_SPIN_CONTROL
ClassName()	"ZafSpinControl"

ZafSpinControl(int left, int top, int width, ZafData *delta);

This constructor is useful in straight-code situations. *left*, *top* and *width* specify the position and size of the ZafSpinControl (which includes its child object). ZafSpinControl objects are always one cell high. *left*, *top*, and *width* are specified in cell coordinates by default, but may be specified using another coordinate system if desired. See ZafWindowObject::SetCoordinateType() for more information.

delta may be an instance of any ZafData subclass, but the spin control's child object must be able to utilize delta when incrementing and decrementing. For more information on using ZafData objects, see ZafData. The following are ZAF classes already supported by ZafSpinControl, listed together with the corresponding delta class that must be used:

ZafSpinControl child object	Corresponding delta object
ZafBignum	ZafBignumData
ZafDate	ZafDateData
ZafInteger	ZafIntegerData
ZafReal	ZafRealData
ZafTime	ZafTimeData
ZafUTime	ZafUTimeData

ZafSpinControl(const ZafSpinControl ©);

The copy constructor is used in conjunction with the overloaded Duplicate() function. It allocates a new ZafSpinControl object and initializes its data from *copy*. If the original ZafSpinControl's internal data objects are StaticData() then the new ZafSpinControl object points to the originals, otherwise copies are made.

```
ZafSpinControl(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 `ZafSpinControl` objects:

```
// Create a spin control using a ZafInteger object and
// ZafIntegerData delta.
ZafIntegerData *delta = new ZafIntegerData(1);

// Use the ZafIntegerData delta with the spin control.
ZafSpinControl *spinner = new ZafSpinControl(1, 1, 12, delta);

// Add the ZafInteger field object to the spin control.
spinner->Add(new ZafInteger(0, 0, 0, 1));
window->Add(spinner);
```

Destructor

```
virtual ~ZafSpinControl(void);
```

The destructor is used to free the memory associated with a `ZafSpinControl` 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 `ZafSpinControl` 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. If the `Set*()` function does not successfully change the state as requested, however, it will instead return the current state.

Delta

```
const ZafData *Delta(void);
virtual ZafError SetDelta(ZafData *data);
```

The `Delta()` object contains the actual data used by the `ZafSpinControl` when spinning its child object. Each time the child object is spun, it is incremented or decremented by the value of `Delta()`. The `Delta()` object must be an instance of a class that the spin control's child object supports with its `Increment()` and `Decrement()` functions. `ZafData` objects may be shared among several `ZafSpinControl` objects (to save memory, for example) or they may belong to a single `ZafSpinControl` object. `SetDelta()` may be used to associate a `Delta()`

object with a ZafSpinControl object. See [ZafDataManager](#) for more information on data sharing.

Delta() returns a pointer to the Delta() object associated with the ZafSpinControl. SetDelta() normally returns ZAF_ERROR_NONE.

WrappedData

```
bool WrappedData(void) const;  
virtual bool SetWrappedData(bool wrappedData);
```

If WrappedData() is true, the spin control wraps from end to beginning or from beginning to end when it reaches the maximum or minimum of the value or of the range. This attribute is true by default, but SetWrappedData() may be called to change it.

ZafSplitter

Live	Position	SplitterType
NextPaneObject	PreviousPaneObject	Thickness

Inheritance ZafSplitter : ZafWindowObject : ZafElement

Declaration #include <z_split.hpp>

Description The ZafSplitter object allows a window to be divided or “split” into sections or “panes” so that different information may be displayed in each pane. The panes may also be sized by the user at runtime (by the end user). The ZafSplitter object may be added to any type of window to provide this functionality.

Each splitter object divides a region called a “split region” into two logical panes. The split region is divided horizontally or vertically depending on the type of splitter. Each pane is simply the region within the split region on one side or the other of the visual splitter object. The pane above or to the left of the splitter object is called the next pane.

While a pane is only a conceptual region, a “pane object” is a window object that is forced to occupy an entire pane. Each splitter object controls the sizing of two pane objects, one for each pane. Its previous pane object occupies its entire previous pane, and its next pane object occupies its entire next pane. The splitter object acts as a geometry manager for its pane objects, forcing them to occupy its panes.

In order to achieve multiple splits within a window, a splitter object may also be a pane object. As a pane object, a splitter object divides its pane into sub-panes.

This class is not available in the Personal or Registered versions of ZAF, but is included with the Professional version.

Constructors All ZafSplitter constructors initialize the member variables associated with an instantiated ZafSplitter object. The default values set by the ZafSplitter 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 ZafSplitter. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafSplitter	
Live()	true (false on Motif)
NextPaneObject()	null
Position()	50 percent

Member Initializations

PreviousPaneObject()	null
SplitterType()	parameter
Thickness()	3

ZafWindowObject

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

ZafElement

ClassID()	ID_ZAF_SPLITTER
ClassName()	"ZafSplitter"

```
ZafSplitter(int left, int top, int width, int height,
             ZafSplitterType splitType, int position = 50, bool
             percent = true);
```

This constructor is useful in straight-code situations. *left*, *top*, *width*, and *height* specify the location and size of the split region within the parent window. Since splitters are usually ZAF_AVAILABLE_REGION or pane objects, these parameters usually have no effect (the split region is determined by the available region or a controlling splitter object) and may be zero. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired.

splitType specifies the orientation of the splitter object, and may be ZAF_HORIZONTAL_SPLITTER or ZAF_VERTICAL_SPLITTER.

position specifies the position of the splitter object within the split region. If *splitType* is ZAF_HORIZONTAL_SPLITTER, *position* specifies the distance from the top edge of the split region to the top edge of the splitter object. If

splitType is ZAF_VERTICAL_SPLITTER, *position* specifies the distance from the left edge of the split region to the left edge of the splitter object.

percent specifies whether or not *position* is given as a percentage. If *percent* is true, *position* specifies the percentage of the split region that will be in the previous pane; otherwise *position* specifies the size (height for horizontal splitters or width for vertical splitters) of the previous pane, and the units (ZAF_CELL, ZAF_MINICELL, etc.) are specified by the coordinate type of the splitter object.

```
ZafSplitter(ZafSplitterType splitType, int position = 50,  
             bool percent = true);
```

This constructor is useful in straight-code situations. Since splitters are usually ZAF_AVAILABLE_REGION or pane objects, this constructor doesn't require *left*, *top*, *width*, and *height* parameters for ease of use. See the previous constructor for descriptions of the parameters to this constructor.

```
ZafSplitter(const ZafSplitter &copy);
```

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

```
ZafSplitter(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 showing how to use ZafSplitter:

```
// Create a sample window with a splitter object.  
ZafWindow *window = new ZafWindow(0, 0, 60, 10);  
ZafSplitter *splitter = new  
    ZafSplitter(ZAF_HORIZONTAL_SPLITTER);  
splitter->SetRegionType(ZAF_AVAILABLE_REGION);  
window->Add(splitter);  
  
// Add a text object to the top side of the splitter.  
ZafStringData *data = new ZafStringData("Sample text object.")  
ZafText *text = new ZafText(0, 0, 0, 0, data);  
text->Add(new ZafScrollBar(ZAF_VERTICAL_SCROLL));  
splitter->SetPreviousPaneObject(text);  
window->Add(text);
```

```
// Add a text object to the other side of the splitter.
text = new ZafText(0, 0, 0, 0, data);
text->Add(new ZafScrollBar(ZAF_VERTICAL_SCROLL));
splitter->SetNextPaneObject(text);
window->Add(text);
```

Destructor

```
virtual ~ZafSplitter(void);
```

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

Generally, the programmer will not directly destroy a ZafSplitter 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.

Live

```
bool Live(void) const;
virtual bool SetLive(bool live);
```

If Live() is true, pane objects will resize as the splitter is being positioned (while the mouse is dragging); otherwise the pane objects won't resize until the mouse button is released. The default value of this attribute is true, but the user may call SetLive() to change it (except in Motif, where this attribute is always false).

NextPaneObject PreviousPaneObject

```
ZafWindowObject *NextPaneObject(void) const;
ZafWindowObject *PreviousPaneObject(void) const;
virtual void SetNextPaneObject(ZafWindowObject *object);
virtual void SetPreviousPaneObject(ZafWindowObject
    *object);
```

These functions are used to get and set the next and previous pane objects. SetNextPaneObject() and SetPreviousPaneObject() both modify the regions of the objects being set to occupy the appropriate panes. These attributes default to null, but may be set by the programmer using SetNextPaneObject() and SetPreviousObject().

Position

```
int Position(bool percent = true) const;
virtual void SetPosition(int position, bool percent =
    true);
```

Position() gets a splitter object's position and **SetPosition()** sets it. If *percent* is true, the position specifies the percentage of the split region contained in the previous pane; otherwise the position specifies the size (height for horizontal splitters and width for vertical splitters) of the previous pane, and the units (**ZAF_CELL**, **ZAF_MINICELL**, etc.) are specified by the coordinate type of the splitter object.

SplitterType

```
ZafSplitterType SplitterType(void) const;
virtual void SetSplitterType(ZafSplitterType
    splitterType);
```

SplitterType() and **SetSplitterType()** are used to get and set the type of a splitter object. The valid splitter types are **ZAF_HORIZONTAL_SPLITTER** and **ZAF_VERTICAL_SPLITTER**. A horizontal splitter is a horizontal bar that splits the split region into two panes, one above the splitter object and one below the splitter object. A vertical splitter is a vertical bar that splits the split region into two panes, one to the left of the splitter object and one to the right of the splitter object.

Thickness

```
int Thickness(void) const;
virtual void SetThickness(int thickness);
```

Thickness() specifies the thickness of the splitter object in pixels. The thickness of a splitter object is the height of a horizontal splitter bar or the width of a vertical splitter bar. (Height and width refer to the **zafRegion** or in other words, the height and width of the actual splitter object.) The default value of this attribute is 3, but the user may call **SetThickness()** to change it.

ZafStandardArg

RearrangeArgs
SetSprintf
SetSscanf
vsprintf

vsscanf

Inheritance Root class

Declaration `#include <z_stdarg.hpp>`

Description ZafStandardArg provides support for internationalized standard argument functions that work correctly with Unicode strings, as well as ISO strings. ZAF standard functions such as `sprintf()` and `sscanf()` utilize ZafStandardArg to internationalize their use of standard arguments. These functions should generally be used in ZAF applications instead of their standard library equivalents. As these functions are all static, and ZafStandardArg has no constructor, no ZafStandardArg object needs to be created by the programmer.

Members `static void RearrangeArgs(bool isSscanf, void *newBuffer, const ZafIChar *format, const va_list ap, ZafIChar *newFormat, va_list *toRet);`

RearrangeArgs RearrangeArgs() is at the heart of ZAF's internationalization of variable argument lists. It allows a format character to specify which variable argument it matches with, regardless of its order within the format string. This is especially useful when string translations causes the order of translated words to be different. RearrangeArgs() is generally not called by the programmer, since the standard ZAF functions call it when necessary.

isSscanf should be true when RearrangeArgs() requires the `sscanf()` functionality of [...] arguments. *newBuffer* is the buffer used to store the new variable argument list, so it must be large enough to hold them (and the Borland compilers require this buffer to be on the stack). *format* is the original format string and *ap* is the original variable argument list. *newFormat* is the resulting format string, and *toRet* is the resulting variable argument list.

For example, to specify the second variable argument in a format character for a string, "%2s" would be used in the format string. The following code snippet shows how this feature is used within ZAF's `sscanf()`:

```
int sscanf(const ZafIChar *dst, const ZafIChar *format, ...)
{
    // Rearrange the arguments before passing to _vsscanf().
    va_list args;
    va_start(args, format);
    #if defined(ZAF_REARRANGEARGS)
```

```
ZafIChar *newFormat = new ZafIChar[ZAF_MAXPARAMLEN];
char buff[ZAF_MAXPARAMLEN];    // Borland expects this in the
    stack.
va_list newArgs;
ZafStandardArg::RearrangeArgs(true, buff, format, args,
    newFormat, &newArgs);
int i = ZafStandardArg::_vsscanf(dst, newFormat, newArgs);
delete []newFormat;
#else
    int i = ZafStandardArg::_vsscanf(dst, format, &args);
#endif
    va_end(args);
    return i;
}
```

SetSprintf

```
static void SetSprintf(ZafIChar fmtCH, objectFormat
    format);
```

SetSprintf() adds support for the format character *fmtCH* through the format routine *format*. This routine should normally not be called by the programmer, but is used internally by ZAF. For example, ZafIntegerData adds support for the format character 'd' in the format routine ZafIntegerData::Format() by calling ZafStandardArg::SetSprintf('d', ZafIntegerData::Format). Thereafter, calls to sprintf() using the format character 'd' will be formatted by passing the argument to ZafIntegerData::Format().

SetSscanf

```
static void SetSscanf(ZafIChar fmtCH, objectParse parse);
```

SetSscanf() adds support for the format character *fmtCH* through the parsing routine *parse*. This routine should normally not be called by the programmer, but is used internally by ZAF. For example, ZafIntegerData adds support for the format character 'd' in the parsing routine ZafIntegerData::Parse() by calling ZafStandardArg::SetSscanf('d', ZafIntegerData::Parse). Thereafter, calls to sscanf() using the format character 'd' will be parsed by passing the argument to ZafIntegerData::Parse().

vsprintf

```
static int vsprintf(ZafIChar *buffer, const ZafIChar
    *format, va_list *argList);
static int _vsprintf(ZafIChar *fp, const ZafIChar *fmt,
    va_list *ap);
```

vsprintf() calls RearrangeArgs() if the macro ZAF_REARRANGEARGS is defined, then calls _vsprintf() with the fixed arguments. If ZAF_REARRANGEARGS is not defined, then vsprintf() simply calls _vsprintf() with the same arguments. *buffer* is the output buffer, *format* is the

format string, and *argList* is the list of variable arguments. See the compiler's documentation for the standard function `vsprintf()` for more information.

In addition to the normal format options supported by the standard library `vsprintf()`, ZAF's `vsprintf()` provides a few more:

Format option	Description
%Z[\$@+- ,]B	allows a <code>ZafBignumData</code> pointer to be an argument (see <code>ZafBignumData::FormattedText()</code> for more information)
%ZU	allows a <code>ZafUTimeData</code> pointer to be an argument

These format options are normally only used internally by ZAF. The following code snippet shows the preferred method of using `sprintf()` with a complex data object:

```
// Load outputString with a bignum.
ZafBignumData bignum(123.456);
ZafIChar bignumString[64], outputString[64];
bignum.FormattedText(bignumString, strlen(bignumString),
    "$@B");
sprintf(outputString, "Bignum: %s", bignumString);
```

vsscanf

```
static int vsscanf(const ZafIChar *buffer, const ZafIChar
    *format, va_list *argList);
static int _vsscanf(const ZafIChar *fp, const ZafIChar
    *fmt, va_list *ap);
```

`vsscanf()` calls `RearrangeArgs()` if the macro `ZAF_REARRANGEARGS` is defined, then calls `_vsscanf()` with the fixed arguments. If `ZAF_REARRANGEARGS` is not defined, then `vsscanf()` simply calls `_vsscanf()` with the same arguments. *buffer* is the input buffer, *format* is the format string, and *argList* is the list of variable arguments. See the compiler's documentation for the standard function `vsscanf()` for more information.

ZafStatusBar

Inheritance `ZafStatusBar : ZafWindow : (ZafWindowObject :
 ZafElement), ZafList`

Declaration `#include <z_status.hpp>`

Description The ZafStatusBar object is generally used for informational purposes. For example, a string field on a status bar may display help messages (see [ZafHelp-Tips](#) 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 Noncurrent(), since the status bar is Noncurrent(), meaning that it does not receive focus.

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

<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>Modal()</code>	<code>false[†]</code>
<code>Moveable()</code>	<code>false[†]</code>
<code>NormalHotKeys()</code>	<code>false[†]</code>
<code>SelectionType()</code>	<code>ZAF_SINGLE_SELECTION[†]</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>Focus()</code>	<code>false[†]</code>
<code>Noncurrent()</code>	<code>true[†]</code>
<code>RegionType()</code>	<code>ZAF_AVAILABLE_REGION[†]</code>
<code>SupportObject()</code>	<code>true[†]</code>
<code>UserFunction()</code>	<code>null[†]</code>

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 ©);

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 status bar 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\(\)](#).

ZafStorage

ChDir	GetCWD	Remove
Close	MkDir	Rename
Create	Open	Rmdir
FindClose	OpenCreate	Save
FindFirst	ReadOnly	SaveAs
FindNext	ReadWrite	Version

Inheritance ZafStorage : [ZafFileSystem](#) : [ZafElement](#)

Declaration #include <z_store.hpp>

Description ZafStorage and [ZafStorageFile](#) provide support for persistence in ZAF. Any information may be portably stored with ZafStorage, and most ZAF classes provide built-in support for persistence through ZafStorage.

ZafStorage may be thought of as a file system. There can be many ZafStorage-Files and levels of subdirectories. ZafStorage allows ZafStorageFiles to be copied, deleted, and moved across directories.

ZafStorage is typically used for persistent objects (usually created in Zinc Designer) and for storing internationalization data. It is also commonly used as a simple database.

Constructors All ZafStorage constructors initialize the member variables associated with an instantiated ZafStorage object. Default values set by the ZafStorage follow.

Member Initializations

ZafStorage	
Version()	0

```
ZafStorage(const ZafIChar *name, ZafFileMode mode =
ZAF_FILE_READWRITE);
```

This constructor initializes the members associated with a ZafStorage object and opens the disk file specified by *name*. The file is opened using the *mode* file mode (see the [ZafFile](#) constructor for more information). This constructor chains to the [ZafFileSystem](#) and [ZafElement](#) constructors.

```
ZafStorage(void);
```

This constructor initializes the members associated with a `ZafStorage` object and opens a temporary storage object that is useful to store and retrieve temporary information. This constructor chains to the `ZafFileSystem` and `ZafElement` constructors.

Destructor

```
virtual ~ZafStorage(void);
```

This virtual destructor is used to free the memory associated with an instantiated `ZafStorage` object and chains to the `ZafFileSystem` and `ZafElement` destructors.

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.

ChDir

```
virtual int ChDir(const ZafIChar *newPath, ZafStringID  
    stringID = ZAF_NULLP(ZafIChar), ZafNumberID numberID =  
    0);
```

`ChDir()` changes the current `ZafStorage` directory. *newPath* specifies the path name of the directory to be changed to. *stringID* specifies the string identification constant associated with the directory and *numberID* specifies the numeric identification constant associated with the directory. `ChDir()` returns 0 on success, and -1 on failure. `Error()` is also set to an appropriate value.

Close

```
virtual void Close(ZafFile *file);
```

`Close()` closes *file*, which is a `ZafStorageFile` previously opened with `Open()`.

Create

```
bool Create(void) const;
```

If `Create()` is true, the `ZafStorage` object is created, even if it already exists.

FindClose

```
virtual int FindClose(ZafFileInfoStruct &fileInfo);
```

`FindClose()` finalizes a find operation begun with `FindFirst()`. *fileInfo* is the same `ZafFileInfoStruct` object used by `FindFirst()` and `FindNext()`, and memory allocated in *fileInfo* is deleted by `FindClose()`. `FindClose()` returns 0 if the operation was successful; otherwise it returns -1.

FindFirst

```
virtual int FindFirst(ZafFileInfoStruct &fileInfo, const
    ZafIChar *searchName, ZafStringID stringID =
    ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0);
```

FindFirst() initializes a find operation. FindFirst() searches for a ZafStorageFile with the search pattern specified by *searchName*, or with the string identification constant specified by *stringID*, or with the numeric identification constant specified by *numberID*. *searchName* may contain wildcards. For example, "?" means any single character, and "*" means any string of characters. *fileInfo* is a ZafFileInfoStruct object allocated by the programmer, and *fileInfo* is initialized by FindFirst(). FindFirst() returns 0 if a ZafStorageFile was found; otherwise it returns -1.

FindNext

```
virtual int FindNext(ZafFileInfoStruct &fileInfo, const
    ZafIChar *searchName, ZafStringID stringID =
    ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0);
```

FindNext() continues a find operation initiated by FindFirst(). FindNext() searches for the next ZafStorageFile with the same values for *searchName*, *stringID*, or *numberID* as were specified in the call to FindFirst(). *fileInfo* is the same ZafFileInfoStruct object passed to FindFirst(). FindNext() returns 0 if a ZafStorageFile was found; otherwise it returns -1.

GetCWD

```
virtual int GetCWD(ZafIChar *pathName, int pathLength);
```

GetCWD() copies the path name of the current storage directory (without a terminating path separating character) into the buffer specified by *pathName*. *pathName* must be allocated by the programmer, and *pathLength* specifies the number of ZafIChar characters in *pathName*. GetCWD() returns 0 on success, and -1 on failure. Error() is also set to an appropriate value.

MkDir

```
virtual int MkDir(const ZafIChar *pathName, ZafStringID
    stringID = ZAF_NULLP(ZafIChar), ZafNumberID numberID =
    0);
```

MkDir() creates a new storage directory. *pathName* specifies the path name of the directory to be created. *stringID* specifies the string identification constant associated with the directory and *numberID* specifies the numeric identification constant associated with the directory. MkDir() returns 0 on success, and -1 on failure. Error() is also set to an appropriate value.

Open `virtual ZafFile *Open(const ZafIChar *fileName, const
 ZafFileMode mode, ZafStringID stringID =
 ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0);`

`Open()` opens the `ZafStorageFile` specified by *fileName*. *mode* specifies the mode the `ZafStorageFile` is opened with (see the `ZafFile` constructor for more information). *stringID* specifies the string identification constant associated with the `ZafStorageFile` and *numberID* specifies the numeric identification constant associated with the `ZafStorageFile`. A pointer to the `ZafStorageFile` opened is returned.

OpenCreate `bool OpenCreate(void) const;`

If `OpenCreate()` is true, the `ZafStorage` object is opened if it exists, or it is created if it doesn't exist.

ReadOnly `bool ReadOnly(void) const;`

If `ReadOnly()` is true, the `ZafStorage` object and its `ZafStorageFile` objects cannot be written.

ReadWrite `bool ReadWrite(void) const;`

If `ReadWrite()` is true, the `ZafStorage` object and its `ZafStorageFile` objects may be read and written.

Remove `virtual int Remove(const ZafIChar *fileName);`

`Remove()` deletes the `ZafStorageFile` specified by *name*. `Remove()` returns 0 on success, and -1 on failure. `Error()` is also set to an appropriate value.

Rename `virtual int Rename(const ZafIChar *oldName, const
 ZafIChar *newName, ZafStringID stringID =
 ZAF_NULLP(ZafIChar), ZafNumberID numberID = 0);`

`Rename()` renames the storage file or directory specified by *oldName* to *newName*. A non-null *stringID* specifies the string identification constant to be copied to the file or directory. A non-zero *numberID* specifies the numeric identification constant to be copied to the file or directory. If *oldName* and *newName* are the same, *stringID* and *numberID* are still copied to the file or directory (if not null and 0, respectively). `Rename()` returns 0 on success, and -1 on failure. `Error()` is also set to an appropriate value.

RmDir `virtual int RmDir(const ZafIChar *pathName, bool
 deleteContents = false);`

RmDir() deletes a storage directory. *pathName* specifies the path name of the directory to be deleted. If *deleteContents* is false, **RmDir()** will only delete the directory if it is empty; otherwise, **RmDir()** will delete the contents of the directory along with the directory itself. **RmDir()** returns 0 on success, and -1 on failure. **Error()** is also set to an appropriate value.

Save `int Save(int backups = 0);`

Save() causes the ZafStorage object's **Changed()** ZafStorageFile objects to be saved permanently. The ZafStorage object keeps its name for the operation. *backups* specifies the number of backups the ZafStorage object keeps of itself. **Save()** returns 0 on success, and -1 on failure. **Error()** is also set to an appropriate value.

SaveAs `int SaveAs(const ZafIChar *newName, int backups = 0);`

SaveAs() changes its name to *newName*, then calls **Save()**, passing it *backups*. **SaveAs()** returns whatever **Save()** returns.

Version `ZafVersion Version(void) const;
ZafVersion SetVersion(ZafVersion version);`

Version() returns the version of the ZafStorage object. The high byte contains the major version number, and the low byte contains the minor version number. **SetVersion()** may be called to change it, but **SetVersion()** is an advanced routine and should normally not be called by the programmer. The definition of **ZafVersion** follows:

```
typedef ZafInt16 ZafVersion;
```

The following code snippet shows how to get the version of a ZafStorage object:

```
// Make sure the version of the ZafStorage is at least 5.0.
ZafVersion version = storage->Version();
if ((version >> 8) < 5)
    break;
```

ZafStorageFile

Changed	ReadData	WriteData
Data	Seek	
Length	Tell	

Inheritance	ZafStorageFile : ZafFile : ZafElement
Declaration	#include <z_store.hpp>
Description	<p>ZafStorage and ZafStorageFile provide support for persistence in ZAF. Any information may be portably stored with ZafStorage, and most ZAF classes provide built-in support for persistence through ZafStorage. ZafStorageFile is used exclusively with ZafStorage for persistence support. See ZafStorage for more information.</p>
Constructor	<p>ZafStorageFile(ZafFileMode mode, char *initData = ZAF_NULLP(char), int initDataSize = 0);</p> <p>This constructor is protected, and is only called from ZafStorage::Open(). It initializes the members associated with a ZafStorageFile object, and chains to the ZafFile and ZafElement constructors. <i>mode</i> specifies the mode in which the storage file is opened (see the ZafFile constructor for more information). <i>initData</i> specifies the data that the ZafStorageFile object is initialized with, and <i>initDataSize</i> specifies the number of bytes in <i>initData</i>.</p>
Destructor	<p>virtual ~ZafStorageFile(void);</p> <p>This virtual destructor is protected, and is only called from ZafStorage::Close(). It frees the memory associated with an instantiated ZafStorageFile object and chains to the ZafFile and ZafElement destructors.</p>
Members	<p>bool Changed(void);</p> <p><i>Changed</i> returns true if the contents of the storage file have changed since it was opened; otherwise it returns false.</p>
<i>Data</i>	<p>const char *Data(void);</p> <p><i>Data</i>() returns a pointer the internal data buffer for the storage file. This is an advanced function, and should not be called by the programmer.</p>
<i>Length</i>	<p>virtual ZafOffset Length(void) const;</p> <p><i>Length</i>() returns the length of the storage file in bytes, or it returns -1 if an error occurs.</p>

ReadData

```
virtual int ReadData(void *buffer, int size);
```

ReadData() reads data from the storage file. The data is read into *buffer*, which must have already been allocated by the programmer, and *size* specifies the number of bytes to read. On success, **ReadData()** returns the number of bytes read into *buffer*; otherwise zero is returned. **Error()** is also set to an appropriate value.

Seek

```
virtual int Seek(ZafOffset offset, ZafSeek location);
```

Seek() moves the file pointer of the storage file. *location* specifies the position in the file from where *offset* specifies. Possible values of *location* and what they mean follow:

ZafSeek	ZafOffset
ZAF_SEEK_START	The offset is measured from the beginning of the file toward the end of the file
ZAF_SEEK_CURRENT	A positive offset is measured from the current file pointer, and a negative offset is measured toward the beginning of the file.
ZAF_SEEK_END	The offset is measured from the end of the file toward the beginning of the file

Seek() returns 0 if successful, and -1 if an error occurs (in which case **Error()** is also set appropriately).

Tell

```
virtual ZafOffset Tell(void) const;
```

Tell() returns the current offset of the file pointer in the storage file, or it returns -1 if an error occurs.

WriteData

```
virtual int WriteData(const void *buffer, int size);
```

WriteData() writes data to a storage file. *buffer* is a pointer to the data to be written, and *size* specifies the number of bytes to write. On success, **WriteData()** returns the number of bytes written; otherwise zero is returned. **Error()** is also set to an appropriate value.

ZafString

AllowInvalid	LowerCase	SetSelected
AutoClear	OutputFormatData	StringData
CursorOffset	OutputFormatText	Text
Decrement	Password	Unanswered
DefaultValidateFunction	RangeData	UpperCase
Event	RangeText	Validate
HzJustify	SetInputFormat	VariableName
Increment	SetOutputFormat	ViewOnly
InputFormatData	SetRange	
Invalid	ReportInvalid	

Inheritance ZafString : ZafWindowObject : ZafElement

Declaration #include <z_str1.hpp>

Description 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.

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

AllowInvalid()	false
AutoClear()	true
HzJustify()	ZAF_HZ_LEFT
InputFormatData()	null
Invalid()	false
LowerCase()	false
OutputFormatData()	null
Password()	false
RangeData()	null

Member Initializations

ReportInvalid()	true
StringData()	null
Unanswered()	false
UpperCase()	false
VariableName()	false
ViewOnly()	false

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. *text* is the string that initially appears in the new ZafString object, and *maxLength* is passed to the ZafStringData constructor, the maximum number of characters the user may type into the field (see the [ZafStringData](#) constructor for more information). If you pass -1 in for the *maxLength* parameter, the number of characters the user may type is not restricted.

```
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 ZafString-

Data 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 [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 information. 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.

AllowInvalid

```
bool AllowInvalid(void) const;
virtual bool SetAllowInvalid(bool allowInvalid);
```

AllowInvalid() is used in classes derived from ZafString that require validation, such as ZafBignum and ZafTime. If AllowInvalid() is false when the data is validated during a focus change, the field will not be allowed to lose focus (such as when the end user uses the <Tab> key). The default value of this attribute is false, but the user may call SetAllowInvalid() to change it.

AutoClear

```
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.

CursorOffset

```
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);
```

Decrement

```
virtual ZafError Decrement(ZafData *data, bool wrapData =
    false);
```

Decrement() does nothing by default in the ZafString class, but is overloaded by other classes such as ZafDate and ZafInteger to decrement values. The corresponding data object is decremented via the -= operator by the amount specified in *data*. *wrapData* specifies whether or not to allow wrapping at the range minimum. If *wrapData* is true and the new value would be less than the allow-

able range minimum, the new value will be the range maximum; otherwise the new value will be the range minimum. The visual object is updated accordingly. If *data* is not of the correct class, nothing happens. For example, *data* for a *ZafDate* object must be a *ZafDateData* object.

DefaultValidateFunction

```
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 *Validate()*, 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.

Event

```
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

Event	Description
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

HzJustify

```
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.

Increment

```
virtual ZafError Increment(ZafData *data, bool wrapData =
    false);
```

Increment() does nothing by default in the ZafString class, but is overloaded by other classes such as ZafDate and ZafInteger to increment values. The corresponding data object is incremented via the += operator by the amount specified in *data*. *wrapData* specifies whether or not to allow wrapping at the range maximum. If *wrapData* is true and the new value would be greater than the allowable range maximum, the new value will be the range minimum; otherwise the new value will be the range maximum. The visual object is updated accordingly. If *data* is not of the correct class, nothing happens. For example, *data* for a ZafDate object must be a ZafDateData object.

InputFormatData
SetInputFormat

```
ZafStringData *InputFormatData(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. `SetInputFormatData()` will delete the previous `InputFormatData()` object if it is `Destroyable()` and no other object uses it.

`InputFormatData()` returns the string data object used for the input formatting and `InputFormatText()` returns the text contained in the string data object. `SetInputFormat()` and `SetInputFormatData()` generally return `ZAF_ERROR_NONE`. For more information on format string arguments and what they mean, see [ZafStringData](#) and derived `ZafStrings` ([ZafBignum](#), [ZafInteger](#), [ZafReal](#), [ZafDate](#), [ZafTime](#), and [ZafUTime](#)). 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);
```

Invalid

```
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");
```

LowerCase

```
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 `SetLowerCase()` 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.

*OutputFormatData**OutputFormatText**SetOutputFormat*

```
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. `SetOutputFormatData()` will delete the previous `OutputFormatData()` object if it is `Destroyable()` and no other object uses it.

Derived classes such as `ZafDate` and `ZafTime` may concatenate several format strings separated with '\n' characters in `OutputFormatText()`. For example, a sample `OutputFormatText()` for `ZafDate` may be:

```
"%m/%d/%y\n%d/%m/%y\n%m-%d-%y\n%d-%m-%y".
```

For more information on format string arguments and what they mean, see [ZafStringData](#) and derived `ZafStrings` ([ZafBignum](#), [ZafInteger](#), [ZafReal](#), [ZafDate](#), [ZafTime](#), and [ZafUTime](#)). Set `SetInputFormat()` for sample code.

Format Argument	Substitution
%c	ZafIChar character
%s	ZafIChar array (string)

Password

```
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.

*RangeData**RangeText**SetRange*

```
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. SetRangeData() will delete the previous RangeData() object if it is Destroyable() and no other object uses it.

Sample usage follows:

```
// Get the range two different ways.
const ZafStringData *rangeFormat = object->RangeData();
const ZafIChar *rangeText = object->RangeText();
...
// Set the range two different ways.
ZafInteger *myInt = new ZafInteger(10);
ZafReal *myReal = new ZafReal(10.0);
myInt->SetRange("1..100");
ZafStringData *realRange = new ZafStringData("1.0..100.0");
myReal->SetRangeData(realRange);
```

ReportInvalid

```
bool ReportInvalid(void) const;
virtual bool SetReportInvalid(bool reportInvalid);
```

ReportInvalid() is used in classes derived from **ZafString** that require validation, such as **ZafBigNum** and **ZafTime**. If **ReportInvalid()** is true when the data is validated in **Validate()**, an error window is presented to the end user. The default value of this attribute is true, but the user may call **SetReportInvalid()** to change it.

SetSelected

```
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.

StringData

```
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](#). **SetStringData()** will delete the previous **StringData()** object if it is **Destroyable()** and no other object uses it.

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);
```

Text

```
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");
```

Unanswered

```
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()`.

UpperCase

```
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.

Validate

```
virtual ZafError Validate(bool processError = true);
```

`Validate()` does nothing by default in the `ZafString` class, but is overloaded by other classes such as `ZafDate` and `ZafInteger` that require validation. `Validate()` is called by `DefaultValidateFunction()`, but may be called in user functions for custom validation (see `ZafWindowObject::UserFunction()`). If *processError* is true and an error occurs, `Validate()` causes `ZafErrorSystem` to report the error; otherwise `Validate()` simply returns the appropriate error code. If `Validate()` is called as a result of a focus change (such as `N_CURRENT` and

N_NON_CURRENT messages), *processError* must be false so that an error window does not cause an untimely focus change to occur.

The return value for `Validate()` 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.

VariableName

```
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.

ViewOnly

```
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	Remove	operator +
Char	Replace	operator +=
Clear	StaticData	operator <
DynamicText	SetOSText	operator <=
DynamicOSText	SetOSWText	operator =
DynamicOSWText	Text	operator ==
Compare	ZafChar	operator >
Insert	operator -	operator >=
Length	operator -=	operator []
MaxLength	operator !=	

Inheritance ZafStringData : ZafFormatData : ZafData :
 ZafNotification, ZafElement

Declaration #include <z_str.hpp>

Description 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.

Multi-line text stored in a ZafStringData object for use by the native environment (as when presented visually to the end user) contains carriage return/line feed ("r\n") pairs for portability. When passing multi-line text to a ZafStringData object via the constructor or the SetText() methods for use by the native environment, "r\n" pairs should be used. Any multi-line text passed to the native environment by ZAF is converted internally by the libraries if needed, and any multi-line text retrieved from the native environment is converted internally by the libraries if needed to maintain "r\n" pairs. For example, Unix strings utilize only the 'n' character to denote the end of a line, and Macintosh strings utilize only the 'r' character to denote the end of a line. ZAF converts these strings when necessary.

All ZafData objects may make use of printf-style formatting and parsing arguments during string operations. Refer to user interface object reference for descriptions of the format arguments available for each data type. (For example, see [ZafString](#) for format arguments available when displaying a ZafStringData.)

Numerous overloaded operators are available to facilitate manipulation of ZafStringData objects.

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)
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 ©);

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

Append

```
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!!!
```

Char

```
ZafIChar Char(int offset) const;
virtual ZafError SetChar(int offset, ZafIChar value, bool
ignoreNotification = true);
```

`Char()` returns the character at the zero-based index *offset*.

`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.

Clear

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

Clear() sets the Text() portion of the specified object to the empty string ("").

Compare

```
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.
```

DynamicText

DynamicOSText

DynamicOSWText

```
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). These functions return the result in different formats.

DynamicOSText() and DynamicOSWText() convert "\r\n" pairs in multi-line text to what the native environment expects if needed, so the return value may be passed directly to the native environment. For example, on Unix the "\r\n" pairs are converted to '\n' and on Macintosh the "\r\n" pairs are converted to '\r'.

DynamicOSWText() is only available in Unicode mode. It converts the data to a wide character array in the native OS codeset.

Insert

```
virtual ZafStringData &Insert(int offset, const ZafIChar
    *text, int length = -1);
```

Insert() will insert the string *text* into a ZafStringData object at the zero-based index *offset*. *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
```

Length

```
int Length(void) const;
```

Length() returns the number of characters in the ZafStringData regardless of MaxLength().

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.

Remove

```
virtual ZafString &Remove(int offset, int size);
virtual ZafString &Remove(const ZafIChar *text);
```

These functions remove text from the ZafStringData object. The first method removes *size* characters beginning with the character at zero-based *offset*. 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
```

Replace

```
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
```

SetOSText

SetOSWText

```
virtual ZafError SetOSText(const char *text);
ZafError SetOSWText(const wchar_t *text);
```

`SetOSText()` converts *text* from the internal representation used by the native OS to the portable representation used by ZAF. These functions are used internally by ZAF and will generally not be called by the programmer.

`SetOSText()` and `SetOSWText()` convert native environment end-of-line representations in multi-line text to `"\r\n"` pairs if needed, so the new value of the

ZafStringData object is portable. For example, on Unix the '\n' characters are converted to "\r\n" pairs and on Macintosh the '\r' characters are converted to "\r\n" pairs.

SetOSWText() is only available in Unicode mode.

StaticData

```
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.

Text

```
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
```

ZafIChar

```
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();
```

operator -

```
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 see [Remove\(\)](#).

operator -=

```
ZafStringData &operator-=(const ZafStringData &value);
ZafStringData &operator-=(const ZafIChar *value);
```

These operators provide shortcuts to using the Remove methods. For more information see [Remove\(\)](#).

operator !=

```
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 see [Compare\(\)](#).

operator +

```
ZafStringData operator+(const ZafStringData &string1,
    const ZafStringData &string2);
ZafStringData operator+(const ZafStringData &string,
    const ZafIChar *value);
ZafStringData operator+(const ZafStringData &string,
    const ZafIChar value);
ZafStringData operator+(const ZafIChar *value, const
    ZafStringData &string);
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 (or character) with the second string (or character) appended to it. For more information see [Append\(\)](#).

operator +=

```
ZafStringData &operator+=(const ZafStringData &value);  
ZafStringData &operator+=(const ZafIChar value);  
ZafStringData &operator+=(const ZafIChar *value);
```

These operators provide shortcuts to using the `Append()` methods. For more information see [Append\(\)](#). The second of the three operators provides support for adding a single `ZafIChar` to the end of the string data.

operator <

```
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 see [Compare\(\)](#).

operator <=

```
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 see [Compare\(\)](#).

operator =

```
ZafStringData &operator=(const ZafStringData &value);  
ZafStringData &operator=(const ZafIChar *value);  
ZafStringData &operator=(const ZafIChar value);
```

These operators provide shortcuts to using the `SetText()` methods. For more information see [SetText\(\)](#).

Note that these operators are different than other assignment operators in that they return const values.

```
operator ==      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 see [Compare\(\)](#).

```
operator >       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 see [Compare\(\)](#).

```
operator >=     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 see [Compare\(\)](#).

```
operator []     ZafIChar operator[](int offset) const;
ZafIChar &operator[](int offset);
```

These operators provide a method of accessing individual characters in the string contained in the string data object. *offset* is the zero-based index of the character to access. 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!
```

ZafSystemButton

Add	Get	Subtract
Count	Index	SystemButtonType
Current	Last	operator +
Destroy	menu	operator -
First	Sort	

Inheritance

ZafSystemButton : ZafButton : ZafWindowObject : ZafElement

Declaration

#include <z_sys.hpp>

Description

The ZafSystemButton object may only be added to a ZafWindow. The ZafSystemButton is the system button decoration on a ZafWindow, and is generally drawn by the environment. The ZafSystemButton object is used to close the parent window and on environments that support it, to present a system menu with other functions available such as move and size.

Constructors

All ZafSystemButton constructors initialize the member variables associated with an instantiated ZafSystemButton object. The default values set by the ZafSystemButton 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 ZafSystemButton. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafSystemButton	
menu	ZafPopUpMenu(0, 0)
menu.parent	this
menu.Temporary()	true
menu.Destroyable()	false
menu.NumberID()	ZAF_NUMID_SYSTEM_BUTTON_MENU
menu.StringID()	"ZAF_NUMID_SYSTEM_BUTTON_MENU"
SystemButtonType()	ZAF_NATIVE_SYSTEM_BUTTON
ZafButton	
AllowDefault()	false [†]
AllowToggling()	false [†]
AutoRepeatSelection()	false [†]
AutoSize()	true [†]

Member Initializations

ButtonType()	ZAF_3D_BUTTON [†]
Depth()	1 [†]
HotKeyChar()	0 [†]
HotKeyIndex()	-1 [†]
HzJustify()	ZAF_HZ_CENTER [†]
SelectOnDoubleClick()	false [†]
SelectOnDownClick()	false [†]
SendMessageText()	null [†]
SendMessageWhenSelected()	false [†]
Value()	0 [†]
VtJustify()	ZAF_VT_CENTER [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	false [†]
CopyDraggable()	false [†]
Disabled()	false [†]
HelpContext()	null [†]
HelpObjectTip()	null [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
Noncurrent()	false [†]
ParentDrawBorder()	false [†]
ParentDrawFocus()	false [†]
ParentPalette()	false [†]
QuickTip()	null [†]
RegionType()	ZAF_AVAILABLE_REGION [†]
Selected()	false [†]
SupportObject()	true [†]
SystemObject()	false
UserFunction()	null [†]

ZafElement

ClassID()	ID_ZAF_SYSTEM_BUTTON
ClassName()	"ZafSystemButton"
NumberID()	ZAF_NUMID_SYSTEM
StringID()	"ZAF_NUMID_SYSTEM"

```
ZafSystemButton(ZafSystemButtonType systemButtonType =
    ZAF_NATIVE_SYSTEM_BUTTON);
```

This constructor is useful in straight-code situations to create a `ZafSystemButton` object. *systemButtonType* specifies the type of system button to be created. See [SystemButtonType\(\)](#) for more information.

```
ZafSystemButton(const ZafSystemButton &copy);
```

The copy constructor creates a new `ZafSystemButton` object and initializes its data from *copy*.

```
ZafSystemButton(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 system button follows:

```
// Create a sample window with a native system button.
ZafWindow *window = new ZafWindow(0, 0, 40, 10);
ZafSystemButton *sys = new ZafSystemButton;
window->Add(sys);
```

Destructor

```
virtual ~ZafSystemButton(void);
```

The destructor is used to free the memory associated with a `ZafSystemButton` object. It chains to the `ZafButton`, `ZafWindowObject`, and `ZafElement` destructors. Generally, the programmer will not directly destroy a `ZafSystemButton` 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.

Add

```
virtual ZafWindowObject *Add(ZafWindowObject *object,
    ZafWindowObject *position =
    ZAF_NULLP(ZafWindowObject));
```

operator +

```
ZafSystemButton &operator+(ZafWindowObject *object);
```

This function and operator are used to create system menu items on environments that support them by adding ZafPopUpItem objects to the ZafSystemButton. 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 ZafSystemButton object's system 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.

Count

```
int Count(void);
```

Count() returns the number of ZafPopUpItem objects in the system menu, or 0 if there is no system menu. See ZafList::Count() for more information.

Current

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

Current() returns the current ZafPopUpItem object in the system 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.

Destroy

```
virtual void Destroy(void);
```

Destroy() causes all the ZafPopUpItem objects in the system menu to be destroyed, if there is one. This is useful if a system menu is to be recreated from scratch. See ZafList::Destroy() for more information.

First

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

The First() function returns the first ZafPopUpItem object in the system menu, if any. See ZafList::First() for more information.

Get

```
ZafWindowObject *Get(int index);
```

Get() returns the ZafPopUpItem object at position *index* in the system menu, if there is one. *index* is zero-based, meaning the first ZafPopUpItem object in the system menu is at position 0. See ZafList::Get() for more information.

<i>Index</i>	<pre>int Index(ZafWindowObject const *element);</pre> <p>If there is a system menu, the Index() function returns the zero-based index of the ZafPopUpItem object element in the system menu. If there is no system menu or element is not found in the system menu, Index() returns -1. See ZafList::Index() for more information.</p>
<i>Last</i>	<pre>ZafWindowObject *Last(void) const;</pre> <p>Last() returns the last ZafPopUpItem object in the system menu, if any. See ZafList::Last() for more information.</p>
<i>menu</i>	<pre>ZafPopupMenu menu;</pre> <p>The menu member, used internally by the ZAF libraries to maintain the ZafPopUpItem objects added to the ZafSystemButton object (forming the system menu), should normally not be accessed by the programmer. See ZafPopupMenu for more information.</p>
<i>Sort</i>	<pre>virtual void Sort(void);</pre> <p>Sort() causes the ZafPopUpItem objects in the system menu (if there is one) to be sorted according to the function returned by CompareFunction(). See ZafList::CompareFunction() for more information. Since some environments maintain the system menu, Sort() may have no effect.</p>
<i>Subtract</i>	<pre>virtual ZafWindowObject *Subtract(ZafWindowObject</pre>
<i>operator -</i>	<pre> *object);</pre> <pre>ZafSystemButton &operator-(ZafWindowObject *object);</pre> <p>This function and operator are used for subtracting (or removing) ZafPopUpItem objects from the ZafSystemButton object's system menu. The functionality is provided by the ZafPopupMenu class through the menu member. <i>object</i> is a pointer to the ZafPopUpItem object to be subtracted. See ZafWindow::Subtract() for more information.</p>
<i>SystemButtonType</i>	<pre>ZafSystemButtonType SystemButtonType(void) const;</pre> <pre>virtual ZafSystemButtonType</pre> <pre> SetSystemButtonType(ZafSystemButtonType</pre> <pre> systemButtonType);</pre> <p>SystemButtonType() specifies the type of system button a ZafSystemButton object is. The default value of this attribute is</p>

ZAF_NATIVE_SYSTEM_BUTTON, but the user may call SetSystemButtonType() to change it. Here are the possible system button types:

IconType()	Description
ZAF_EMPTY_SYSTEM_BUTTON	Creates a system button object with no menu items in it so that the system menu may be custom-built (or remain empty)
ZAF_NATIVE_SYSTEM_BUTTON	Creates a native system button object with the default menu if supported on the environment (may look different from environment to environment)

ZafTable

BottomOffset	HeaderWidth	SeekRandomRecord
CurrentOffset	InsertRecord	SetColumnText
DeleteRecord	MaxOffset	SetCoordinateType
Flush	ReadRecord	SetReadFunction
FocusOffset	Record	SetWriteFunction
Grid	Repopulate	Tell
HeaderBackgroundColor	RowHeight	TopOffset
HeaderHeight	SeekNextRecord	VirtualRecord
HeaderTextColor	SeekPreviousRecord	WriteRecord

Inheritance

ZafTable : ZafWindow : (ZafWindowObject : ZafElement),
ZafList

Declaration

#include <z_table.hpp>

Description

ZafTable is designed to display and modify data in a tabular format. It is optimized for interaction with a database, but can be used to interact with data in a variety of formats. It is not, however, intended to be used as a spreadsheet.

Data in a ZafTable is displayed in a familiar row/column format. Each row in a table represents one record, while each column represents a field within each record. If desired, ZafTableHeader objects are automatically provided to display row and column labels.

The columns (or fields) of a table are defined by adding objects to the table. For example, if a ZafString object is added to the table, each record in the table will contain a ZafString object. If the ZafString is the first object in the table, then the first column of the table will consist of ZafString objects, each ZafString object being the first field of a record.

Each row of a table is identical, each being one record, except for the data displayed in the fields. The number of rows is determined by MaxOffset() which is modified by using SetMaxOffset(), InsertRecord(), and DeleteRecord(). SetMaxOffset() should be called in the constructor for the derived table class to specify the number of records in the table. See the code sample below.

ZafTable defines two functions that must be overridden by a derived class: ReadRecord() and WriteRecord(). ReadRecord() is used to obtain data and make it available to the fields of a ZafTableRecord object. WriteRecord() is used to take data from the fields of a ZafTableRecord object and update the external data source. (See [ReadRecord\(\)](#) and [WriteRecord\(\)](#) for more information.)

In addition to `ReadRecord()` and `WriteRecord()`, a class derived from `ZafTable` can optionally overload three other functions to optimize performance: [SeekNextRecord\(\)](#), [SeekPreviousRecord\(\)](#), and [SeekRandomRecord\(\)](#). Whenever possible, `ZafTable` reads and writes records in sequential order using `SeekNextRecord()` or `SeekPreviousRecord()`. Database implementations that are optimized by sequential access should overload these functions. (See `SeekNextRecord()`, `SeekPreviousRecord()`, and `SeekRandomRecord()` for more information.)

This class is not available in the Personal or Registered versions of ZAF, but is included with the Professional version.

Constructors

All `ZafTable` constructors initialize the member variables associated with an instantiated `ZafTable` object. The default values set by the `ZafTable` 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 `ZafTable`. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafTable

<code>CurrentOffset()</code>	-1
<code>Grid()</code>	true
<code>HeaderHeight()</code>	user-supplied parameter
<code>HeaderWidth()</code>	user-supplied parameter
<code>MaxOffset()</code>	-1
<code>readFunction</code>	<code>ZafTable::DefaultReadFunction</code>
<code>RowHeight()</code>	user-supplied parameter
<code>VirtualRecord()</code>	null
<code>writeFunction</code>	<code>ZafTable::DefaultWriteFunction</code>

ZafWindow

<code>Destroyable()</code>	false [†]
<code>Locked()</code>	false [†]
<code>Maximized()</code>	false [†]
<code>Minimized()</code>	false [†]
<code>Modal()</code>	false [†]
<code>Moveable()</code>	false [†]
<code>NormalHotKeys()</code>	false [†]
<code>SelectionType()</code>	<code>ZAF_MULTIPLE_SELECTION</code>
<code>Sizeable()</code>	false [†]
<code>Temporary()</code>	false [†]

ZafWindowObject

Member Initializations

AcceptDrop()	false [†]
Bordered()	true
OSDraw()	false

ZafElement

ClassID()	ID_ZAF_TABLE
ClassName()	"ZafTable"

```
ZafTable(int left, int top, int width, int height, int
          headerWidth, int headerHeight, int rowHeight = 1);
```

This constructor is useful in straight-code situations. `ZafTableHeader` objects are automatically provided for displaying labels for the rows and columns of a table. *left*, *top*, *width* and *height* specify the position and size of the table on its parent. *headerWidth* and *headerHeight* specify the size of the row header and column header respectively. Zero values for *headerWidth* and/or *headerHeight* cause the corresponding headers not to be displayed. *rowHeight* specifies the height of each row of the table. All values are specified in cell coordinates by default, but may be specified using another coordinate system if desired.

```
ZafTable(const ZafTable &copy);
```

The copy constructor creates a new `ZafTable` object and initializes its data from *copy*.

```
ZafTable(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 database-linked table follows:

```
// Declare a generic database.
extern SimpleDatabase *database;

// Derive a user table class.
class SampleTable : public ZafTable
{
```

```
public:
    SampleTable(void);
    virtual ZafError ReadRecord(ZafTableRecord &record,
        ZafWindowObject *row);
    virtual ZafError WriteRecord(ZafTableRecord &record,
        ZafWindowObject *row);
};

SampleTable::SampleTable(void) :
    ZafTable(5, 10, 400, 100, 25, 15, 15)
{
    // Use pixel coordinate type for the table, headers, and
    // records. SetCoordinateType() on the table affects them all.
    SetCoordinateType(ZAF_PIXEL);

    // Add a vertical scroll bar.
    Add(new ZafScrollBar(0, 0, 0, 0));

    // Add a string column for "Name".
    Add(new ZafString(0, 0, 20, ZAF_NULLP(ZafIChar), 32), "Name");

    // Add a formatted string column for "SSN".
    Add(new ZafFormattedString(20, 0, 20, ZAF_NULLP(ZafIChar), new
        ZafStringData("NNNLNNLNNNN"), new ZafStringData("...-...-
        ....")), "SSN");

    // Add a string column for "Grade".
    Add(new ZafString(40, 0, 10, ZAF_NULLP(ZafIChar), 32),
        "Grade");

    // Set the number of records.
    // This should be in the constructor.
    SetMaxOffset(database->Records() - 1);
}

// ReadRecord() must be overloaded in the derived class.
ZafError SampleTable::ReadRecord(ZafTableRecord &record,
    ZafWindowObject *row)
{
    // Read the fields in the record.
    ZafIChar name[32], ssn[16], grade[8];
    database->SeekRandomRecord(record.Offset(), name, ssn, grade);
    record(0)->SetText(name);
    record(1)->SetText(ssn);
    record(2)->SetText(grade);

    // Set the row header.
    ZafIChar rowText[16];
    sprintf(rowText, "%d", record.Offset() + 1);
```

```

        row->SetText(rowText);
        return (ZAF_ERROR_NONE);
    }

    // WriteRecord() must be overloaded in the derived class.
    ZafError SampleTable::WriteRecord(ZafTableRecord &record,
        ZafWindowObject *)
    {
        // Write the fields in the record.
        database->ModifyRecord(record.Offset(), record(0)->Text(),
            record(1)->Text(), record(2)->Text());
        return (ZAF_ERROR_NONE);
    }

    int ZafApplication::Main(void)
    {
        // Ensure main() is linked properly.
        LinkMain();

        // Initialize the database.
        database = new SimpleDatabase;

        // Create the table and add it to a window.
        ZafWindow *window = new ZafWindow(0, 0, 60, 15);
        window->AddGenericObjects(new ZafStringData("Table Window"));
        window->Add(new SampleTable);
        zafWindowManager->Add(window);

        // Get the user input and return success.
        Control();
        return (0);
    }

```

Destructor

```
virtual ~ZafTable(void);
```

The destructor is used to free the memory associated with a ZafTable object. It chains to the ZafWindow, ZafWindowObject, ZafList, and ZafElement destructors. Generally, the programmer will not directly destroy a ZafTable 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.

BottomOffset

```
ZafOffset BottomOffset(void) const;  
virtual ZafOffset SetBottomOffset(ZafOffset  
    currentOffset);
```

BottomOffset() specifies the zero-based index of the record currently at the bottom of the table's viewable area. **SetBottomOffset()** may be called to change it. **SetBottomOffset()** causes **ReadRecord()** to get any necessary data, but does not change which record has focus. See [TopOffset\(\)](#).

SetColumnText

```
ZafError SetColumnText(int columnNumber, const ZafIChar  
    *text);
```

SetColumnText() sets the text on a column header. *columnNumber* is the zero-based index of the column number, and *text* is the text copied into the column header.

SetCoordinateType

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

This overloaded function sets the coordinate type for the table headers and **VirtualRecord()**, as well as for the **ZafTable** object itself. See **ZafWindowObject::SetCoordinateType()** and the code snippet for the constructors for more information.

CurrentOffset**Tell**

```
ZafOffset CurrentOffset(void) const;  
virtual ZafOffset Tell(void);  
virtual ZafOffset SetCurrentOffset(ZafOffset  
    currentOffset);
```

CurrentOffset() and **Tell()** return the zero-based offset of the record that was most recently read into the table. **CurrentOffset()** and **Tell()** are used internally by the ZAF libraries to optimize database seeks, and may be modified by the derived **ZafTable** class by overriding **SeekNextRecord()**, **SeekPreviousRecord()**, and **SeekRandomRecord()** as described in this chapter.

DeleteRecord

```
virtual ZafError DeleteRecord(ZafOffset deleteOffset);
```


DeleteRecord() causes the ZafTableRecord object at zero-based offset *deleteOffset* to be deleted from the ZafTable. The corresponding database record, if any, is not deleted. DeleteRecord() generally returns ZAF_ERROR_NONE.

Flush

```
ZafError Flush(ZafOffset offset = -1);
```

Flush() causes the table record at zero-based index *offset* to be written to the database using WriteRecord(). If *offset* specifies a record that is not in the viewable area of the database, nothing happens. If *offset* is -1, all records in the viewable area of the table are written to the database. If the operation was successful, ZAF_ERROR_NONE is returned; otherwise ZAF_ERROR_INVALID is returned.

FocusOffset

```
ZafOffset FocusOffset(void) const;
virtual ZafOffset SetFocusOffset(ZafOffset focusOffset);
```

FocusOffset() specifies the zero-based index of the record in the table that currently has focus. SetFocusOffset() may be called to set the focus to the record with zero-based index *focusOffset*. If the record gaining focus is not in the viewable area of the table, it is scrolled into view. The table is not scrolled if the *focusOffset* record is in view.

Grid

```
bool Grid(void) const;
virtual bool SetGrid(bool grid);
```

If Grid() is true, grid lines are drawn between each field in the record, and between each record. This attribute defaults to true, but the programmer may change it by calling SetGrid().

HeaderBackgroundColor

```
ZafLogicalColor HeaderBackgroundColor(ZafLogicalColor
    *color = ZAF_NULLP(ZafLogicalColor), ZafLogicalColor
    *mono = ZAF_NULLP(ZafLogicalColor));
virtual ZafLogicalColor
    SetHeaderBackgroundColor(ZafLogicalColor color,
    ZafLogicalColor mono = ZAF_MONO_NULL);
```

HeaderBackgroundColor() is the background color used for the ZafTable-Header objects in the ZafTable. SetHeaderBackgroundColor() may be called to change the background color used by all the ZafTableHeader objects attached to the ZafTable. See ZafWindowObject::BackgroundColor() for more information.

HeaderHeight

```
int HeaderHeight(void) const;  
virtual int SetHeaderHeight(int headerHeight);
```

HeaderHeight() specifies the height of the column **ZafTableHeader** object in the same coordinate system as **Region()**. A zero value indicates the absence of a column **ZafTableHeader** object. **SetHeaderHeight()** should only be called from the constructor for the derived **ZafTable** class.

HeaderTextColor

```
ZafLogicalColor HeaderTextColor(ZafLogicalColor *color =  
    ZAF_NULLP(ZafLogicalColor), ZafLogicalColor *mono =  
    ZAF_NULLP(ZafLogicalColor));  
virtual ZafLogicalColor  
    SetHeaderTextColor(ZafLogicalColor color,  
        ZafLogicalColor mono = ZAF_MONO_NULL);
```

HeaderTextColor() is the text color used for the **ZafTableHeader** objects in the **ZafTable**. **SetHeaderTextColor()** may be called to change the text color used by all the **ZafTableHeader** objects attached to the **ZafTable**. See **ZafWindowObject::TextColor()** for more information.

HeaderWidth

```
int HeaderWidth(void) const;  
virtual int SetHeaderWidth(int headerWidth);
```

HeaderWidth() specifies the width of the row **ZafTableHeader** object in the same coordinate system as **Region()**. A zero value indicates the absence of a row **ZafTableHeader** object. **SetHeaderWidth()** should only be called from the constructor for the derived **ZafTable** class.

InsertRecord

```
virtual ZafError InsertRecord(ZafOffset insertOffset);
```

InsertRecord() inserts the database record at offset *insertOffset* into the **ZafTable** at offset *insertOffset*. The corresponding database record, if any, is not modified. **InsertRecord()** generally returns **ZAF_ERROR_NONE**.

MaxOffset

```
ZafOffset MaxOffset(void) const;  
virtual ZafOffset SetMaxOffset(ZafOffset maxOffset);
```

MaxOffset() specifies one less than the number of records in the **ZafTable**. **MaxOffset()** is zero- based, so a value of 0 means that there is one record in the **ZafTable**, and the default value of -1 means that there are no records in the **ZafTable**. **MaxOffset()** must be set in the constructor for the derived **ZafTable** class using **SetMaxOffset()**. It is automatically updated appropriately by **DeleteRecord()** and **InsertRecord()**.

SetReadFunction

```
MemberTableFunction SetReadFunction(MemberTableFunction
    readFunction);
```

Any function may be used to override ReadRecord() by passing that function into SetReadFunction(). SetReadFunction() returns *readFunction*. See [ReadRecord\(\)](#) for more information. The definition for MemberTableFunction is as follows:

```
typedef ZafError
    (ZafTable::*MemberTableFunction)(ZafTableRecord &record,
    ZafWindowObject *row);
```

ReadRecord

```
virtual ZafError ReadRecord(ZafTableRecord &record,
    ZafWindowObject *row);
```

ReadRecord() must be overridden by the derived ZafTable class to read the data from a record in the database. The data must be loaded into the ZafTableRecord *record* (see [ZafTableRecord](#) for more information). *row* specifies the row header, and the programmer must call SetText() for *row* to set the row header text. If the programmer's ReadRecord() method takes a true ZafTableRecord parameter (where there is no derivation of ZafTableRecord), the base ZafTable::ReadRecord() must be called before any additional functionality occurs.

ReadRecord() returns ZAF_ERROR_INVALID if an error occurred, otherwise ZAF_ERROR_NONE is returned. Any function may be used to override ReadRecord() by passing that function into [SetReadFunction\(\)](#).

Record

```
ZafTableRecord *Record(ZafOffset offset);
```

Record() returns a pointer to the table record at zero-based index *offset* if it is in the viewable area of the table. If the specified record is not in the viewable area of the table, or if it doesn't exist, null is returned.

Repopulate

```
ZafError Repopulate(ZafOffset offset = -1);
```

Repopulate() causes the table record at zero-based index *offset* to be read from the database using ReadRecord(). If *offset* specifies a record that is not in the viewable area of the database, nothing happens. If *offset* is -1, all records in the viewable area of the table are read from the database. If the operation was successful, ZAF_ERROR_NONE is returned; otherwise ZAF_ERROR_INVALID is returned.

RowHeight

```
int RowHeight(void) const;
virtual int SetRowHeight(int rowHeight, ZafCoordinateType
    coordinateType = ZAF_CELL);
```

RowHeight() specifies the height of each ZafTableRecord object in the same coordinate system as Region(). To modify the height of the ZafTableRecord objects, SetRowHeight() may be called before the table is added to the window manager. *rowHeight* specifies the new row height, and *coordinateType* specifies the coordinate system that rowHeight is specified in.

SeekNextRecord

```
virtual ZafError SeekNextRecord(void);
```

SeekNextRecord() causes CurrentOffset() to specify the next record in the ZafTable. If CurrentOffset() already specifies the last record, ZAF_ERROR_INVALID is returned; otherwise, ZAF_ERROR_NONE is returned. SeekNextRecord() may be overridden by the derived ZafTable class to provide additional functionality.

SeekPrevious-Record

```
virtual ZafError SeekPreviousRecord(void);
```

SeekPreviousRecord() causes CurrentOffset() to specify the previous record in the ZafTable. If CurrentOffset() already specifies the first record, ZAF_ERROR_INVALID is returned; otherwise, ZAF_ERROR_NONE is returned. SeekPreviousRecord() may be overridden by the derived ZafTable class to provide additional functionality.

SeekRandom-Record

```
virtual ZafError SeekRandomRecord(ZafOffset offset,
    ZafSeek location);
```

SeekRandomRecord() modifies CurrentOffset() according to *offset* and *location*. The possible values of location and how they interpret *offset* follow:

ZafSeek	Description
ZAF_SEEK_START	Interprets <i>offset</i> as the offset from the beginning of the ZafTable
ZAF_SEEK_CURRENT	Interprets <i>offset</i> as the offset from CurrentOffset()
ZAF_SEEK_END	Interprets <i>offset</i> as the offset from the end of the ZafTable

If the resulting offset is invalid, ZAF_ERROR_INVALID is returned; otherwise, ZAF_ERROR_NONE is returned. SeekRandomRecord() may be overridden by the derived ZafTable class to provide additional functionality.

TopOffset

```
ZafOffset TopOffset(void) const;
virtual ZafOffset SetTopOffset(ZafOffset topOffset);
```

TopOffset() specifies the zero-based index of the record currently at the top of the table's viewable area. **SetTopOffset()** may be called to change it. **SetTopOffset()** does not change which record has focus.

VirtualRecord

```
ZafTableRecord *VirtualRecord(void) const;
virtual ZafTableRecord *SetVirtualRecord(ZafTableRecord
    *virtualRecord);
```

VirtualRecord() is the one **ZafTableRecord** that actually exists for a **ZafTable**. It is used as a template record for all records in the **ZafTable**. **VirtualRecord()** is created automatically by the constructor, but **SetVirtualRecord()** may be called to change it. Normally, the programmer should not need to call **SetVirtualRecord()**, but **VirtualRecord()** may be necessary to modify the virtual record template.

SetWriteFunction

```
MemberTableFunction SetWriteFunction(MemberTableFunction
    writeFunction);
```

Any function may be used to override **WriteRecord()** by passing that function into **SetWriteFunction()**. **SetWriteFunction()** returns *writeFunction*. See [WriteRecord\(\)](#) for more information. The definition for **MemberTableFunction** is as follows:

```
typedef ZafError
    (ZafTable::*MemberTableFunction)(ZafTableRecord &record,
    ZafWindowObject *row);
```

WriteRecord

```
virtual ZafError WriteRecord(ZafTableRecord &record,
    ZafWindowObject *row);
```

WriteRecord() must be overridden by the derived **ZafTable** class to write the data in *record* to the corresponding record in the database. *row* specifies the row header, and the programmer may call **Text()** for *row* to get the row header text. If the programmer's **WriteRecord()** method takes a true **ZafTableRecord** parameter (where there is no derivation of **ZafTableRecord**), the base **ZafTable::WriteRecord()** must be called before any additional functionality occurs.

WriteRecord() returns **ZAF_ERROR_INVALID** if an error occurred, otherwise **ZAF_ERROR_NONE** is returned. Any function may be used to override **WriteRecord()** by passing that function into [SetWriteFunction\(\)](#).

ZafTableHeader

	HeaderType	Table	VirtualField
Inheritance	ZafTableHeader : ZafWindow : (ZafWindowObject : ZafElement), ZafList		
Declaration	#include <z_table.hpp>		
Description	<p>ZafTableHeader provides support for displaying header rows and columns in a ZafTable object. ZafTableHeader is designed as a support class only, so the programmer should normally not need to create a ZafTableHeader object. ZafTableHeader objects are automatically added to a ZafTable object in the ZafTable constructor.</p> <p>This class is not available in the Personal or Registered versions of ZAF, but is included with the Professional version.</p>		

Constructors All ZafTableHeader constructors initialize the member variables associated with an instantiated ZafTableHeader object. The default values set by the ZafTableHeader 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 ZafTableHeader. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafTableHeader

HeaderType()	user-supplied parameter
VirtualField()	null

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Modal()	false [†]
Moveable()	false [†]
NormalHotKeys()	false [†]
Sizeable()	false [†]
Temporary()	false [†]

ZafWindowObject

Member Initializations

AcceptDrop()	false [†]
Bordered()	false [†]
Focus()	false [†]
HelpContext()	null [†]
Noncurrent()	true [†]
OSDraw()	false
RegionType()	ZAF_AVAILABLE_REGION [†]
SupportObject()	true [†]
UserFunction()	null [†]

ZafElement

ClassID()	ID_ZAF_TABLE_HEADER
ClassName()	"ZafTableHeader"
NumberID()	ZAF_NUMID_COLUMN_HEADER, ZAF_NUMID_CORNER_HEADER, or ZAF_NUMID_ROW_HEADER
StringID()	"ZAF_COLUMN_HEADER", "ZAF_CORNER_HEADER", or "ZAF_ROW_HEADER"

ZafTableHeader(int width, int height, ZafTableHeaderType headerType = ZAF_ROW_HEADER);

This constructor is useful in straight-code situations. *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. *headerType* specifies the header type (see [HeaderType\(\)](#) for more information).

ZafTableHeader(const ZafTableHeader ©);

The copy constructor creates a new ZafTableHeader object and initializes its data from *copy*.

ZafTableHeader(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.

A simple example of how to create table headers follows:

```
// To ensure proper sizing, always add the corner header first.
```

```
table->Add(new ZafTableHeader(5, 1, ZAF_CORNER_HEADER));

//Add the "top" column header.
table->Add(new ZafTableHeader(0, 1, ZAF_COLUMN_HEADER));

//Create the "side" row header.
ZafTableHeader *rowHeader = new ZafTableHeader(5, 0,
    ZAF_ROW_HEADER);
ZafString *string = new ZafString(0, 0, 5, "", -1);
string->SetHzJustify(ZAF_HZ_RIGHT);
rowHeader->SetVirtualField(string);
table->Add(rowHeader);
```

Destructor

```
virtual ~ZafTableHeader(void);
```

The destructor is used to free the memory associated with a ZafTableHeader object. It chains to the ZafWindow, ZafWindowObject, ZafList, and ZafElement destructors. Generally, the programmer will not directly destroy a ZafTableHeader object, since it is automatically destroyed when its parent table 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.

HeaderType

```
ZafTableHeaderType HeaderType(void) const;
virtual ZafTableHeaderType
    SetHeaderType(ZafTableHeaderType headerType);
```

HeaderType() specifies what type of header this ZafTableHeader is. SetHeaderType() may be called to change the value of this attribute before the header is added to the ZafTable. The possible values for HeaderType() follow:

HeaderType()	Description
ZAF_COLUMN_HEADER	Specifies a column header to be positioned at the top edge of the ZafTable
ZAF_CORNER_HEADER	Specifies a corner header to be positioned in the space between the column and row headers in the top left corner of the ZafTable
ZAF_ROW_HEADER	Specifies a row header to be positioned at the left edge of the ZafTable

Table

```
ZafTable *Table(void) const;
```

Table() returns a pointer to the ZafTable object that the ZafTableHeader is attached to. Table() returns null if the header has not been added to a table.

VirtualField

```
ZafWindowObject *VirtualField(void) const;  
virtual ZafWindowObject *SetVirtualField(ZafWindowObject  
    *virtualField);
```

VirtualField() is the one ZafWindowObject that actually exists for a row ZafTableHeader. It is used as a template field for all fields in the row ZafTableHeader. VirtualField() must be set before the row header is added to the ZafTable by calling SetVirtualField(). Column and corner headers display the text of their respective children, and do not use VirtualField().

ZafTableRecord

ActivateObject	Offset
DeactivateObject	Table

Inheritance ZafTableRecord : ZafWindow : (ZafWindowObject : ZafElement), ZafList

Declaration #include <z_table.hpp>

Description ZafTableRecord provides support for displaying records in a ZafTable object. ZafTableRecord is designed as a support class only, so the programmer should normally not need to create a ZafTableRecord object. A virtual ZafTableRecord object is automatically added to a ZafTable object in the ZafTable constructor. See [ZafTable](#) for more information.

This class is not available in the Personal or Registered versions of ZAF, but is included with the Professional version.

Constructors All ZafTableRecord constructors initialize the member variables associated with an instantiated ZafTableRecord object. The default values set by the ZafTableRecord 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 ZafTableRecord. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafTableRecord

Offset()	-1
----------	----

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Modal()	false [†]
Moveable()	false [†]
NormalHotKeys()	false [†]
SelectionType()	ZAF_MULTIPLE_SELECTION
Sizeable()	false [†]
Temporary()	false [†]

Member Initializations

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	false [†]
Disabled()	false [†]
Noncurrent()	false [†]
ParentDrawFocus()	true [†]
RegionType()	ZAF_INSIDE_REGION [†]
SupportObject()	false [†]

ZafElement

ClassID()	ID_ZAF_TABLE_RECORD
ClassName()	"ZafTableRecord"

ZafTableRecord(int width, int height);

This constructor is useful in straight-code situations. *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.

ZafTableRecord(const ZafTableRecord ©);

The copy constructor creates a new ZafTableRecord object and initializes its data from *copy*.

ZafTableRecord(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.

See [ZafTable](#) for an example of how to utilize ZafTableRecord objects.

Destructor

virtual ~ZafTableRecord(void);

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

Members*ActivateObject*

```
virtual bool ActivateObject(ZafWindowObject *object);
```

ActivateObject() causes *object* (which is a child of the table record) to be activated. In other words, the object becomes editable. ActivateObject() returns true if successful, and false otherwise. Normally, the programmer should not call ActivateObject(), as it is used internally by ZAF.

DeactivateObject

```
virtual bool DeactivateObject(ZafWindowObject *object);
```

DeactivateObject() causes *object* (which is a child of the table record) to be deactivated. In other words, if the object is editable, it becomes non-editable. DeactivateObject() returns false if successful, and true otherwise. Normally, the programmer should not call DeactivateObject(), as it is used internally by ZAF.

Offset

```
ZafOffset Offset(void) const;
virtual ZafOffset SetOffset(ZafOffset offset);
```

Offset() is the zero-based record offset for the ZafTableRecord in the ZafTable. Offset() is maintained by the ZafTable object, so the programmer should normally not call SetOffset(), but Offset() is useful when a record's offset in the parent table is needed.

Table

```
ZafTable *Table(void) const;
```

Table() returns a pointer to the ZafTable object that the ZafTableRecord is attached to. Table() returns null if the header has not been added to a table.

ZafText

AutoClear	HzJustify	Unanswered
CursorOffset	Invalid	ViewOnly
CursorPosition	StringData	WordWrap
Event	Text	

Inheritance ZafText : ZafWindow : (ZafWindowObject : ZafElement),
 ZafList

Declaration #include <z_text.hpp>

Description 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. Only 32K of text may be portably contained in a ZafText object, due to the limitation of various native environments.

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

AutoClear()	false
HzJustify()	ZAF_HZ_LEFT
Invalid()	false
StringData()	null
Unanswered()	false
ViewOnly()	false
WordWrap()	true

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Modal()	false [†]
Moveable()	false [†]
NormalHotKeys()	false [†]

Member Initializations

SelectionType()	ZAF_SINGLE_SELECTION [†]
Sizeable()	false [†]
Temporary()	false [†]

ZafWindowObject

Bordered()	true
------------	------

ZafElement

ClassID()	ID_ZAF_TEXT
ClassName()	"ZafText "

```
ZafText(int left, int top, int width, int height, const  
        ZafIChar *text, int maxLength);
```

This constructor is useful in straight-code situations, particularly if you wish the ZafText object to create, maintain and destroy its own ZafStringData object automatically. *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. *text* is the text you wish to initially appear in the new ZafText object, and *maxLength* is passed to the ZafStringData constructor, the maximum number of characters the user may type into the field (see the [ZafStringData](#) constructor for more information). If you pass -1 in for the *maxLength* parameter, the number of characters the user may type is not restricted.

```
ZafText(int left, int top, int width, int height,  
        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 ZafText class create and maintain the data pieces automatically. For example, to maintain a database of ZafStringData objects and tie them into ZafText objects, maintain your own ZafStringData objects and create ZafText objects using your ZafStringData objects by passing them into the *stringData* parameter of this constructor. For more information on using ZafStringData objects, see [ZafStringData](#). The *left*, *top*, *width* and *height* parameters are the same as the previous constructor.

```
ZafText(const ZafText &copy);
```

The copy constructor is used in conjunction with the overloaded Duplicate() function. It accepts another ZafText object and copies the object's information. If the data objects are StaticData() then the new ZafText object simply points to the original data objects, otherwise a copy is made of them for the 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.

AutoClear

```
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.

CursorOffset

```
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);
```

CursorPosition

```
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);
```

Event

```
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
S_COPY	causes the object to copy its selected data to the clipboard
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

HzJustify

```
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.

Invalid

```
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.

StringData

```
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](#). `SetStringData()` will delete the previous `StringData()` object if it is `Destroyable()` and no other object uses it.

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);
```

Text

```
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");
```

Unanswered

```
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()`.

ViewOnly

```
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.

WordWrap

```
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
Inheritance	ZafTime : ZafString : ZafWindowObject : ZafElement	
Declaration	#include <z_time1.hpp>	
Description	<p>ZafTime is a single-line time object that allows user input through the keyboard. ZafTime is fully internationalized to display and input using any format. See ZafString::AllowInvalid() and ZafString::ReportInvalid() for information on these attributes and how they affect validation for this class.</p> <p>All ZafTime objects refer to data contained in a ZafTimeData object (refer to this class for additional essential information).</p>	
Formats	<p>ZafString and derived classes parse input and display output based on default or programmer-defined input and output formats. Programmers may use the SetInputFormat() and SetOutputFormat() families of functions to change the default format. The functions are documented in the ZafString reference.</p> <p>Each class derived from ZafFormatData handles a different set of formatting arguments, in addition to those supported by its base classes. ZafTimeData (and therefore ZafTime) handles the following “time” subset of the arguments supported by ZafUTimeData (the parent class to ZafTimeData):</p>	

Format Argument	Substitution
%%	'%' character
%h	Same as %H
%H	Hour (24-hour clock) as a decimal number [00, 23]
%i	Same as %I; can be one or two digits
%I	Hour (12-hour clock) as a decimal number [01, 12]
%J	Seconds in tenths (.1)
%k	Seconds in hundredths (.01)
%K	Seconds in thousandths (.001)
%p	Locale equivalent of either a.m. or p.m.
%r	Time in a.m. or p.m. notation; in the POSIX locale this is equivalent to: %I:%M:%S %p
%R	Time in 24-hour notation (%H:%M)
%s	Same as %S; can be one or two digits
%S	Seconds as a decimal number [00, 59]

Format Argument	Substitution
%T	Time (%H:%M:%S)
%X	Locale-specific time representation
%Z	Timezone name or abbreviation

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 the ZafTime object is to create, maintain and destroy its own ZafTimeData object automatically. *left*, *top*, and *width* are 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 that 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 `ZafTimeData` object, rather than having the `ZafTime` class create and maintain the data object automatically. For more information on using `ZafTimeData` objects, see [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

Event

```
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
N_RESET_I18N	causes the object to redisplay its data according to the new internationalization values
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

TimeData

```
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](#). SetTimeData() will delete the previous TimeData() object if it is Destroyable() and no other object uses it.

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 code snippet for an example using ZafTimeData objects with ZafTime.

ZafTimeData

	Clear	FormattedText	SetTime																				
Inheritance	ZafTimeData : ZafUTimeData : ZafFormatData : ZafData : ZafElement, ZafNotification																						
Declaration	#include <z_time.hpp>																						
Description	<p>ZafTimeData combines time encapsulation with data notification and object notification from ZafData. It is most often used in conjunction with the Zaf-Time user interface object but may be used as a stand-alone object if desired. Refer to the ZafUTimeData documentation for a discussion of member methods, inherited by ZafTimeData, used to retrieve and set time-specific information (e.g., hour, minute, second, etc.).</p> <p>All ZafData objects may make use of printf-style formatting and parsing arguments during string operations. In addition, all ZafUTimeData objects may make use of strftime- and strptime-style formatting and parsing arguments during string operations. Refer to standard library documentation for detailed information on printf, strftime, and strptime functions as well as their corresponding conversion characters.</p>																						
Constructors	<p>ZafTimeData constructors initialize the member variables associated with a new ZafTimeData object and allocate space to hold the time data. The default values set by ZafTimeData follow, if they are overridden from those set by base class constructors:</p> <h3>Member Initializations</h3> <table><tr><td colspan="2">ZafUTimeData</td></tr><tr><td>Hour()</td><td>(varies by constructor)</td></tr><tr><td>MilliSecond()</td><td>(varies by constructor)</td></tr><tr><td>Minute()</td><td>(varies by constructor)</td></tr><tr><td>Second()</td><td>(varies by constructor)</td></tr><tr><td>Value()</td><td>(varies by constructor)</td></tr><tr><td>ZoneOffset()</td><td>(varies by constructor)</td></tr></table> <table><tr><td colspan="2">ZafElement</td></tr><tr><td>ClassID()</td><td>ID_ZAF_time_DATA</td></tr><tr><td>ClassName()</td><td>"ZafTimeData"</td></tr></table>			ZafUTimeData		Hour()	(varies by constructor)	MilliSecond()	(varies by constructor)	Minute()	(varies by constructor)	Second()	(varies by constructor)	Value()	(varies by constructor)	ZoneOffset()	(varies by constructor)	ZafElement		ClassID()	ID_ZAF_time_DATA	ClassName()	"ZafTimeData"
ZafUTimeData																							
Hour()	(varies by constructor)																						
MilliSecond()	(varies by constructor)																						
Minute()	(varies by constructor)																						
Second()	(varies by constructor)																						
Value()	(varies by constructor)																						
ZoneOffset()	(varies by constructor)																						
ZafElement																							
ClassID()	ID_ZAF_time_DATA																						
ClassName()	"ZafTimeData"																						


```
ZafTimeData(void);
```

The basic constructor allocates a ZafTimeData instance and initializes its value to the current system time.

```
ZafTimeData(int hour, int minute, int second, int
             milliSecond);
```

This constructor allocates a ZafTimeData instance and initializes its contents to the time corresponding to *hour*, *minute*, *second* and *milliSecond*.

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

This constructor allocates a ZafTimeData instance and initializes its contents to the time equivalent of *string*. The conversion uses the strptime-style specifier *format* to interpret the string. If *format* is null ZafTimeData uses its locale-specific default format.

```
ZafTimeData(const ZafTimeData &copy);
ZafTimeData(const ZafUTimeData &copy);
```

These constructors are the copy constructors. They each allocate a new ZafTimeData instance and copy all member data from copy.

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

This constructor is the persistent constructor. It allocates a new ZafTimeData 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 ZafTimeData creation techniques
ZafTimeData time1(13, 34, 20, 0);
ZafTimeData copyTime = time1;
ZafTimeData time2("16 15 30", "%H %M %S");
ZafTimeData systemTime;
```

Destructor

```
virtual ~ZafTimeData(void);
```

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

Members

Clear

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

Clear() sets the value of a ZafTimeData object to the current system time.

FormattedText

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

FormattedText() fills *buffer* with a string representation of the ZafTimeData using the strftime-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* characters. FormattedText() returns the integer value it receives from its call to strftime().

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

```
ZafTimeData time(18, 25, 30);  
time.FormattedText(buffer, 256, "%I:%M:%S %p");  
printf("time - %s\n", buffer);
```

```
time.FormattedText(buffer, 256, "%H:%M");  
printf("time - %s\n", buffer);
```

```
=====  
time - 06:25:30 p.m.  
time - 18:25
```

SetTime

```
virtual ZafError SetTime(int hour, int minute, int second,  
    int milliSecond);
```

```
virtual ZafError SetTime(const ZafIChar *buffer, const  
    ZafIChar *format);
```

```
virtual ZafError SetTime(const ZafTimeData &time);
```

SetTime() functions set the value of the ZafTimeData object from numeric input, another time, or an interpreted string. Refer to FormattedText() for more information on time/string conversions.

ZafTimer

Add	NotifyMessage	operator +
Event	Poll	operator -
Interval	QueueEvents	
NotifyCount	Subtract	

Inheritance ZafTimer : ZafDevice : ZafElement

Declaration #include <z_timer.hpp>

Description The ZafTimer device notifies a list of objects when a specified interval of time has passed. Notification is made by sending a ZafEvent to each object in the notification list. This message is N_TIMER by default, but may be specified by the programmer.

Since ZafTimer is not interrupt-driven, it is not guaranteed that the objects will receive a notification message within the exact interval specified. Although native OS timers may queue timer events with relative accuracy, these events do not interrupt applications and are therefore subject to limitations of event polling. For maximum accuracy, ZafTimer uses native timers where available.

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

Member Initializations

ZafTimer

Interval()	0
NotifyMessage()	N_TIMER
QueueEvents()	false

ZafDevice

DeviceType()	E_TIMER
---------------	---------

ZafElement

ClassID()	ID_ZAF_TIMER
ClassName()	"ZafTimer"
NumberID()	ID_ZAF_TIMER
StringID()	"ZafTimer"

```
ZafTimer(ZafDeviceState state = D_ON, unsigned long  
interval);
```

This constructor is used to instantiate a **ZafTimer** object to be added to the **ZafEventManager**. *state* specifies the initial state of the device, and *interval* specifies the time interval in milliseconds, that expires before a message is sent to the list of notification objects (see [ZafDevice](#) for valid states).

```
ZafTimer(const ZafTimer &copy);
```

The copy constructor accepts another **ZafTimer** object and copies the object's information. The list of notification objects is not copied, since two timers should not send notifications to the same objects. An example of how to create a **ZafTimer** object follows:

```
// Create a timer.  
ZafTimer *timer = new ZafTimer(D_ON, 1000);  
// Add the field to be notified.  
timer->Add(myObject);  
// Add the timer to the event manager, which also activates the  
timer.  
zafEventManager->Add(timer);
```

Destructor

```
virtual ~ZafTimer(void);
```

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

The programmer may delete a **ZafTimer** object when it is no longer needed; otherwise 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.

Add

```
ZafWindowObject *Add(ZafWindowObject *object,  
ZafEventType newNotifyMessage, ZafWindowObject  
*position = ZAF_NULLP(ZafWindowObject));
```

```

ZafWindowObject *Add(ZafWindowObject *object,
    ZafWindowObject *position =
    ZAF_NULLP(ZafWindowObject));
operator +
ZafTimer &operator+(ZafWindowObject *object);

```

These functions and operator add object to the timer's notification list. When the time interval expires, the timer sends either *newNotifyMessage* (unique to the object) or the default notify message to each objects on the notification list.

```

Event
virtual ZafEventType Event(const ZafEventStruct &event);

```

This overloaded function handles all events sent to the ZafTimer object. The following events are handled:

ZafEventType	Description
D_DEINITIALIZE	causes the timer to deinitialize its notification mechanism
D_INITIALIZE	causes the timer to initialize its notification mechanism
D_OFF	causes the timer to stop sending notification messages
D_ON	causes the timer to begin sending notification messages
D_STATE	causes the timer to return its device state
S_ADD_OBJECT	causes event.windowObject to be added to the notification list
S_SUBTRACT_OBJECT	causes event.windowObject to be removed from the notification list

```

Interval
unsigned long Interval(void) const;
virtual unsigned long SetInterval(unsigned long
    interval);

```

Interval() returns the notification time interval in milliseconds. SetInterval may be used to change the timer delay after a ZafTimer has been created

```

NotifyCount
int NotifyCount(void) const;

```

NotifyCount() returns the number of objects in the timer's notification list. This function is useful if the programmer wishes to delete a ZafTimer when it is no longer used by any objects.

NotifyMessage

```
ZafEventType NotifyMessage(void) const;  
ZafEventType SetNotifyMessage(ZafEventType  
    notifyMessage);
```

When `Interval()` expires, the timer sends an event of type `NotifyMessage()` to all the objects on the notification list that don't have their own specific event type assigned via `Add()`. `NotifyMessage()` defaults to `N_TIMER`, but the programmer may pass a different event type into the constructor or `SetNotifyMessage()`.

Poll

```
virtual void Poll(void);
```

In some environments, the `Poll()` function checks the timer device to see if the `Interval()` has expired, if the `ZafTimer`'s state is not `D_OFF`. In other environments where timer events are handled automatically by the native environment, `Poll()` may simply block timer events from coming through ZAF's event manager queue if the `ZafTimer`'s state is `D_OFF`.

QueueEvents

```
bool QueueEvents(void) const;  
virtual bool SetQueueEvents(bool queueEvents);
```

`QueueEvents()` causes the timer to put an event of type `NotifyMessage()` on ZAF's event manager queue after all objects in the notification list have been notified directly

Using this option, a `ZafTimer` may "notify" the Zaf system generally rather than directly calling the event function for individual objects. `QueueEvents()` defaults to false, but may be changed with `SetQueueEvents()`.

*Subtract
operator -*

```
ZafWindowObject *Subtract(ZafWindowObject *object);  
ZafTimer &operator-(ZafWindowObject *object);
```

This function and operator subtract *object* from the timer's notification list, so that the *object* is no longer sent notification messages.

ZafTitle

Inheritance ZafTitle : ZafButton : ZafWindowObject : ZafElement

Declaration #include <z_title.hpp>

Description ZafTitle is the title bar decoration on a ZafWindow, and is generally drawn by the environment. A ZafTitle object may only be added to a ZafWindow, and allows a user to move the parent window with a ZafMouse device.

ZafTitle can be automatically added to a window along with other window “decorations” by using ZafWindow::AddGenericObjects().

Constructors All ZafTitle constructors initialize the member variables associated with an instantiated ZafTitle object. The default values set by the ZafTitle 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 ZafTitle. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafButton

AllowDefault()	false [†]
AllowToggling()	false [†]
AutoRepeatSelection()	false [†]
AutoSize()	true [†]
BitmapData()	null [†]
ButtonType()	ZAF_FLAT_BUTTON [†]
Depressed()	false [†]
Depth()	0 [†]
HotKeyChar()	0 [†]
HotKeyIndex()	-1 [†]
HzJustify()	ZAF_HZ_CENTER [†]
SelectOnDoubleClick()	true [†]
SelectOnDownClick()	true [†]
SendMessageText()	null [†]
SendMessageWhenSelected()	true [†]
Value()	Used internally by ZAF [†]
VtJustify()	ZAF_VT_CENTER [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	false [†]

Member Initializations

CopyDraggable()	false [†]
Disabled()	false [†]
Focus()	false [†]
HelpContext()	null [†]
HelpObjectTip()	null [†]
LinkDraggable()	false [†]
MoveDraggable()	false [†]
Noncurrent()	true [†]
ParentDrawBorder()	false [†]
ParentDrawFocus()	false [†]
ParentPalette()	false [†]
QuickTip()	null [†]
RegionType()	ZAF_AVAILABLE_REGION [†]
Selected()	false [†]
SupportObject()	true [†]
SystemObject()	false
UserFunction()	null [†]

ZafElement

ClassID()	ID_ZAF_TITLE
ClassName()	"ZafTitle"
NumberID()	ZAF_NUMID_TITLE
StringID()	"ZAF_NUMID_TITLE"

ZafTitle(const ZafIChar *text);

This constructor is useful in straight-code situations, particularly if you wish the ZafTitle object to create, maintain and destroy its own ZafStringData object automatically. Simply pass the text to appear in the title bar into the *text* parameter.

ZafTitle(ZafStringData *textData);

This constructor is also useful in straight-code situations, particularly if you have already created a ZafStringData object to be associated with the title bar. This constructor could be used to maintain string data pieces manually, rather than having the ZafTitle class create and maintain the string data pieces automatically. For more information see [ZafStringData](#). *textData* specifies the string data object to be associated with the title bar.


```
ZafTitle(const ZafTitle &copy);
```

The copy constructor creates a new ZafTitle object and initializes its data from copy.

```
ZafTitle(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 title bar follows:

```
// Create a sample window.  
ZafWindow *window = new ZafWindow(0, 0, 40, 10);  
ZafTitle *title = new ZafTitle("Test Window");  
window->Add(title);
```

Destructor

```
virtual ~ZafTitle(void);
```

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

ZafToolBar

	DockType	WrapChildren
Inheritance	ZafToolBar : ZafWindow : (ZafWindowObject : ZafElement), ZafList	
Declaration	#include <z_tbar.hpp>	
Description	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.	
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

DockType()	ZAF_DOCK_TOP
WrapChildren()	true

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Modal()	false [†]
Moveable()	false [†]
NormalHotKeys()	false [†]
Sizeable()	false [†]
Temporary()	false [†]

ZafWindowObject

AcceptDrop()	false [†]
Bordered()	true
Noncurrent()	true
RegionType()	ZAF_AVAILABLE_REGION [†]

Member Initializations

SupportObject() true

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, while *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, *left* and *top* 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* 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, "Button 1",
    ZAF_NULLP(ZafBitmapData)));
toolBar->Add(new ZafButton(0, 0, 10, 1, "Button 2",
    ZAF_NULLP(ZafBitmapData)));
ZafButton *spacer = new ZafButton(0, 0, 1, 1,
    ZAF_NULLP(ZafIChar), ZAF_NULLP(ZafBitmapData));
```

```
toolBar->Add(spacer);
toolBar->Add(new ZafButton(0, 0, 10, 1, "Button 3",
    ZAF_NULLP(ZafBitmapData)));
toolBar->Add(new ZafButton(0, 0, 10, 1, "Button 4",
    ZAF_NULLP(ZafBitmapData)));
// 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.

DockType

```
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

WrapChildren

```
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	Expanded	TreeList
DepthCurrent	NormalBitmap	ViewCurrent
DepthFirst	SelectedBitmap	ViewFirst
DepthLast	SetSelectionType	ViewLast
DepthNext	StringData	ViewLevel
DepthPrevious	Text	ViewNext
Expandable	ToggleExpanded	ViewPrevious

Inheritance ZafTreeItem : ZafWindow : (ZafWindowObject : ZafElement),
 ZafList

Declaration #include <z_tree.hpp>

Description 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. Like ZafHzList and ZafVtList items, tree items may not be Noncurrent(). Since the ZafTreeList and ZafTreeItem classes were specially designed to work together, only other ZafTreeItem objects may be added to a ZafTreeItem object. See [ZafTreeList](#) for a more detailed description.

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 [†]

Member Initializations

Minimized()	false [†]
Modal()	false [†]
Moveable()	false [†]
NormalHotKeys()	false [†]
Sizeable()	false [†]
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, const 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 expanded 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 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, 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.

AutoSortData

```
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.

DepthCurrent

```
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 viewable.

DepthFirst

```
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). This traversal will find every `ZafTreeItem` in the sub-tree, whether in an expanded branch or not. This is a depth traversal, as opposed to a depth-first traversal. See `ZafTreeList::DepthFirst()` for more information. The following code shows how to do this traversal:

```
// Traverse every item of the entire sub-tree hierarchy.
for (ZafTreeItem *item = this; item; item = item->DepthNext())
    if (item == matchItem)
        break;
```

For the following sub-tree, the traversal would find the nodes in this order: A, A1, A2.

```
A
|
|__A1
|__A2
```

DepthLast `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. This traversal will find every ZafTreeItem in the sub-tree, whether in an expanded branch or not. This is a depth traversal, as opposed to a depth-first traversal. See [DepthFirst\(\)](#) for example code.

DepthNext `ZafTreeItem *DepthNext(void);`

DepthNext() returns the next item in a depth traversal of the sub-tree. See [DepthFirst\(\)](#) for more information.

DepthPrevious `ZafTreeItem *DepthPrevious(void);`

DepthPrevious() returns the previous item in a depth traversal of the sub-tree. See [DepthFirst\(\)](#) and [DepthLast\(\)](#) for more information.

Expandable `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.

Expanded `bool Expanded(void) const;`
`virtual bool SetExpanded(bool expanded);`
ToggleExpanded `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. Tog-

gleExpanded() will toggle the value of this attribute, collapsing an Expanded() ZafTreeItem object, and expanding a collapsed ZafTreeItem object.

NormalBitmap
SelectedBitmap

```
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](#). SetNormalBitmap() will delete the previous NormalBitmap() object if it is Destroyable() and no other object uses it. SetSelectedBitmap() will delete the previous SelectedBitmap() object if it is Destroyable() and no other object uses it.

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);
```

SetSelectionType

```
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.

StringData

```
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](#). SetStringData() will delete the previous StringData() object if it is Destroyable() and no other object uses it.

The return value for SetStringData() is normally ZAF_ERROR_NONE.

Refer to Set*Bitmap() for more information.

Text

```
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.

ToggleExpanded

```
virtual bool ToggleExpanded(void);
```

See [Expanded\(\)](#).

TreeList

```
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.

ViewCurrent

```
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.

ViewFirst

```
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 additional ZafTreeItem

objects. The following code shows how to search through the viewable items in the sub-tree:

```
// Find a visible item in the sub-tree.
for (ZafTreeItem *item = ViewFirst(); item; item = item->
    ViewNext())
    if (item->NumberID() == matchID)
        break;
```

ViewLast

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 additional ZafTreeItem objects. See [ViewFirst\(\)](#) for example code.

ViewLevel

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.

ViewNext

ZafTreeItem ***ViewNext**(void);

ViewNext() returns the next viewable item in the sub-tree, beginning with this ZafTreeItem object. See [ViewFirst\(\)](#) for more information.

ViewPrevious

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	DrawLines	ViewCurrent
AutoSortData	SelectionType	ViewFirst
DepthCurrent	SetBackgroundColor	ViewLast
DepthFirst	SetTextColor	
DepthLast	ViewCount	

Inheritance ZafTreeList : ZafWindow : (ZafWindowObject : ZafElement),
 ZafList

Declaration #include <z_tree.hpp>

Description ZafTreeList is a scrollable hierarchical list object that intuitively presents a “tree” of list items. Each object contained in a ZafTreeList may or may not contain other objects. Objects that contain children may be expanded and collapsed to expose or hide contents. Since the ZafTreeList and ZafTreeItem classes were specially designed to work together, only ZafTreeItem objects may be added to a ZafTreeList. The ZafTreeList class utilizes the native OS tree list API if available, or is closely modeled after popular tree controls on each platform.

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 [†]
Modal()	false [†]
Moveable()	false [†]
NormalHotKeys()	false [†]
Sizeable()	false [†]

Member Initializations

Temporary()	false [†]
-------------	--------------------

ZafWindowObject

Bordered()	true
------------	------

ZafElement

ClassID()	ID_ZAF_TREE_LIST
ClassName()	"ZafTreeList"

ZafTreeList(int left, int top, int width, int height);

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, while *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. See [ZafWindowObject::SetCoordinateType\(\)](#) for more information.

ZafTreeList(const ZafTreeList ©);

The copy constructor is used in conjunction with the overloaded [Duplicate\(\)](#) function. It allocates a new [ZafTreeList](#) object and initializes its data from *copy*.

ZafTreeList(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 [ZafTreeList](#) creation techniques:

```
// 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 children and add to the tree list
```

```
ZafTreeItem *item1 = new ZafTreeItem(nBitmap, sBitmap, "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 ~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.

AddDepthItem

```
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);
```

AutoSortData

```
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.

DepthCurrent

```
ZafTreeItem *DepthCurrent(void);
```

`DepthCurrent()` returns the current leaf `ZafTreeItem` object in the current branch of the hierarchy. The current leaf item is the item in the tree list that has focus when the tree list has focus and all its ancestors (higher level or enclosing tree items) are expanded. The current leaf item will not necessarily be viewable, meaning that not all its ancestors may be expanded.

DepthFirst

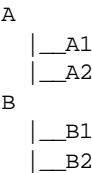
```
ZafTreeItem *DepthFirst(void);
```

`DepthFirst()` returns the first top-level `ZafTreeItem` object in the tree. If there are no tree items, null is returned. This function may be used together with `ZafTreeItem::DepthNext()` to perform a depth traversal of the entire hierarchy of the tree list. This traversal will find every `ZafTreeItem` in the list, whether in an expanded branch or not. This is a depth traversal, as opposed to a depth-first traversal. The following code shows how to do this traversal:

```
// Traverse all the tree items in the entire hierarchy.
for (ZafTreeItem *item = treeList->DepthFirst(); item; item =
    item->DepthNext())
    if (item == matchItem)
```

```
break;
```

For the following tree, the traversal would find the nodes in this order: A, A1, A2, B, B1, B2.



DepthLast

```
ZafTreeItem *DepthLast(void);
```

DepthLast() returns the last leaf ZafTreeItem object in the last branch of the hierarchy. If there are no tree items, null is returned. The last leaf item will not necessarily be viewable, since it may be in a non-expanded branch. This function may be used together with ZafTreeItem::DepthPrevious() to perform a reverse depth traversal of the entire hierarchy of the tree list. This traversal will find every ZafTreeItem in the list, whether in an expanded branch or not. This is a depth traversal, as opposed to a depth-first traversal. See [DepthFirst\(\)](#) for sample code. See also ZafTreeItem::DepthNext().

DrawLines

```
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.

SelectionType

```
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.

SelectionType()	Description
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.

SetBackgroundColor

```
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.

SetTextColor

```
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.

ViewCount

```
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.

ViewCurrent

```
ZafTreeItem *ViewCurrent(void);
ZafTreeItem *SetViewCurrent(ZafTreeItem *item);
```

The ViewCurrent() item is the current viewable ZafTreeItem (none of its ancestor tree items is not expanded) in the current branch of the tree list. The ViewCurrent() item has focus when the tree list has focus. ZafTreeList objects

call `SetViewCurrent()` internally, and the programmer should normally not call `SetViewCurrent()`.

ViewFirst

```
ZafTreeItem *ViewFirst(void);
```

`ViewFirst()` returns a pointer to the first viewable item in the tree list, which is always the first top-level child of the tree list. If there are no items in the tree list, null is returned. In the context of `ZafTreeList`, the “viewable” items may be seen by scrolling through the list without expanding any additional `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;
```

ViewLast

```
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
Inheritance	ZafUTime : ZafString : ZafWindowObject : ZafElement	
Declaration	#include <z_utime1.hpp>	
Description	<p>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. See ZafString::AllowInvalid() and ZafString::ReportInvalid() for information on these attributes and how they affect validation for this class.</p> <p>All ZafUTime objects refer to data contained in a ZafUTimeData object (refer to this class for additional essential information). ZafUTime includes “year 2000 compliance,” meaning that through the underlying ZafUTimeData object, a ZafUTime object keeps track of the exact year, rather than just the last two digits.</p>	
Formats	<p>ZafString and derived classes parse input and display output based on default or programmer-defined input and output formats. Programmers may use the SetInputFormat() and SetOutputFormat() families of functions to change the default format. The functions are documented in the ZafString reference.</p> <p>Each class derived from ZafFormatData handles a different set of formatting arguments, in addition to those supported by its base classes. ZafUTimeData (and therefore ZafUTime) handles the following arguments:</p>	
	Format Argument	Substitution
	%%	'%' character
	%a	Locale-specific abbreviated weekday name
	%A	Locale-specific full weekday name
	%b	Locale-specific abbreviated month name
	%B	Locale-specific full month name
	%c	Locale-specific date and time representation
	%C	Century number (the year divided by 100 and truncated to an integer) as a decimal number [00-99]
	%d	Day of month as a decimal number [01, 31]
	%D	Same as %m/%d/%Y
	%e	Day of the month as a decimal number [1, 31]; a single digit is preceded by a space

Format Argument	Substitution
%EC	Number of the base year (period) in the locale's alternative representation
%Ey	The offset from %EC (year only) in the locale's alternative representation
%EY	The full alternative year representation
%g	Month as an abbreviated month name
%G	Day as an abbreviated day name
%h	Same as %H
%H	Hour (24-hour clock) as a decimal number [00, 23]
%i	Same as %I; may be one or two digits
%I	Hour (12-hour clock) as a decimal number [01, 12]
%j	Day of the year as a decimal number [01, 366]
%J	Seconds in tenths (.1)
%k	Seconds in hundredths (.01)
%K	Seconds in thousandths (.001)
%m	Month as a decimal number [01, 12]
%M	Minute as a decimal number [00, 59]
%p	Locale equivalent of either a.m. or p.m.
%r	Time in a.m. or p.m. notation; in the POSIX locale this is equivalent to: %I:%M:%S %p
%R	Time in 24-hour notation (%H:%M)
%s	Same as %@; may be one or two digits
%S	Seconds as a decimal number [00, 59]
%T	Time (%H:%M:%S)
%u	Weekday as a decimal number [1, 7], with 1 representing Monday
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00, 53]
%v	Month number as a decimal [1, 12]
%V	Week number of the year (Monday as the first day of the week) as a decimal number [01, 53]; if the week containing 1 January has four or more days in the new year then it is considered week 1; otherwise, it is week 53 of the previous year, and the next week is week 1
%w	Replaced by the weekday as a decimal number [0, 6], with 0 representing Sunday

Format Argument	Substitution
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00, 53]; all days in a new year preceding the first Sunday are considered to be in week 0
%x	Locale-specific date representation
%X	Locale-specific time representation
%y	Year without century as a decimal number [00, 99]
%Y	Year with century as a decimal number
%Z	Timezone name or abbreviation

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 the ZafU-
Time object is 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 that 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 `ZafUTimeData` object, rather than having the `ZafUTime` class create and maintain the data object automatically. For more information on using `ZafUTimeData` objects, see [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

Event

```
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
N_RESET_I18N	Causes the object to redisplay its data according to the new internationalization values (as a result of calling ZafI18nData::ResetI18n())
S_COPY_DATA	Causes the object to copy into its existing UTimeData() object event.windowObject's UTimeData(), but only if event.windowObject is a ZafUTime object
S_SET_DATA	Causes the object to create for itself a new UTimeData() object, then copy into it event.windowObject's UTimeData(), but only if event.windowObject is a ZafUTime object

UTimeData

```
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](#). SetUTimeData() will delete the previous UTimeData() object if it is Destroyable() and no other object uses it.

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 [Constructors](#) code snippet for an example using ZafUTimeData objects with ZafUTime.

ZafUTimeData

BasisYear	MilliSecond	operator -
Clear	Minute	operator --
Day	Month	operator =
DayName	MonthName	operator +=
DayOfWeek	Second	operator -=
DayOfYear	SetUTime	operator <
DaysInMonth	TimeName	operator <=
DaysInYear	Value	operator >
FormattedText	Year	operator >=
Hour	ZoneOffset	operator ==
LeapYear	operator +	operator !=
long	operator ++	

Inheritance ZafUTimeData : ZafFormatData : ZafData : ZafElement, ZafNotification

Declaration #include <z_utime.hpp>

Description ZafUTimeData combines date and time encapsulation with data notification and object notification from ZafData. It is most often used in conjunction with the ZafUTime user interface object but may be used as a stand-alone object if desired. ZafUTimeData serves as the base class for the ZafDateData and ZafTimeData classes.

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

Format conversion char	Description
"U"	Formats the printf() argument as an utime.
Parse conversion char	
"U"	Parses a scanf() stream component as an utime and stores the value in the supplied utime argument.

Refer to standard library documentation for detailed information on printf, strftime, and strptime functions as well as their corresponding conversion characters.

Constructors

ZafUTimeData constructors initialize the member variables associated with a new ZafUTimeData object and allocate space to hold the date and time data.

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

Member Initializations**ZafUTimeData**

BasisYear()	ZAF_BASIS_YEAR (in z_untime.hpp)
Day()	(varies by constructor)
DayOfWeek()	(varies by constructor)
Hour()	(varies by constructor)
JulianDay()	(varies by constructor)
MilliSecond()	(varies by constructor)
Minute()	(varies by constructor)
Month()	(varies by constructor)
Second()	(varies by constructor)
Value()	(varies by constructor)
Year()	(varies by constructor)
ZoneOffset()	(varies by constructor)

ZafElement

ClassID()	ID_ZAF_ETIME_DATA
ClassName()	"ZafUTimeData"

ZafUTimeData(void);

The basic constructor allocates a ZafUTimeData instance and initializes its value to the current system date and time.

ZafUTimeData(ZafUInt32 seconds);

This constructor allocates a ZafUTimeData instance and initializes its contents to *seconds*. The data is converted to an offset based on the number of seconds elapsed since the year 1970 and stored in an internal buffer.

ZafUTimeData(int year, int month, int day, int hour, int minute, int second, int milliSecond);

This constructor allocates a ZafUTimeData instance and initializes its contents to the date and time values corresponding to *year*, *month*, *day*, *hour*, *minute*,

second and *milliSecond*. The *hour* value is assumed to be in 24-hour format (i.e., 22 means 10 p.m.).

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

This constructor allocates a `ZafUTimeData` instance and initializes its contents to the date and time equivalent of *string*. The conversion uses the `strptime`-style specifier *format* to interpret the string. If *format* is null `ZafUTimeData` uses its locale-specific default format.

```
ZafUTimeData(const ZafUTimeData &copy);
```

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

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

This constructor is the persistent constructor. It allocates a new `ZafUTimeData` 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 ZafUTimeData creation techniques.  
ZafUTimeData uTime1(1997, 3, 20, 10, 25);  
ZafUTimeData copyUTime = uTime1;  
ZafUTimeData uTime2("1997 2 14 14 25", "%Y %m %d %H %M");  
ZafUTimeData systemUTime;
```

Destructor

```
virtual ~ZafUTimeData(void);
```

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

Members

BasisYear

```
int BasisYear(void) const;
virtual ZafError SetBasisYear(int basisYear);
```

BasisYear() returns the basis year value of the ZafUTimeData object. By default, the basis year is set to ZAF_BASIS_YEAR, defined in z_untime.hpp. ZAF_BASIS_YEAR is set to 1950 in the shipping ZAF header file, but may be modified by the programmer before rebuilding the libraries for the new value to take effect. **SetBasisYear()** sets the basis year used by the ZafUTimeData object to interpret 2-digit year values.

Clear

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

Clear() sets the value of the ZafUTimeData object to the current system date and time.

Day

```
int Day(void) const;
```

Day() returns the day in the month (1..31) for the ZafUTimeData object.

DayName

```
static ZafIChar ZAF_FARDATA *DayName(void);
```

DayName() returns the name string of the day of the week (e.g., Tuesday) for the ZafUTimeData object. This name string will be the language-specific string set up by the application's language initialization.

DayOfWeek

```
int DayOfWeek(void) const;
```

DayOfWeek() returns the numerical value of the day of the week (Sunday = 1, Monday = 2, ... Saturday = 7) for the ZafUTimeData object.

DayOfYear

```
int DayOfYear(void) const;
```

DayOfYear() returns the numerical value of the day of the year (1..366) for the ZafUTimeData object.

DaysInMonth

```
int DaysInMonth(int month = 0) const;
```

DaysInMonth() returns the number of days in the month corresponding to *month*. If *month* is missing, the number of days in the month represented by the ZafUTimeData object is returned.

DaysInYear

```
int DaysInYear(void) const;
```

DaysInYear() returns the number of days in the year represented by the ZafU-TimeData object.

FormattedText

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

FormattedText() fills *buffer* with a string representation of the ZafUTimeData using the strftime-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* characters. FormattedText() returns the integer value it receives from its call to strftime().

Hour

```
int Hour(void) const;
```

Hour() returns the hour value (0..23) of the ZafUTimeData object.

JulianDay

```
long JulianDay(void) const;  
virtual ZafError SetJulianDay(long jDay);
```

JulianDay() returns the Julian day value of the ZafUTimeData object. SetJulianDay() may be called to set the Julian day of the ZafUTimeData object.

LeapYear

```
bool LeapYear(void) const;
```

LeapYear() returns true if the current year value of the ZafUTimeData object is a leap year. Otherwise, this function returns false.

long

```
operator long();
```

See [Value\(\)](#).

MilliSecond

```
int MilliSecond(void) const;
```

MilliSecond() returns the time value, in milliseconds (0..999), of the ZafU-TimeData object.

Minute

```
int Minute(void) const;
```

Minute() returns the time value, in minutes (0..59), of the ZafUTimeData object.

Month `int Month(void) const;`

Month() returns the numerical month value (January = 1, February = 2, ... December = 12) of the ZafUTimeData object.

MonthName `static ZafIChar ZAF_FARDATA *MonthName(void);`

MonthName() returns the name string for the month value of the ZafUTimeData object. This name string will be the language-specific string set up by the application's language initialization.

Second `int Second(void) const;`

Second() returns the time value, in seconds (0..59), of the ZafUTimeData object.

TimeName `static ZafIChar ZAF_FARDATA *TimeName(void);`

TimeName() returns the name string (e.g., a.m.) for the time value of a ZafUTimeData object. This name string will be the language-specific string set up by the application's language initialization.

SetUTime `virtual ZafError SetUTime(ZafUInt32 seconds);
virtual ZafError SetUTime(int year, int month, int day,
 int hour, int minute, int second, int milliSecond);
virtual ZafError SetUTime(const ZafIChar *buffer, const
 ZafIChar *format);
virtual ZafError SetUTime(const ZafUTimeData &number);`

SetUTime() functions set the value of the ZafUTimeData object from numeric input, another utime, or an interpreted string. Refer to FormattedText() for more information on utime/string conversions.

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

Value() returns the difference in time between the ZafUTimeData object's value and the year 1900. If the time period is at least a day, the return value is measured in Julian days; otherwise the return value is measured in milliseconds. The convenience operator long(), which returns Value(), is more commonly used.

Year `int Year(void) const;`
Year() returns the year value of the ZafUTimeData object.

ZoneOffset `int ZoneOffset(void) const;`
`virtual ZafError SetZoneOffset(int zoneOffset);`
ZoneOffset() returns the timezone offset used by the ZafUTimeData object to interpret time values. SetZoneOffset() sets the timezone offset used by the ZafUTimeData object to interpret time values.

operator - `ZafUTimeData operator-(const ZafUTimeData &utime);`
`ZafUTimeData operator-(long seconds);`
These operators allow simple subtraction operations involving ZafUTimeData objects and longs.

operator -- `ZafUTimeData operator--(void);`
`ZafUTimeData operator--(int);`
These pre- and post-operators decrement the ZafUTimeData object's date and time values by 1.

operator + `ZafUTimeData operator+(const ZafUTimeData &utime2);`
`ZafUTimeData operator+(long seconds);`
These operators allow simple addition operations involving ZafUTimeData objects and longs.

operator ++ `ZafUTimeData operator++(void);`
`ZafUTimeData operator++(int);`
These pre- and post-operators increment the ZafUTimeData object's date and time values by 1.

operator = `ZafUTimeData &operator=(const ZafUTimeData &utime);`
`ZafUTimeData &operator=(long seconds);`
These operators assign the ZafUTimeData object's value to the input value which may be a long or another ZafUTimeData.

operator -=

```
ZafUTimeData &operator-=(const ZafUTimeData &utime);  
ZafUTimeData &operator-=(long seconds);
```

These operators decrement the ZafUTimeData object's value by the input value.

operator +=

```
ZafUTimeData &operator+=(const ZafUTimeData &utime);  
ZafUTimeData &operator+=(long seconds);
```

These operators increment the ZafUTimeData object's value by the input value.

operator < `bool operator<(const ZafUTimeData &utime2);`
This operator returns true if *utime2* is earlier than this ZafUTimeData object.

operator <= `bool operator<=(const ZafUTimeData &utime2);`
This operator returns true if *utime2* is earlier than or the same as this ZafUTimeData object.

operator > `bool operator>(const ZafUTimeData &utime2);`
This operator returns true if *utime2* is later than this ZafUTimeData object.

operator >= `bool operator>=(const ZafUTimeData &utime2);`
This operator returns true if *utime2* is later than or the same as this ZafUTimeData object.

operator == `bool operator==(const ZafUTimeData &utime2);`
This operator returns true if *utime2* is the same as this ZafUTimeData object.

operator != `bool operator!=(const ZafUTimeData &utime2);`
This operator returns true if *utime2* is not the same as this ZafUTimeData object.

ZafVtList

AutoSortData	SetBackgroundColor	SetTextColor
SelectionType	SetFont	

Inheritance	ZafVtList : ZafWindow : (ZafWindowObject : ZafElement), ZafList
Declaration	#include <z_vlist.hpp>
Description	The ZafVtList object is a single-column list object that arranges list items vertically. A ZafVtList object may have a vertical scroll bar associated with it. Like ZafHzList it supports single, multiple and extended selection methods, and list children may not be Noncurrent().
Constructors	All ZafVtList constructors initialize the member variables associated with an instantiated ZafVtList object. The default values set by the ZafVtList 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 ZafVtList. “†” Indicates a blocking function that prevents changes to the attribute in this class.

Member Initializations

ZafVtList

AutoSortData()	false
----------------	-------

ZafWindow

Destroyable()	false [†]
Locked()	false [†]
Maximized()	false [†]
Minimized()	false [†]
Modal()	false [†]
Moveable()	false [†]
NormalHotKeys()	false [†]
Sizeable()	false [†]

ZafWindowObject

Bordered()	true
------------	------

ZafElement

ClassID()	ID_ZAF_VT_LIST
ClassName()	"ZafVtList"

```
ZafVtList(int left, int top, int width, int height);
```

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, ZAF_NULLP(ZafIChar),  
                          bitmap1));
```

```

vList2->Add(new ZafButton(0, 0, 20, 1, ZAF_NULLP(ZafIChar),
    bitmap2));
vList2->Add(new ZafButton(0, 0, 20, 1, ZAF_NULLP(ZafIChar),
    bitmap3));
vList2->Add(new ZafButton(0, 0, 20, 1, ZAF_NULLP(ZafIChar),
    bitmap4));
// 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.

AutoSortData

```

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.

SetBackgroundColor

```

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.

SelectionType `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.

SetFont `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.

SetTextColor `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	GeometryManager	Subtract
AddGenericObjects	HorizontalScrollBar	support
AutomaticUpdate	Locked	SupportCurrent
Border	MaximizeButton	SupportDestroy
BroadcastEvent	Maximized	SupportFirst
Changed	MinimizedButton	SupportLast
ClientRegion	Minimized	SystemButton
CompareAscending	MinimizeIcon	SystemButtonMenu
CompareDescending	Modal	Temporary
CornerScrollBar	Moveable	Text
DefaultButton	NormalHotKeys	Title
Destroy	Owner	VerticalScrollBar
Destroyable	PullDownMenu	operator +
Event	SelectionType	operator -
FocusObject	Sizeable	operator ()

Inheritance `ZafWindow : (ZafWindowObject : ZafElement), ZafList`

Declaration `#include <z_win.hpp>`

Description 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.

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
Modal()	false
Moveable()	true
NormalHotKeys()	false
Owner()	null
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, respectively. 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, "Button",
    ZAF_NULLP(ZafBitmapData)));
// 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,
    zafDefaultDataConstructor, zafDefaultObjectConstructor);
```

```
// 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, "Button",
    ZAF_NULLP(ZafBitmapData)));
...
// 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.

Add

```
virtual ZafWindowObject *Add(ZafWindowObject *object,
    ZafWindowObject *position =
    ZAF_NULLP(ZafWindowObject));
```

operator +

```
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");
```

AddGenericObjects

```
ZafWindow *AddGenericObjects(ZafIChar *title,
    ZafWindowObject *minObject =
    ZAF_NULLP(ZafWindowObject));
```

```
ZafWindow *AddGenericObjects(ZafStringData *title,  
    ZafWindowObject *minObject =  
    ZAF_NULLP(ZafWindowObject));
```

AddGenericObjects() may be called on a ZafWindow to add all the common decorations to it before it has been added to the window manager. The decorations included on a window created with AddGenericObjects() are (added to the window in this order) ZafBorder, ZafMaximizeButton, ZafMinimizeButton, ZafSystemButton, ZafTitle, and optionally a minimize icon (ZafIcon). If AddGenericObjects() is not called to add these objects to a window, the desired objects must be added by the programmer in the proper order (as shown above); if these objects are not added in this order, the window will not appear as expected since they each in turn pare down the client region of the window. *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);
```

AutomaticUpdate

```
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);
}

```

Border

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.

BroadcastEvent

virtual void **BroadcastEvent**(const ZafEventStruct &event,
int levels = 0);

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:

```

ZafEventType MyWindow::Event(const ZafEventStruct &event)
{
    ZafEventType ccode = event.type;
    if (ccode == MY_EVENT)
        BroadcastEvent(event);
    ccode = ZafWindowObject::Event(event);
    return (ccode);
}

```

levels specifies the number of hierarchical levels of children BroadcastEvent() traverses. If *levels* is 0, BroadcastEvent() does not traverse children other than

the window's immediate children. If *levels* is -1, `BroadcastEvent()` traverses all levels of children in a window's hierarchy.

The return values from the children's `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);
}
```

Changed

```
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 it's 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.
}
```

ClientRegion

```
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()`.

CompareAscending

```
static int CompareAscending(ZafWindowObject *object1,  
                             ZafWindowObject *object2);
```

*Compare-
Descending*

```
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();
}
```

CornerScrollBar

```
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.

DefaultButton

```
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, "OK",  
    ZAF_NULLP(ZafBitmapData));  
button1->SetAllowDefault(true);  
window->Add(button1);  
window->SetDefaultButton(button1);
```

Destroy

```
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. See [SupportDestroy\(\)](#) for more information.

Destroyable

```
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()`.

Event

```
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

FocusObject

```
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.

GeometryManager

```
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.

HorizontalScrollBar

```
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.

Locked

```
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()`.

MaximizeButton

```
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.

Maximized

```
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.

MinimizedButton

```
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.

Minimized

```
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.

MinimizeIcon

```
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.

Modal

```
bool Modal(void) const;  
virtual bool SetModal(bool modal);
```

`Modal()` is most commonly used with `ZafDialogWindow` objects, since users are generally only accustomed to modal dialogs. But in special circumstances, a normal `ZafWindow` may be set to be `Modal()`.

If `Modal()` is true, the window will stay in front of all other windows while it is open. Otherwise, the window may be sent to the background by the end user. Commonly, `Modal()` dialogs are used when information must be specified by the end user before the program may continue. This attribute defaults to false,

but it may be changed before the window appears on screen by calling SetModal().

Moveable

```
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().

NormalHotKeys

```
bool NormalHotKeys(void) const;
virtual bool SetNormalHotKeys(bool normalHotKeys);
```

If NormalHotKeys() is true, keystrokes are interpreted as hot keys without an accompanying modifier key (such as <Alt> or <Command>). For example, a NormalHotKeys() calculator window that has child buttons with hot keys may recognize a '1' keystroke without a modifier key to activate the '1' button. The default value of this attribute is false, but it may be changed by calling SetNormalHotKeys().

Owner

```
ZafWindow *Owner(void) const;
virtual ZafWindow *SetOwner(ZafWindow *owner);
```

If Owner() is non-null, Owner() is a pointer to the owner of this window. A window with an Owner() always appears on top of its Owner(), and “follows” its Owner(). For example, when the Owner() is minimized, restored, or killed, this window is also minimized, restored, or killed, respectively. And a window with an Owner() does not appear in the Microsoft Windows task list. The default value of this attribute is null, but it may be changed by calling SetOwner(). Some environments may not support this attribute.

PullDownMenu

```
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.

SelectionType

```
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);
```

Sizeable

```
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()`.

Subtract virtual ZafWindowObject ***Subtract**(ZafWindowObject
 *object);

operator - ZafWindow &**operator-**(ZafWindowObject *object);

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
- string1
- string2
- button1
- button2;
```

support ZafList **support**;

SupportCurrent ZafWindowObject ***SupportCurrent**(void) const;

SupportDestroy void **SupportDestroy**(void);

SupportFirst ZafWindowObject ***SupportFirst**(void) const;

SupportLast 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 examples demonstrate correct use of some of 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;
...
SupportDestroy();
```

SystemButton

`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.

SystemButtonMenu

`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 chil-

dren (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.

Temporary

```
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()`.

Text

```
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.

Title

```
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 sec-

tion 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.

VerticalScrollBar

```
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.

operator +

```
ZafWindow &operator+(ZafWindowObject *object);
```

See [Add\(\)](#).

operator -

```
ZafWindow &operator-(ZafWindowObject *object);
```

See [Subtract\(\)](#).

operator ()

```
ZafWindowObject *operator()(int index);
```

This operator returns the child object corresponding to the zero-based *index* specified.

ZafWindowManager

CaptureMouse	Event	MouseEventRoute
Center	ExitFunction	mouseObject
DefaultEventRoute	focusObject	oldFocusObject
dragObject	helpObject	

Inheritance ZafWindowManager : ZafWindow : (ZafWindowObject : ZafElement), ZafList

Declaration #include <z_win.hpp>

Description 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.

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

DefaultEventRoute()	null
dragObject	null
exitFunction	null
focusObject	null
helpObject	null
MouseEventRoute()	null
mouseObject	null
oldFocusObject	null

ZafWindow

NormalHotKeys()	false [†]
SelectionType()	ZAF_SINGLE_SELECTION [†]

Member Initializations

Temporary() false[†]

ZafWindowObject

AcceptDrop() false[†]
AutomaticUpdate() true[†]
Bordered() false[†]
OSDraw() true[†]
ParentDrawBorder() false[†]
ParentDrawFocus() false[†]
ParentPalette() false[†]
Region() ZAF_PIXEL, 0, 0, Display()->columns
 - 1, Display()->lines - 1
RegionType() ZAF_AVAILABLE_REGION[†]

ZafElement

ClassID() ID_ZAF_WINDOW_MANAGER
ClassName() "ZafWindowManager"

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.

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.

Event

```
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>) if <code>event.windowObject</code> is non-null; otherwise <code>event.windowObject</code> is closed
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 <code>ExitFunction()</code> , it is called first)
S_NEXT_WINDOW	causes the next window to become the top-most

ExitFunction

```
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);
```

focusObject

```
ZafWindowObject *focusObject;
```

The `focusObject` member is used during focus changes and points to the object trying to gain focus. This advanced member is used internally by ZAF, and should not be modified by the programmer.

helpObject

```
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.

mouseObject

```
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.

oldFocusObject

`ZafWindowObject *oldFocusObject ;`

The `helpObject` member is used during focus changes and points to the object losing focus. This advanced member is used internally by ZAF, and should not be modified by the programmer.

ZafWindowObject

AcceptDrop	DrawShadow	ParentDrawFocus
AutomaticUpdate	Duplicate	ParentPalette
BackgroundColor	EndDraw	Previous
BeginDraw	Error	QuickTip
Bordered	Event	Read
Changed	eventManager	Redisplay
ClassID	Focus	RedisplayData
ClassName	FocusObject	Region
ConvertCoordinates	Font	RegionType
ConvertRegion	GetObject	RepeatDelay
ConvertToDrawRegion	HelpContext	RootObject
ConvertToObjectPosition	HelpObjectTip	screenID
ConvertToScreenPosition	InitialDelay	ScrollEvent
ConvertToOSPosition	IsA	Selected
ConvertToZafPosition	LinkDraggable	SupportObject
ConvertToOSRegion	LogicalEvent	SystemObject
ConvertToZafRegion	LogicalPalette	ToggleSelected
ConvertToZafEvent	MemberUserFunction	Text
CoordinateType	MoveDraggable	TextColor
CopyDraggable	MoveEvent	userFlags
DefaultUserFunction	Next	UserFunction
Disabled	Noncurrent	UserInformation
Display	NotifyFocus	userObject
DragDropEvent	NotifySelection	UserPaletteData
Draggable	OSDraw	userStatus
Draw	OSScreenID	Visible
DrawBackground	PaletteState	windowManager
DrawBorder	Parent	Write
DrawFocus	ParentDrawBorder	zafRegion

Inheritance ZafWindowObject : ZafElement

Declaration #include <z_win.hpp>

Description 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.

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

AcceptDrop()	false
AutomaticUpdate()	true
Bordered()	false
Changed	false
CoordinateType()	ZAF_CELL
CopyDraggable	false
Disabled()	false
EditMode()	false
Error()	ZAF_ERROR_NONE
Focus()	false
HelpContext()	NO_HELP_CONTEXT
HelpObjectTip()	null
LinkDraggable()	false
MoveDraggable()	false
memberUserFunction	ZafWindowObject::DefaultUserFunction
Noncurrent()	false
OSDraw()	true
parent	null
ParentDrawBorder()	false
ParentDrawFocus	false
ParentPalette()	false
QuickTip()	null

Member Initializations

<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>
<code>Selected()</code>	<code>false</code>
<code>SupportObject()</code>	<code>false</code>
<code>SystemObject()</code>	<code>true</code>
<code>userObject</code>	<code>null</code>
<code>userFlags</code>	<code>0</code>
<code>userStatus</code>	<code>0</code>
<code>userText</code>	<code>null</code>
<code>UserFunction()</code>	<code>null</code>
<code>UserPaletteData()</code>	<code>null</code>
<code>Visible()</code>	<code>true</code>

ZafElement

<code>ClassID()</code>	<code>ID_ZAF_WINDOW_OBJECT</code>
<code>ClassName()</code>	<code>"ZafWindowObject"</code>

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,
    const ZafIChar *text, ZafBitmapData *zBitmapData,
    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 above.
ZafButton *button = new ZafButton(0, 0, 10, 1, "button");
ZafWindow *window = new ZafWindow(0, 0, 50, 10);
ZafString *string = new ZafString(2, 2, 20, "message1", 100);
```

```
ZafWindowObject(const ZafWindowObject &copy);
```

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 reprint 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,
        zafDefaultDataConstructor, zafDefaultObjectConstructor);
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*AcceptDrop*

```
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(ZafLanguageManager::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	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.
S_DROP_COPY	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.
S_DROP_LINK	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 but should instead take application-specific action.
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() 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);
```

AutomaticUpdate

```
bool AutomaticUpdate(bool traverse = true) 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
- 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.

If *traverse* is true, AutomaticUpdate() ascends the parental tree up to the root window until it finds a value of false (otherwise it returns true). If *traverse* is false, the value of the object's AutomaticUpdate() is returned.

The return value for AutomaticUpdate() and SetAutomaticUpdate() is the final or current update status associated with the object (true or false). Under nor-

mal 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).

BackgroundColor

```
ZafLogicalColor BackgroundColor(ZafLogicalColor color,
                                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.

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
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	

Color Values	Monochrome Values
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.

BeginDraw
EndDraw

```
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);
}
```

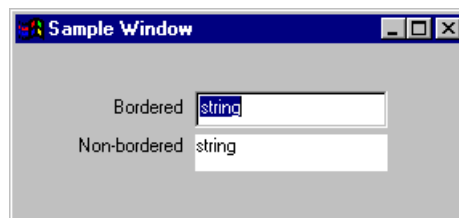
EndDraw() tells the system that you are finished with your drawing operation, and frees up any operating system specific memory used as part of the drawing operation.

Each call to BeginDraw() must be matched with a call to EndDraw(), and BeginDraw()/EndDraw() pairs should not be nested. In other words, a call to BeginDraw() should not be followed by another call to BeginDraw() before calling EndDraw() first. Nested calls do not properly restore display characteristics.

Bordered

```
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. Windows 95 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 top calling SetBordered(true), the bottom SetBordered(false).



Note that the prompts shown to the left of the strings also respond to `SetBordered()` by aligning themselves differently, but do not actually draw borders. By setting the “Bordered” `ZafPrompt` to `SetBordered(true)` and the “Non-bordered” `ZafPrompt` to `SetBordered(false)` the prompts’ text automatically aligns with the text baseline of the `ZafStrings` to their right. Since the text baseline of input fields is different on each environment, this is the preferred method for aligning prompts with input fields.

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.

Changed

```
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.

ClassID

ClassName

```
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 vise-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:

ConvertCoordinates

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

This function converts `zafRegion` using `ConvertRegion()` with the specified coordinate type.

ConvertRegion

```
virtual void ConvertRegion(ZafRegionStruct &region,  
    ZafCoordinateType newType);
```

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, minicell, 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`.

ConvertToDrawRegion

```
virtual ZafRegionStruct ConvertToDrawRegion(const  
    ZafWindowObject *object, const ZafRegionStruct  
    *zafRegion = ZAF_NULLP(ZafRegionStruct)) const;
```

This function converts the object's `zafRegion` to a region compatible with the environment's 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.

ConvertToObjectPosition

```
virtual ZafPositionStruct ConvertToObjectPosition(const  
    ZafPositionStruct screenPosition) const;
```

ConvertToScreenPosition

```
virtual ZafPositionStruct ConvertToScreenPosition(const  
    ZafPositionStruct objectPosition) const;
```

These functions convert a global screen display position (relative to the top left corner of the screen) to a ZAF object position, and vice-versa. These functions are particularly useful when a position must be used in a completely different context. (Programmers may also find `ConvertToScreenPosition` useful when preparing coordinates for a call to a native operating system API.)

For example, a programmer may trap a right-mouse click on a window and desire to display a pop-up menu at the coordinates of the click. Because the original event was trapped at the window level, `LogicalEvent()` will have returned window-based coordinates. The popup menu must be attached to the window manager using screen coordinates, however. `ConvertToScreenPosition()` is the solution. The reverse condition may exist if a position is obtained at the `ZafEventManager` or `ZafWindowManager` level and requires conversion to an object-relative coordinate for use by a window object.

Finally, these functions may be used to convert between two window object coordinate systems by using screen position as a temporary conversion. For example, a group child receives a click which must be used by its parent group's parent window. `ConvertToScreenPosition()` followed by `parent->parent->ConvertToObjectPosition()` will yield the correctly converted coordinate.

`LogicalEvent()` also converts mouse coordinates relative to the calling object. In some cases this may provide a more straight-forward solution. See [Logical-Event\(\)](#) for more information.

*ConvertToOS-
Position*

```
virtual ZafPositionStruct ConvertToOSPosition(const
    ZafWindowObject *object, const ZafPositionStruct
    *zafPosition = ZAF_NULLP(ZafPositionStruct)) const;
```

*ConvertToZaf-
Position*

```
virtual ZafPositionStruct ConvertToZafPosition(const
    ZafWindowObject *object, const ZafPositionStruct
    *osPosition = ZAF_NULLP(ZafPositionStruct)) const;
```

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).

Sometimes these ZAF positions match the OS, other times, they do not. These functions allow Zinc to readjust the position based on potential coordinate shifting needs, including: shadowing, bordering, native vs. non-native OS objects, etc.

*ConvertToOS-
Region*

```
virtual ZafRegionStruct ConvertToOSRegion(const
    ZafWindowObject *object, const ZafRegionStruct
    *zafRegion = ZAF_NULLP(ZafRegionStruct)) const;
```

*ConvertToZaf-
Region*

```
virtual ZafRegionStruct ConvertToZafRegion(const
    ZafWindowObject *object, const ZafRegionStruct
    *osRegion = ZAF_NULLP(ZafRegionStruct)) const;
```

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.

ConvertToZafEvent virtual bool **ConvertToZafEvent**(ZafEventStruct &event);

ConvertToZafEvent() 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 value 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 character in the 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.</p> <p>The pixel conversion values for ZAF_CELL are contained in two ZafDisplay members, cellHeight and cellWidth which are determined at run-time by the ZafDisplay constructor.</p> <pre>int ZafDisplay::cellHeight; int ZafDisplay::cellWidth;</pre>
ZAF_MINICELL	<p>A fractional cell. ZAF defines mini-numerator and denominator values in ZafDisplay. These values are used to subdivide a cell into programmer-defined units. Minicells provide a higher degree of positional control than cells without compromising portability more than necessary.</p> <p>These ZafDisplay values are pre-defined to be the following values:</p> <pre>long ZafDisplay::miniNumeratorX = 1; long ZafDisplay::miniDenominatorX = 10; long ZafDisplay::miniNumeratorY = 1; long ZafDisplay::miniDenominatorY = 10;</pre>

Type	Description
ZAF_PIXEL	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.
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>The pixel conversion values for ZAF_POINTS are contained in two ZafDisplay members, pixelsPerInchX and pixelsPerInchY and are determined at run-time by the ZafDisplay constructor.</p> <pre>long ZafDisplay::pixelsPerInchX; long ZafDisplay::pixelsPerInchY;</pre>
ZAF_TWIPS	Defined to be 1/20 of a point. This value is computed at run-time, based on the value of the ZafDisplay’s pixelsPerInchX and pixelsPerInchY members.

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 `SetCoordinateType()` function, but may be a different value if one of the conditions described above is met.

CopyDraggable

```
bool CopyDraggable(void) const;
bool SetCopyDraggable(bool copyDraggable);
```

See `ZafWindowObject::Draggable()`.

*DefaultUser-
Function*

```
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
N_NON_CURRENT	When Focus() 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 this message.
N_CURRENT	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 this message.
L_SELECT (keyboard)	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 this message, but the contents of the event argument will be event.InputType() == E_KEY, signifying that a key event caused the selection.
L_SELECT (mouse)	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 this 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.
L_DOUBLE_CLICK	User double-clicks on the object. When this occurs and if you specify a user-function, the user-function is called with this 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.

A user function returns 0 if no error occurs; otherwise it returns a non-zero value. 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 =
                (ZafWindowObject::MemberUserFunction)
                &ZafButton::SendMessage;
        else
            memberUserFunction = ZafWindowObject::DefaultUserFunction;
    }

    // Return the current attribute.
    return (sendMessageWhenSelected);
}
```

Disabled

```
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, 1, "Save");
window->Add(button1);
ZafButton *button2 = new ZafButton(22, 2, 20, 1, "Delete");
button2->SetDisabled(true);
```

```
window->Add(button2);

// Check the status of an 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.

Display
display

```
ZafDisplay *Display(void) const;
static ZafDisplay *display;
```

`Display()` returns a pointer to the application’s current display, whether a [ZafScreenDisplay](#) or [ZafPrinter](#). It is initialized when the `ZafWindowManager` constructor is called and should not be modified. All derived window objects use this method when drawing information to the screen. They do not use the global `zafDisplay` or the static display member.

Here is a code snippet that shows how the `Motif ZafButton::Draw()` function uses `Display()`.

```
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--;
```



```

    }
    ...
}

```

display, 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.

DragDropEvent

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

This function is used internally by ZAF as a dispatch function for system defined drag/drop messages and should not normally be overloaded by the programmer. 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 `DragDropEvent()` which may be called by the base `ZafWindowObject` class whenever one of the aforementioned messages is generated.

The programmer should not call `DragDropEvent()`, but it is used internally by ZAF. When deriving to intercept drag/drop events, the programmer should always use the `Event()` function itself.

The return value for `DragDropEvent()` is normally `event.type` if processing is successful. Otherwise, `S_ERROR` or `S_UNKNOWN` may be returned, indicating the object either detected an error on the message, or that the function did not recognize the specified message.

```
Draggable           bool Draggable(void) const;  
CopyDraggable      bool CopyDraggable(void) const;  
                    bool SetCopyDraggable(bool copyDraggable);  
MoveDraggable      bool MoveDraggable(void) const;  
                    bool SetMoveDraggable(bool moveDraggable);  
LinkDraggable      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.

Draw `virtual ZafEventType Draw(const ZafEventStruct &event, ZafEventType ccode);`

DrawBackground `virtual ZafEventType DrawBackground(ZafRegionStruct ®ion, 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. In overloaded `Draw()` implementations, drawing should be encapsulated by `BeginDraw()` and `EndDraw()` calls, and `ZafDisplay::SetPalette()` should be called before using `ZafDisplay`'s drawing primitives. If the object is derived from `ZafWindow`, `Draw()` only updates the client region of the window (`DrawBorder()` is called automatically elsewhere in ZAF). `Draw()` returns `S_ERROR` if some error occurs; otherwise it returns *ccode*.

`DrawBackground()` is called whenever the system needs to 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.

DrawBorder `virtual ZafEventType DrawBorder(ZafRegionStruct ®ion, 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.

DrawFocus

```
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.

DrawShadow

```
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.

Duplicate

```
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, 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());

```

EditMode

```

bool EditMode(void) const;
virtual bool SetEditMode(bool editMode);

```

If `EditMode()` is true, the object acts as if it were being edited by Zinc Designer. The `EditMode()` object may be moved and sized on its parent, and it receives various `D_*` events associated with Zinc Designer that are defined in the header file `z_dsnevt.hpp`. The normal value of this attribute is false, but it may be changed by calling `SetEditMode()`.

The return value for `EditMode()` and `SetEditMode()` is the final, or current edit-mode attribute associated with the object (true or false). Under normal circumstances, this will be the value passed into the `SetEditMode()` function, but may be different if the object does not allow edit mode.

EndDraw

```

void EndDraw(void);

```

See `ZafWindowObject::BeginDraw()`.

Error

```

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.
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!
```


Event

```
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.

Some types of events are dispatched to other functions. These functions are DragDropEvent(), MoveEvent(), and ScrollEvent(). The programmer should not call these specialized functions, but they are used internally by ZAF. When deriving to intercept events, the programmer should always use the Event() function itself.

Here is the specific handling of these types of messages within the ZafWindowObject::Event() function. (See the [Event Definitions](#) Appendix for a complete list of supported events and functionality.)

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 messages and calling routines in the following order: S_INITIALIZE, S_REGISTER_OBJECT, S_COMPUTE_SIZE, OSSize(). This event is received when the object is added to its parent (or a window added to the window manager).
S_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_NON_CURRENT	

Message	Handling
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	
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.

Message	Handling
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.
S_HELP	Causes the help system to be called with a requested help context contained in HelpContext().
S_VSCROLL	These messages are dispatched to ScrollEvent(). For more information, see the ZafWindowObject::ScrollEvent() section of this chapter.
S_HSCROLL	
S_VSCROLL_SET	
S_HSCROLL_SET	
S_VSCROLL_CHECK	
S_HSCROLL_CHECK	
S_VSCROLL_COMPUTE	
S_HSCROLL_COMPUTE	
N_VSCROLL	
N_HSCROLL	
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.

Message	Handling
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.
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	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:
event.position	Contains a ZAF "0,0,left-top" based coordinate that identifies a position within the receiving window object.

Message	Handling
<code>event.rawCode</code>	Contains a combination of the ZAF defined mouse states: <code>M_LEFT</code> , <code>M_LEFT_CHANGE</code> , <code>M_MIDDLE</code> , <code>M_MIDDLE_CHANGE</code> , <code>M_RIGHT</code> , and/or <code>M_RIGHT_CHANGE</code> . These states are converted into OS specific parameters that describe the mouse on the native environment.
<code>event.modifiers</code>	Contains a combination of the ZAF keyboard shift-states including: <code>S_SHIFT</code> , <code>S_RIGHT_SHIFT</code> , <code>S_LEFT_SHIFT</code> , <code>S_CTRL</code> , <code>S_ALT</code> , <code>S_CMD</code> , <code>S_SCROLL_LOCK</code> , <code>S_NUM_LOCK</code> , <code>S_CAPS_LOCK</code> , <code>S_INSERT</code> , and <code>S_OPT</code> .

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());
    }
}
```

eventManager

```
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.

Focus

```
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 window manager's `Focus()` attribute is true when the application is in the foreground, and false when the application is in the background.

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.

FocusObject

```
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.

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());
```

Font

```
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.

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::UserPaletteData\(\)](#).

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.

GetObject

```
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));
```

HelpContext

```
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).

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);
```

HelpObjectTip

```
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. The HelpObjectTip may be displayed on any object that provide Text() and SetText() functions. 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).

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_HELP_TIPS_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, "Save",
    ZAF_NULLP(ZafBitmapData));
save->SetQuickTip("Save the application.");
save->SetHelpObjectTip("Select this button to save the
    application.");
window->Add(save);

ZafButton *cancel = new ZafButton(0, 0, 20, "Cancel",
    ZAF_NULLP(ZafBitmapData));
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.

InitialDelay

```
static int InitialDelay(void);
```

```
static int SetInitialDelay(int initialDelay);
```

RepeatDelay

```
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.

IsA

```
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.”

LinkDraggable

```
bool LinkDraggable(void) const;
```

See ZafWindowObject::Draggable().

LogicalEvent

```
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 event handling functions of Event(), 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).

A partial list of the default event table associated with the ZafWindow and ZafWindowObject classes follows:

```
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.

Coordinate Conversions

In addition to this simple event translation (raw event -> logical mapping), `LogicalEvent()` also causes mouse coordinates to be converted relative to the calling object. For example, a mouse click on a group child generates a raw event that ultimately arrives at the group child. The programmer calls `LogicalEvent(event)` to interpret the event, and in the process the mouse pointer coordinates are converted relative to *this*. If the programmer then needs these coordinates at the parent group level or higher, `parent->LogicalEvent(event)` will remap the coordinates relative to the calling object (“parent” in this case). This remapping of coordinates may be done by any window object up the window object hierarchy, ultimately terminating at the `ZafWindowManager`. The `ZafWindowManager` will map coordinates relative to the `Display()`—the entire screen.

Converting coordinates relative to various objects may be accomplished using other methods as well. These alternatives are particularly useful when converting to or from screen coordinates. See [ConvertToScreenPosition\(\)](#) and [ConvertToObjectPosition\(\)](#) for details.

There are many more details associated with `LogicalEvent()` that are not discussed in this section. These details are necessary for the internal workings of the ZAF libraries, but should not be necessary for full use of ZAF by developers.

As a real-world example, the following code shows how the derived `ZafDate` and `ZafTable` use `LogicalEvent()` to determine 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;
    }
}
```

Each window object may map raw events differently. A complete table of possible “Logical Events” is contained in the “[Event Definitions](#)” appendix. 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.

Logical Event	Description
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 ZafMouse::DoubleClickRate().
L_VIEW	Mapped when an unpressed mouse is positioned over a window object.

LogicalPalette

```
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.

Argument	Description
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.
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.

Argument	Description
ZAF_PM_ANY_TYPE	Specifying this value will match any palette request, as long as the specified state is ZAF_PM_ANY_STATE or the value matches the palette map entry.

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.
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.

Value	Description
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:

Palette 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	These are the foreground/background color pairs used on color systems when drawing the object's visual information.
colorBackground	
monoForeground	These are the foreground/background monochrome color pairs used on black/white systems when drawing the object's visual information.
monoBackground	
font	This is the logical font to be used when rendering text information to the display.
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);
...
}

```

MemberUser- Function

memberUser- Function

```

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. A user function returns 0 if no error occurs; otherwise it returns a non-zero value. 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
{
    ZafDiskFile *database;
public:
    MyButton(void);
    ZafEventType MyNotification(const ZafEventStruct &event,
        ZafEventType ccode);
};

ZafEventType MyButton::MyNotification(const ZafEventStruct
    &event, ZafEventType ccode)
{
    database->Seek(0, ZAF_SEEK_START);
    return (0);
}

ZafEventType MyCallback(MyButton *myButton, ZafEventStruct
    &event, ZafEventType ccode)
{
    myButton->database->Seek(0, ZAF_SEEK_START); // oops!
    database is private!
    return (0);
}

void Initialize(void)
{
    MyButton *button1 = new MyButton;
    button1->userFunction = (ZafUserFunction)MyCallback;
    MyButton *button2 = new MyButton;
    button2->memberUserFunction =
        (ZafWindowObject::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.


```
MoveDraggable    bool MoveDraggable(void) const;
```

See `ZafWindowObject::Draggable()`.

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

The Event() function dispatches some events to MoveEvent() which provides filtered input of movement events. The programmer should not generally overload MoveEvent() but should instead trap all events in Event().

The following events may be dispatched to `MoveEvent()`:

- L_LEFT
- L_RIGHT
- L_UP
- L_DOWN
- L_FIRST
- L_LAST
- L_NEXT
- L_PREVIOUS
- L_PGDN
- L_PGUP.

Whenever an object receives a movement event, the messages are first intercepted in the object's `Event()` function. To allow for more encapsulation and a more efficient handling of similar movement functionality, ZAF defines `MoveEvent()` which may be called by the base `ZafWindowObject` class whenever one of the aforementioned messages is generated.

The programmer should not call `MoveEvent()`, but it is used internally by ZAF. When deriving to intercept movement events, the programmer should always use the `Event()` function itself.

The return value for `MoveEvent()` is normally `event.type` if processing is successful. Otherwise, `S_ERROR` or `S_UNKNOWN` may be returned, indicating the object either detected an error on the message, or that the function did not recognize the specified message.

```
Next      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);
```

Noncurrent

```
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.

NotifyFocus

```

virtual ZafWindowObject *NotifyFocus(ZafWindowObject
    *object, bool focus);

```

This is an advanced function that is used to notify a parent object, or set of hierarchical 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.

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);  
}  
...  
}
```

NotifySelection

```
virtual ZafWindowObject *NotifySelection(ZafWindowObject  
    *object, bool selected);
```

This is an advanced function that is used to notify a parent object, or set of hierarchical 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.

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 `Zaf-`

Window 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.

OSDraw

```
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(ZafIChar),
        ZAF_NULLP(ZafBitmapData))
{
    // 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.

OSScreenID

```

virtual OSWindowID OSScreenID(ZafScreenIDType type =
    ZAF_SCREENID) const;

```

This virtual function provides 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. *type* indicates an OS specific value indicating which type of OS handle is to be returned, and may be any of the following according to OS:

Platform	Possible Values	
MS Windows	ZAF_SCREENID	(lowest level of OS object, HWND)
	ZAF_CLIENTID	(client handle)
	ZAF_FRAMEID	(frame handle)
Motif	ZAF_SCREENID	(lowest level of OS object, Widget)
	ZAF_CLIENTID	(client handle)
	ZAF_FRAMEID	(frame handle)
	ZAF_SHELLID	(shell handle)

Platform	Possible Values
Macintosh	ZAF_SCREENID (lowest level of OS object) ZAF_WINDOWREF (parent window, WindowRef)
OS/2	ZAF_SCREENID (lowest level of OS object, HWND) ZAF_CLIENTID (client handle) ZAF_FRAMEID (frame handle)
DOS	ZAF_SCREENID (lowest level of OS object)

See `ZafWindowObject::screenID` for more information.

PaletteState

```
virtual 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.

Parent

```

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 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()`.

```
ParentDrawBorder    bool ParentDrawBorder(void) const;  
virtual bool SetParentDrawBorder(bool parentDrawBorder);  
ParentDrawFocus    bool ParentDrawFocus(void) const;  
virtual bool SetParentDrawFocus(bool parentDrawFocus);  
ParentPalette      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.

Previous

```
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));

```

QuickTip

```

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).

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, "Save",
    ZAF_NULLP(ZafBitmapData));
save->SetQuickTip("Save the application.");

ZafButton *cancel = new ZafButton(25, 0, 20, "Cancel",
    ZAF_NULLP(ZafBitmapData));
cancel->SetQuickTip("Cancel the save operation.");
```

Read

```
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,
    zafDefaultDataConstructor, zafDefaultObjectConstructor);
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`.

Redisplay void **Redisplay**(void);
RedisplayData 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.

zafRegion ZafRegionStruct **zafRegion**;
Region virtual ZafRegionStruct **Region**(void) const;
 virtual void **SetRegion**(const ZafRegionStruct ®ion);

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 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.

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 on
//   ZafDisplay::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);

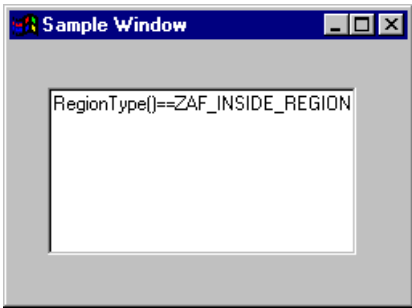
// Center a window on the screen.
zafWindowManager->Center(window);
zafWindowManager->Add(window);
```

RegionType

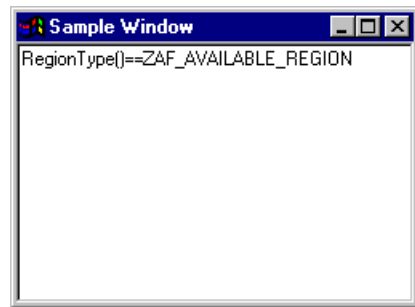
```
ZafRegionType RegionType(void) const;
virtual ZafRegionType SetRegionType(ZafRegionType
    regionType);
```

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 object that uses the ZAF_INSIDE_REGION type:



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.



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.

Region Type	Description
ZAF_OUTSIDE_REGION	<p>Specifies ownership of an area directly outside the specified <code>zafRegion</code>. This setting is similar to <code>ZAF_AVAILABLE_REGION</code>, but the ownership area is defined to be outside the <code>zafRegion</code>, not inside. Currently, only the <code>ZafBorder</code> class uses this type of region.</p> <p>The return value for <code>RegionType()</code> and <code>SetRegionType()</code> is the final, or current attribute associated with the object. Under normal circumstances, this will be the value passed into the <code>SetRegionType()</code> 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_INSIDE_REGION) zafRegion = parent ? parent- >MaxRegion(this) : windowManager- >MaxRegion(this); break; }</pre>

RepeatDelay

```
static int RepeatDelay(void);
```

See ZafWindowObject::InitialDelay().

RootObject

```
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());
```

screenID

```
OSWindowID screenID;
```

This variable provides 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().
OS/2	Represents an HWND. For example, the OS/2 implementation of ZafString creates an “EDIT” with a call to the OS/2 API CreateWindowEx().
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. See `ZafWindowObject::OSScreenID()` for more information.

Another possible use of screenID is to check if an object is on the screen or not. Since the `Visible()` attribute may be set to true before the object’s parent window is actually added to the screen, `Visible()` may be true before the object is really on the screen. On the other hand, none of a parent window’s children receive screenIDs until the parent window is added to the screen. So if screenID is null the object is not on the screen yet. See `Visible()` for more information.

Here are 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);
    }
}

```

ScrollEvent

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

The Event() function may dispatch some events to ScrollEvent which provides filtered input of scrolling events. The programmer should not overload ScrollEvent, but instead trap all events in Event().

The following events may be dispatched to ScrollEvent():

- 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 scrolling messages, the messages are first intercepted in the object's Event() function. To allow for more encapsulation and a

more efficient handling of similar scrolling functionality, ZAF defines `ScrollEvent()` which may be called by the base `ZafWindowObject` class whenever one of the aforementioned messages is generated.

The programmer should not call `ScrollEvent()`, but it is used internally by ZAF. When deriving to intercept scrolling events, the programmer should always use the `Event()` function itself.

The return value for `ScrollEvent()` is normally `event.type` if processing is successful. Otherwise, `S_ERROR` or `S_UNKNOWN` may be returned, indicating the object either detected an error on the message, or that the function did not recognize the specified message.

Selected

```
bool Selected(void) const;  
virtual bool SetSelected(bool selected);  
ToggleSelected 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 finishes 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();
```

SupportObject

```
bool SupportObject(void) const;
virtual bool SetSupportObject(bool supportObject);
```

An object with the `SupportObject()` attribute set is placed in the *support* list of its parent by the parent's `Add()` method. As such, a `SupportObject()` is not considered a regular child of the parent. Some common support objects of a `ZafWindow` object are `ZafBorder`, `ZafTitle` and `ZafScrollBar`. Support objects reserve their space on the parent window before the regular children occupy the window's client region. In other words, `SupportObject()` causes the client region of the parent window to shrink according to the size of the object. This attribute defaults to false, but `SetSupportObject()` may be used to modify it before the object is added to its parent. Most objects that need to, set this attribute to true by default, so the programmer generally does not call the advanced method `SetSupportObject()`.

SystemObject

```
bool SystemObject(void) const;
virtual bool SetSystemObject(bool systemObject);
```

An object with the `SystemObject()` attribute set will cause a corresponding operating system object to be created, if possible. For example, a `SystemObject()` `ZafButton` on Macintosh will request the Control Manager to create a corresponding native control object. On the other hand, if `SystemObject()` is false, no native object is created for the `ZafWindowObject` (such as for list items). This attribute defaults to true, but `SetSystemObject()` may be used to modify it. Most objects that need to, set this attribute to false by default, so the programmer generally does not call the advanced method `SetSystemObject()`.

Text

```
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.

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 <code>SetText()</code> was accepted without any errors.
ZAF_ERROR_INVALID	The value passed to <code>SetText()</code> was invalid for the receiving object. An example of this type of error would be passing “10:00am” to a <code>ZafDate</code> 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("");

```

TextColor

```

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.

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::UserPaletteData()`.

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.

```
userFlags
typedef ZafUInt16 ZafFlags;
ZafFlags userFlags;
```

`userFlags` is a member of `ZafWindowObject` provided exclusively for the programmer's use, and is only initialized to 0 at the `ZafWindowObject` level. Since `userFlags` is defined at the `ZafWindowObject` level, it may be useful for storing user-defined bit flags in any object derived from `ZafWindowObject` (including built-in ZAF objects).

```
typedef ZafEventType (*ZafUserFunction)(ZafWindowObject
    *, ZafEventStruct &, ZafEventType);
```

userFunction
UserFunction

```
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. `ZafString` and its derivatives do not call the user function with mouse clicks.
- 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. `ZafString` and its derivatives do not call the user function with mouse clicks.

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);
```

UserInformation

```
virtual void *UserInformation(ZafInfoRequest request,
    void *data, ZafClassID classID = ID_DEFAULT);
```

UserInformation() is a virtual function provided for the programmer's use, and is a stub at the ZafWindowObject level. Since UserInformation() is defined at the ZafWindowObject level, it may be useful for getting class-specific information from a derived object through a ZafWindowObject pointer to the object. To do so, UserInformation() definitions may be provided by the programmer in derived classes. ZafInfoRequest is an unsigned int.

userObject

```
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 ((MyDatabase *)userObject); }
    MyDatabase *SetDatabase(MyDatabase *database)
    { userObject = database; return (database); }
};

// User callback implementation.
ZafEventType MyUserFunction(ZafWindowObject *object,
    ZafEventStruct &, ZafEventType ccode)
{
    MyDatabase *database = (MyDatabase *)object->userObject;
    ...
    return (0);
}
```

userPaletteData

UserPaletteData

```
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.
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	These are the foreground/background color pairs used on color systems when drawing the object’s visual information.
colorBackground	
monoForeground	These are the foreground/background monochrome color pairs used on black/white systems when drawing the object’s visual information.
monoBackground	
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.

Value	Definition
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.
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.

Value	Definition
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.

Value	Definition
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, "button",
    ZAF_NULLP(ZafBitmapData));

// 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 that `SetUserPaletteData()` 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 `UserPaletteData()` 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 `UserPaletteData()` and `SetUserPaletteData()` is the current user palette data associated with the object. This will always be the color value passed into the `SetUserPaletteData()`.

```
typedef ZafUInt16 ZafStatus;
ZafStatus userStatus;
```

userStatus

`userStatus` is a member of `ZafWindowObject` provided exclusively for the programmer’s use, and is only initialized to 0 at the `ZafWindowObject` level. Since `userStatus` is defined at the `ZafWindowObject` level, it may be useful for

storing user-defined status values in any object derived from ZafWindowObject (including built-in ZAF objects).

Visible

```
bool Visible(bool traverse = true) 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, "No Error",
    ZAF_NULLP(ZafBitmapData));
window->Add(button1);
```

```
ZafButton *button2 = new ZafButton(2, 2, 20, "Kill
    Application!", ZAF_NULLP(ZafBitmapData));
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.

Since the `Visible()` attribute may be set to true before the object's parent window is actually added to the screen, `Visible()` may be true before the object is really on the screen. One way to see if the object is on the screen or not is to check its `screenID`. See [screenID](#) for more information.

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.

If *traverse* is true, `Visible()` ascends the parental tree up to the root window until it finds a value of false (otherwise it returns true). If *traverse* is false, the value of the object's `Visible()` is returned.

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()`.

windowManager

```
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.

Write

```
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,
    zafDefaultDataConstructor, zafDefaultObjectConstructor);
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 sys-

tem 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, "OK",
    ZAF_NULLP(ZafBitmapData));
button->SetStringID("MyButton");
button->Event(S_INITIALIZE);

ZafStorage *storage = new ZafStorage("myfile.dat");
ZafObjectPersistence persist(storage,
    zafDefaultDataConstructor, zafDefaultObjectConstructor);
storage->ChDir(~ZafWindow~MyWindow);
button->Write(persist);
```

For in-depth information on object persistence, refer to the source code for ZafWindow::Write() and the ZafWindowObject constructor section of this chapter.

zafRegion

ZafRegionStruct **zafRegion**;

See [Region\(\)](#).

Function Reference

ZafAbs

```
inline int ZafAbs(int value);
```

Declaration `#include <z_utils.hpp>`

Description `ZafAbs()` returns the absolute value of *value*.

The following code snippet shows how to use `ZafAbs()`:

```
int absolute = ZafAbs(myInteger);  
// Do some calculation with the absolute value of myInteger.
```

ZafCrNlToCr

```
ZAF_EXPORT ZafIChar *ZafCrNlToCr(ZafIChar *src, ZafIChar
    *dst = ZAF_NULLP(ZafIChar));
```

Declaration `#include <z_utils.hpp>`

Description `ZafCrNlToCr()` converts all the `'\r','\n'` pairs in *src* to `'\r'` characters. If *dst* is null, the converted string is returned in *src*. Otherwise, the converted string is returned in *dst*. An example of when this is useful is when converting a ZAF string to a native Mac OS string.

A pointer to the converted string is returned.

The following code snippet shows how to use `ZafCrNlToCr()`:

```
// Convert the text to be written to a native text file.
ZafIChar *text = strdup(object->Text());
ZafCrNlToCr(text);
WriteTextToNativeFile(text);
```

ZafCrNIToNI

```
ZAF_EXPORT ZafIChar *ZafCrNlToNl(ZafIChar *src, ZafIChar
    *dst = ZAF_NULLP(ZafIChar));
```

Declaration `#include <z_utils.hpp>`

Description `ZafCrNlToNl()` converts all the `'\r','\n'` pairs in *src* to `'\n'` characters. If *dst* is null, the converted string is returned in *src*. Otherwise, the converted string is returned in *dst*. An example of when this is useful is when converting a ZAF string to a native Unix string.

A pointer to the converted string is returned.

The following code snippet shows how to use `ZafCrNlToNl()`:

```
// Convert the text to be written to a native text file.
ZafIChar *text = strdup(object->Text());
ZafCrNlToNl(text);
WriteTextToNativeFile(text);
```

ZafCrToCrNl

```
ZAF_EXPORT ZafIChar *ZafCrToCrNl(ZafIChar *src, ZafIChar
    *dst = ZAF_NULLP(ZafIChar));
```

Declaration `#include <z_utils.hpp>`

Description `ZafCrToCrNl()` converts all the `'\r'` characters in *src* to `'\r','\n'` pairs. If *dst* is null, the converted string is returned in *src*. Otherwise, the converted string is returned in *dst*. The return buffer must have been previously allocated large enough to fit the original string along with the inserted `'\n'` characters. An example of when this is useful is when converting a native Mac OS string to a ZAF string.

A pointer to the converted string is returned.

The following code snippet shows how to use `ZafCrToCrNl()`:

```
// Read the text and convert it for use by the ZAF object.
ZafIChar text[MAX_FILE_LENGTH];
ReadTextFromNativeFile(text);
ZafCrToCrNl(text);
object->SetText(text);
```

DynamicPtrCast

```
#define DynamicPtrCast(arg, type)
```

Declaration

```
#include <z_list.hpp>
```

Description

DynamicPtrCast() is a macro provided by ZAF that aids in supporting RTTI. If a compiler doesn't support RTTI via `dynamic_cast()`, DynamicPtrCast() provides similar functionality for classes derived from ZafElement by calling `ZafElement::EvaluateIsA()`. For compilers that support RTTI, DynamicPtrCast() simply calls `dynamic_cast()`. For compilers that don't support RTTI, DynamicPtrCast() only works with non-const types. See `ZafElement::EvaluateIsA()` for more information.

The following code snippet shows how to use DynamicPtrCast():

```
ZafEventType MyClass::Event(const ZafEventStruct &event)
{
    ZafEventType ccode = LogicalEvent(event);
    switch (ccode)
    {
        ...
        case MY_STRING_EVENT:
        {
            // We know event.windowObject is a ZafString, since
            // the application created the event elsewhere.
            ZafString *myString = DynamicPtrCast(myString, ZafString);
            DoSomethingWithString(myString);
        }
        break;
        ...
    }
    return (ccode);
}
```

ZafMax

```
inline int ZafMax(int arg1, int arg2);
```

Declaration `#include <z_utils.hpp>`

Description `ZafMax()` returns the maximum value of *arg1* and *arg2*.

The following code snippet shows how to use `ZafMax()`:

```
int biggest = ZafMax(myInteger1, myInteger2);  
// Do some calculation with the biggest integer.
```

ZafMin

```
inline int ZafMin(int arg1, int arg2);
```

Declaration `#include <z_utils.hpp>`

Description `ZafMin()` returns the minimum value of *arg1* and *arg2*.

The following code snippet shows how to use `ZafMin()`:

```
int smallest = ZafMin(myInteger1, myInteger2);  
// Do some calculation with the smallest integer.
```

ZafNlToCrNl

```
ZAF_EXPORT ZafIChar *ZafNlToCrNl(ZafIChar *src, ZafIChar
    *dst = ZAF_NULLP(ZafIChar));
```

Declaration `#include <z_utils.hpp>`

Description `ZafNlToCrNl()` converts all the `'\n'` characters in *src* to `'\r','\n'` pairs. If *dst* is null, the converted string is returned in *src*. Otherwise, the converted string is returned in *dst*. The return buffer must have been previously allocated large enough to fit the original string along with the inserted `'\r'` characters. An example of when this is useful is when converting a native Unix string to a ZAF string.

A pointer to the converted string is returned.

The following code snippet shows how to use `ZafNlToCrNl()`:

```
// Read the text and convert it for use by the ZAF object.
ZafIChar text[MAX_FILE_LENGTH];
ReadTextFromNativeFile(text);
ZafNlToCrNl(text);
object->SetText(text);
```


ZafRegisterMouse

```
bool ZafRegisterMouse(ZafWindowObject *object, bool view
    = false, bool leftMouse = true, bool rightMouse =
    true);
```

Declaration `#include <m_utils.hpp>`

Description `ZafRegisterMouse()` is a function defined for Motif only. Normally, mouse movement events are not received in a Motif application. In the event that these events are desired for some objects, `ZafRegisterMouse()` may be called to allow mouse movement events. `ZafRegisterMouse()` must be called after an object has been registered (via the `S_REGISTER_OBJECT` event).

object is a pointer to the ZAF object to receive the events. Passing true in *view* will cause the object to receive `L_VIEW` events. Passing true in *leftMouse* causes the object to receive `L_CONTINUE_SELECT` events. Passing true in *rightMouse* causes the object to receive `L_CONTINUE_ESCAPE` events. Passing false into any of these latter three parameters causes the corresponding event not to be received by the object.

`ZafRegisterMouse()` returns false if the operation did not complete successfully (such as when called before the object has been registered); otherwise true is returned.

The following code snippet shows how to use `ZafRegisterMouse()` in the `S_REGISTER_OBJECT` case of a derived button's `Event()` method:

```
...
case S_REGISTER_OBJECT:
    // Pass it to immediate base class first for registration.
    ZafButton::Event(event);

    // Cause this object to receive L_VIEW events.
    ZafRegisterMouse(this, true, false, false);
    break;
...
```

ZafStrColl

```
int ZAF_EXPORT ZafStrColl(const ZafIChar *s1, const
    ZafIChar *s2);
```

Declaration `#include <z_string.hpp>`

Description ZafStrColl() does a comparison of the strings *s1* and *s2*, respecting current locale settings.

ZafStrColl() returns 0 if the strings match. It returns some value less than 0 if the first character in *s1* that doesn't match has an ISO value less than the corresponding character in *s2*, or it returns some value greater than 0 if the first character in *s1* that doesn't match has an ISO value greater than the corresponding character in *s2*.

ZafStrColl() provides the same functionality for ZafIChar strings as the ANSI `strcoll()` function for char strings. If the library has been compiled in ISO mode (see the header file `z_env.hpp`), a macro is defined for `strcoll()` to call `ZafStrColl()` so that the programmer may internationalize source code more easily.

The following code snippet shows how to use `ZafStrColl()`:

```
if (ZafStrColl(object1->Text(), object2->Text()) == 0)
    return (THE_STRINGS_MATCH);
```

ZafStrdup

```
ZAF_EXPORT ZafIChar *ZafStrdup(const ZafIChar
                               *srcString);
```

Declaration `#include <z_string.hpp>`

Description `ZafStrdup()` allocates a buffer, copies *srcString* into it, and returns a pointer to the buffer. The programmer must delete the buffer when done with it. The delete operator must be used with brackets when deleting the buffer.

`ZafStrdup()` provides the same functionality for a `ZafIChar` string as the ANSI `strdup()` function for a `char` string. If the library has been compiled in ISO mode or Unicode mode (see the header file `z_env.hpp`), a macro is defined for `strdup()` to call `ZafStrdup()` so that the programmer may internationalize source code more easily.

The following code snippet shows how to use `ZafStrdup()`:

```
ZafIChar *newBuffer = ZafStrdup(object->Text());
// Use the copy of the object's text.
delete []newBuffer;
```

streq

```
ZAF_EXPORT int streq(const ZafIChar *s1, const ZafIChar  
    *s2);
```

Declaration `#include <z_string.hpp>`

Description `streq()` determines if strings *s1* and *s2* are equal.

`streq()` returns 0 if the strings match. It returns some value less than 0 if the first character in *s1* that doesn't match has an ISO or Unicode value less than the corresponding character in *s2*, or it returns some value greater than 0 if the first character in *s1* that doesn't match has an ISO or Unicode value greater than the corresponding character in *s2*.

`streq()` provides the same functionality for ZafIChar strings as the ANSI `strcmp()` function for char strings.

The following code snippet shows how to use `streq()`:

```
if (streq(object1->Text(), object2->Text()) == 0)  
    return (THE_STRINGS_MATCH);
```

ZafStricmp

```
ZAF_EXPORT int ZafStricmp(const ZafIChar *a, const
                          ZafIChar *b);
```

Declaration `#include <z_string.hpp>`

Description `ZafStricmp()` does a case-insensitive comparison of the strings *a* and *b*.

`ZafStricmp()` returns 0 if the strings match. It returns some value less than 0 if the first character in *a* that doesn't match has an ISO or Unicode value less than the corresponding character in *b*, or it returns some value greater than 0 if the first character in *a* that doesn't match has an ISO or Unicode value greater than the corresponding character in *b*.

`ZafStricmp()` provides the same functionality for `ZafIChar` strings as the ANSI `stricmp()` function for `char` strings. If the library has been compiled in ISO mode or Unicode mode (see the header file `z_env.hpp`), a macro is defined for `stricmp()` to call `ZafStricmp()` so that the programmer may internationalize source code more easily.

The following code snippet shows how to use `ZafStricmp()`:

```
if (ZafStricmp(object1->Text(), object2->Text()) == 0)
    return (THE_STRINGS_MATCH);
```

ZafStrlwr

```
ZAF_EXPORT ZafIChar *ZafStrlwr(ZafIChar *string);
```

Declaration `#include <z_string.hpp>`

Description `ZafStrlwr()` converts uppercase characters in *string* to lowercase, and returns a pointer to *string*.

`ZafStrlwr()` provides the same functionality for a `ZafIChar` string as the ANSI `strlwr()` function for a `char` string. If the library has been compiled in ISO mode or Unicode mode (see the header file `z_env.hpp`), a macro is defined for `strlwr()` to call `ZafStrlwr()` so that the programmer may internationalize source code more easily.

The following code snippet shows how to use `ZafStrlwr()`:

```
ZafIChar *myString = ZafStrdup(object->Text());
ZafStrlwr(myString);
// Use the lowercase string.
delete []myString;
```

strneq

```
ZAF_EXPORT int strneq(const ZafIChar *s1, const ZafIChar
    *s2, int n);
```

Declaration `#include <z_string.hpp>`

Description `strneq()` determines if strings *s1* and *s2* are equal, comparing up to *n* characters.

`strneq()` returns 0 if the characters compared all match. It returns some value less than 0 if the first character in *s1* that doesn't match has an ISO or Unicode value less than the corresponding character in *s2*, or it returns some value greater than 0 if the first character in *s1* that doesn't match has an ISO or Unicode value greater than the corresponding character in *s2*.

`strneq()` provides the same functionality for ZafIChar strings as the ANSI `strncmp()` function for char strings.

The following code snippet shows how to use `strneq()`:

```
if (strneq(object1->Text(), object2->Text(), 2) == 0)
    return (PARTIAL_MATCH);
```


strnicmp

```
ZAF_EXPORT int strnicmp(const ZafIChar *a, const ZafIChar  
    *b, int n);
```

Declaration `#include <z_string.hpp>`

Description `strnicmp()` does a case-insensitive comparison of the strings *a* and *b*, comparing up to *n* characters.

`strnicmp()` returns 0 if the characters compared all match. It returns some value less than 0 if the first character in *a* that doesn't match has an ISO or Unicode value less than the corresponding character in *b*, or it returns some value greater than 0 if the first character in *a* that doesn't match has an ISO or Unicode value greater than the corresponding character in *b*.

The following code snippet shows how to use `strnicmp()`:

```
if (strnicmp(object1->Text(), object2->Text(), 2) == 0)  
    return (PARTIAL_MATCH);
```

strrepc

```
ZAF_EXPORT void strrepc(ZafIChar *string, ZafIChar c,  
                        ZafIChar repc);
```

Declaration `#include <z_string.hpp>`

Description `strrepc()` replaces all the characters in *string* that match *c* with *repc*.

The following code snippet shows how to use `strrepc()`:

```
// Change "Best Brain" to "Test Train".  
strrepc("Best Brain", 'B', 'T');
```

rstrip

```
ZAF_EXPORT void rstrip(ZafIChar *string, ZafIChar c);
```

Declaration `#include <z_string.hpp>`

Description `rstrip()` deletes all the characters in *string* that match *c*.

The following code snippet shows how to use `rstrip()`:

```
// Delete all the '\n' and '\r' characters from myString.  
extern ZafIChar *myString;  
rstrip(myString, '\n');  
rstrip(myString, '\r');
```

ZafStrupr

```
ZAF_EXPORT ZafIChar *ZafStrupr(ZafIChar *string);
```

Declaration

```
#include <z_string.hpp>
```

Description

ZafStrupr() converts lowercase characters in *string* to uppercase, and returns a pointer to *string*.

ZafStrupr() provides the same functionality for a ZafIChar string as the ANSIstrupr() function for a char string. If the library has been compiled in ISO mode or Unicode mode (see the header file z_env.hpp), a macro is defined forstrupr() to call ZafStrupr() so that the programmer may internationalize source code more easily.

The following code snippet shows how to use ZafStrupr():

```
ZafIChar *myString = ZafStrdup(object->Text());  
ZafStrupr(myString);  
// Use the uppercase string.  
delete []myString;
```

ZafStrXFrm

```
int ZAF_EXPORT ZafStrXFrm(ZafIChar *s1, const ZafIChar
    *s2, int n);
```

Declaration `#include <z_string.hpp>`

Description `ZafStrXFrm()` transforms up to n characters in $s2$ into $s1$, such that resulting strings passed into `strcmp()` and `strcoll()` would return the same value. The resulting string is a tokenized string providing weighting information for use in string collation. $s1$ must have been allocated by the programmer large enough to hold the resulting tokenized string. Each tokenized character may be up to 4 times the size of the source character, depending on weighting rules for the character.

`ZafStrXFrm()` returns the number of characters copied not including the null terminator. If the return value is greater than or equal to n , nothing was copied into $s1$.

`ZafStrXFrm()` provides the same functionality for `ZafIChar` strings as the ANSI `strxfrm()` function for `char` strings. If the library has been compiled in ISO mode (see the header file `z_env.hpp`), a macro is defined for `strxfrm()` to call `ZafStrXFrm()` so that the programmer may internationalize source code more easily.

The following code snippet shows how to use `ZafStrXFrm()`:

```
ZafIChar result = new ZafIChar[4 * strlen(object->Text())];
ZafStrXFrm(result, object->Text(), strlen(object->Text()));
// Use the result.
delete []result;
```

WildStrcmp

```
ZAF_EXPORT int WildStrcmp(const ZafIChar *str, const
                          ZafIChar *pattern);
```

Declaration `#include <z_utils.hpp>`

Description WildStrcmp() does a case-insensitive comparison of a string to a pattern that may include wildcards. *string* is the string to be compared, and *pattern* is the pattern to be compared against.

A wildcard may be either '*' or '?'. A '*' in the pattern matches any 0 or more characters in the string, and a '?' in the pattern matches any 1 character in the string. Both '*' and '?' may be embedded in the middle of the string.

WildStrcmp() returns 0 if the string matches the pattern. It returns some value less than 0 if the first character in the string that doesn't match has an ISO or Unicode value less than the corresponding character in the pattern, or it returns some value greater than 0 if the first character in the string that doesn't match has an ISO or Unicode value greater than the corresponding character in the pattern.

The following code snippet shows how to use WildStrcmp():

```
if (WildStrcmp("String999Beans", "String*Beans") == 0)
    return (true);
...
if (WildStrcmp("Model36X", "Model??X") != 0)
    return (NOT_A_MODEL_X);
```

Utility Reference

Convert

“Convert” provides support for converting ZAF 4.x persistent object data files (such as those created with Zinc Designer) to their ZAF version 5 equivalents. Convert persistent data files is one of three major steps in converting a ZAF 4.x project to ZAF 5. (The other steps are running “Rep4to5” and manually converting logic). Convert is not needed when creating new ZAF 5 applications.

Usage

convert [source file] [target file]

Convert may be passed two command line parameters to specify the original ZAF 4.x data file and the ZAF 5 output file name. If the file names are not specified, the user may provide them using the supplied graphical user interface. The ZAF 4.x data file name is specified for “Source file:” and the output file name is specified for “Destination file:”. To begin the conversion process, select the “Convert” button. The names of the objects converted will appear in the scrolling text field.

Convert has limitations:

- User data is not converted. For example, if the programmer has derived an object and overloaded the ZAF 4 Store() function to store custom data, this data cannot be automatically converted. If customer data is encountered during the conversion process, the converter may terminate unexpectedly, or the data file may not be properly converted.
- Only data file directories containing “known” data are searched during the conversion process. If the programmer has stored user data or Zinc objects in custom directories, the information will not be converted. However, these unknown directories will not affect the conversion of other directories.
- After conversion, objects may be stored in different directory names. The programmer should examine the file using Zinc Designer to become familiar with the new structure. Note that different window types (normal, dialog, scrolled, etc.) are stored in different directories.

Source code (convert.cpp) is supplied, and may be modified to handle special cases.

Rep

“Rep” provides support for replacing strings in a file, and can be useful when changing class names or other symbols in source code files. A modification of this utility, Rep4to5, is available to automatically replace many of the ZAF 4.x symbols with their version 5 equivalents. See [Rep4to5](#) for more information.

Rep modifies the source file in place. Backup of the source file is therefore recommended.

Usage

```
rep [options] <file> <oldString,newString>
    [oldString,newString [...]]
```

The command line options for the Rep utility are as follows:

Option	Description
/word	Match complete words only
/force	Force replacement without asking the user to verify

Some examples of using the ZMake utility follow:

Command Line	Description
rep *.?pp dog, rat	“dog” is replaced with “rat”, “dogbert” is replaced with “ratbert”, each replacement is verified with “y” or “n”. All occurrences of “dog” are replaced whether complete words or portions of words.
rep /word /force *.?pp dog, rat cat, mouse	“dog” is replaced with “rat”, “cat” is replaced with “mouse”, “dogbert” remains “dogbert”, replacements are made without verification.

Rep4to5

“Rep4to5” automatically replaces most of the ZAF 4.x symbols to their version 5 equivalents. Rep4to5 is one step in converting a ZAF 4.x project to ZAF 5. (Other steps include running “Convert” and manually converting program logic).

Rep4to5 is a simple string replacement utility. As such, it does not reorder parameters or handle other sophisticated code conversion tasks. It is intended only to provide a quick head start to the code conversion process by pointing the programmer in the right direction.

“Rep4to5” is a customized version of the ZAF 5 “Rep” utility.

Usage

rep4to5 <file>

file specifies the filename to be converted. This file is modified in place and should be backed up before converting.

Rep4to5 may be customized by modifying *replacementArray[]* in *rep4to5.cpp* and rebuilding the application. The programmer may modify which symbols are replaced, and their replacements. The first field in an element of *replacementArray[]* is the old symbol, and the second field is the replacement symbol. The array must be terminated with “{ 0, 0 }”.

Conversions

The pairs of strings used by rep4to5 is listed below as a ZAF 4 alphabetical reference to new names used in ZAF 5.

You will notice that many ZAF 4 strings, particularly flag names, cannot be directly converted to ZAF 5. In ZAF 4 these flags were used both to *set* and *get* values. In ZAF 5 setting of properties is accomplished using *Set** functions while querying the same properties use different accessor functions. In these cases, where the original string was ambiguous, Rep4to5 adds a comment to the converted code to help identify the proper ZAF 5 member. It is left to the programmer to complete the conversion.

ZAF 4 symbol (old)	ZAF 5 replacement
_classID	classID
_className	className
AcceptsDrop	AcceptDrop
AllowsCheckMark	AllowToggling
AllowsDragSelection	
AllowsMultipleSelection	AllowsMultipleSelection /* SelectionType() == ZAF_MULTIPLE_SELECTION */

ZAF 4 symbol (old)	ZAF 5 replacement
AllowsToggling	AllowToggling
ALT_STATE	ZafDevice::ZafAltState
AppliesToOppositeSide	OppositeSide
ATCF_BOTTOM	ATCF_BOTTOM /* Type() == ZAF_ATCF_BOTTOM */
ATCF_LEFT	ATCF_LEFT /* Type() == ZAF_ATCF_LEFT */
ATCF_OPPOSITE	ATCF_OPPOSITE /* OppositeSide() */
ATCF_RIGHT	ATCF_RIGHT /* Type() == ZAF_ATCF_BOTTOM */
ATCF_STRETCH	ATCF_STRETCH /* Stretch() */
ATCF_TOP	ATCF_TOP /* Type() == ZAF_ATCF_TOP */
AutoRepeatsSelection	AutoRepeatSelection
AutoSizes	AutoSize
AutoSortsData	AutoSortData
BTF_AUTO_SIZE	BTF_AUTO_SIZE /* AutoSize() */
BTF_CHECK_BOX	BTF_CHECK_BOX /*ButtonType() == ZAF_CHECK_BOX */
BTF_DEFAULT_BUTTON	BTF_DEFAULT_BUTTON /* parent->DefaultButton() */
BTF_DOUBLE_CLICK	BTF_DOUBLE_CLICK /* SelectOnDoubleClick() */
BTF_DOWN_CLICK	BTF_DOWN_CLICK /* SelectOnDownClick() */
BTF_NO_3D	BTF_NO_3D /* ButtonType() == ZAF_FLAT_BUTTON */
BTF_NO_TOGGLE	BTF_NO_TOGGLE /* !AllowsToggling() */
BTF_RADIO_BUTTON	BTF_RADIO_BUTTON /* ButtonType() == ZAF_RADIO_BUTTON */
BTF_REPEAT	BTF_REPEAT /* AutoRepeatSelection() */
BTF_SEND_MESSAGE	BTF_SEND_MESSAGE /* SendMessageWhenSelected() */
BTF_STATIC_BITMAPARRAY	BTF_STATIC_BITMAPARRAY /* bitmapData->staticArray */
BUTTON_TYPE	ZafButtonType

ZAF 4 symbol (old)	ZAF 5 replacement
BUTTON_TYPE	BUTTON_TYPE /* ButtonType() */
DEVICE_IMAGE	ZafDeviceImage
DeviceImage	SetDeviceImage
DevicePosition	SetDevicePosition
DeviceState	SetDeviceState
EVENT_TYPE	ZafEventType
FALSE	false
Get	Get /*Possibly GetObject*/
ICF_DOUBLE_CLICK	ICF_DOUBLE_CLICK /* SelectOnDoubleClick() */
ICF_MINIMIZE_OBJECT	ICF_MINIMIZE_OBJECT /* IconType() == ZAF_MINIMIZE_ICON */
ICF_STATIC_ICONARRAY	ICF_STATIC_ICONARRAY /* IconData()->staticArray*/
ICHAR_T	ZafIChar
ID_ATTACHMENT	ID_ZAF_ATTACHMENT
ID_BIGNUM	ID_ZAF_BIGNUM
ID_BORDER	ID_ZAF_BORDER
ID_BUTTON	ID_ZAF_BUTTON
ID_CHECK_BOX	ID_ZAF_CHECK_BOX
ID_COMBO_BOX	ID_ZAF_COMBO_BOX
ID_CONSTRAINT	ID_ZAF_CONSTRAINT
ID_DATE	ID_ZAF_DATE
ID_DIALOG_WINDOW	ID_ZAF_DIALOG_WINDOW
ID_DIMENSION_CONSTRAINT	ID_ZAF_DIMENSION_CONSTRAINT
ID_EVENT_MANAGER	ID_ZAF_EVENT_MANAGER
ID_FORMATTED_STRING	ID_ZAF_FORMATTED_STRING
ID_GEOMETRY_MANAGER	ID_ZAF_GEOMETRY_MANAGER
ID_GROUP	ID_ZAF_GROUP
ID_HLIST	ID_ZAF_HZ_LIST
ID_HZ_LIST	ID_ZAF_HZ_LIST
ID_ICON	ID_ZAF_ICON
ID_IMAGE	ID_ZAF_IMAGE
ID_INTEGER	ID_ZAF_INTEGER
ID_LIST	ID_ZAF_LIST
ID_MAXIMIZE_BUTTON	ID_ZAF_MAXIMIZE_BUTTON
ID_MINIMIZE_BUTTON	ID_ZAF_MINIMIZE_BUTTON

ZAF 4 symbol (old)	ZAF 5 replacement
ID_NOTEBOOK	ID_ZAF_NOTEBOOK
ID_POP_UP_ITEM	ID_ZAF_POP_UP_ITEM
ID_POP_UP_MENU	ID_ZAF_POP_UP_MENU
ID_PROMPT	ID_ZAF_PROMPT
ID_PULL_DOWN_ITEM	ID_ZAF_PULL_DOWN_ITEM
ID_PULL_DOWN_MENU	ID_ZAF_PULL_DOWN_MENU
ID_RADIO_BUTTON	ID_ZAF_RADIO_BUTTON
ID_REAL	ID_ZAF_REAL
ID_RELATIVE_CONSTRAINT	ID_ZAF_RELATIVE_CONSTRAINT
ID_SCROLL_BAR	ID_ZAF_SCROLL_BAR
ID_SPIN_CONTROL	ID_ZAF_SPIN_CONTROL
ID_STATUS_BAR	ID_ZAF_STATUS_BAR
ID_STRING	ID_ZAF_STRING
ID_SYSTEM_BUTTON	ID_ZAF_SYSTEM_BUTTON
ID_TABLE	ID_ZAF_TABLE
ID_TABLE_HEADER	ID_ZAF_TABLE_HEADER
ID_TABLE_RECORD	ID_ZAF_TABLE_RECORD
ID_TEXT	ID_ZAF_TEXT
ID_TIME	ID_ZAF_TIME
ID_TITLE	ID_ZAF_TITLE
ID_TOOL_BAR	ID_ZAF_TOOL_BAR
ID_VLIST	ID_ZAF_VT_LIST
ID_VT_LIST	ID_ZAF_VT_LIST
ID_WINDOW	ID_ZAF_WINDOW
ID_WINDOW_MANAGER	ID_ZAF_WINDOW_MANAGER
ID_WINDOW_OBJECT	ID_ZAF_WINDOW_OBJECT
IMF_AUTO_SIZE	IMF_AUTO_SIZE /* AutoSize() */
IMF_BACKGROUND	IMF_BACKGROUND /* Wallpaper() */
IMF_SCALED	IMF_SCALED /* Scaled() */
IMF_TILED	IMF_TILED /* Tiled() */
Inherited	IsA
Is3D	Is3D /* ButtonType() != ZAF_FLAT_BUTTON */
IsAutoClear	AutoClear
IsBackground	Wallpaper
IsBordered	Bordered

ZAF 4 symbol (old)	ZAF 5 replacement
IsCheckBox	IsCheckBox /*ButtonType() == ZAF_CHECK_BOX */
IsCopyDraggable	CopyDraggable
IsDefaultButton	IsDefaultButton /*parent->DefaultButton() == ...*/
IsDefaultInvalid	Invalid
IsDefaultUnanswered	Unanswered
IsDestroyable	Destroyable
IsLocked	Locked
IsLowerCase	LowerCase
IsMinimizeIcon	IsMinimizeIcon /* IconType() == ZAF_MINIMIZE_ICON */
IsModal	Modal
IsMoveable	Moveable
IsMoveDraggable	MoveDraggable
IsNoncurrent	Noncurrent
IsNonselectable	Disabled
IsPasswordStyle	Password
IsRadioButton	ButtonType() == ZAF_RADIO_BUTTON
IsScaled	Scaled
IsSelectable	!Disabled()
IsSeparator	ItemType() == ZAF_SEPARATOR
IsSizeable	Sizeable
IsSupport	SupportObject
IsTemporary	Temporary
IsTiled	Tiled
IsUpperCase	UpperCase
IsViewOnly	ViewOnly
MNIF_ABOUT	MNIF_ABOUT /*ItemType() == ZAF_ABOUT_OPION */
MNIF_CHECK_MARK	MNIF_CHECK_MARK /* AllowToggling */
MNIF_CLOSE	MNIF_CLOSE /*ItemType() == ZAF_CLOSE_OPION */
MNIF_MAXIMIZE	MNIF_MAXIMIZE /* ItemType() == ZAF_MAXIMIZE_OPTION */

ZAF 4 symbol (old)	ZAF 5 replacement
MNIF_MINIMIZE	MNIF_MINIMIZE /* ItemType() == ZAF_MINIMIZE_OPION */
MNIF_MOVE	MNIF_MOVE /*ItemType() == ZAF_MOVE_OPTION */
MNIF_NON_SELECTABLE	MNIF_NON_SELECTABLE /* Disabled() */
MNIF_RESTORE	MNIF_RESTORE /* ItemType() == ZAF_RESTORE_OPION */
MNIF_SEND_MESSAGE	MNIF_SEND_MESSAGE /* SendMessageWhenSelected() */
MNIF_SEPARATOR	MNIF_SEPARATOR /* ItemType() == ZAF_SEPARATOR */
MNIF_SIZE	MNIF_SIZE /*ItemType() == ZAF_SIZE_OPTION */
MNIF_SWITCH	MNIF_SWITCH /*ItemType() == ZAF_SWITCH_OPTION */
NOTIFY_ELEMENT	ZafNotifyElement
NOTIFY_LIST	ZafNotifyList
NUMBERID	ZafNumberID
NUMID_BORDER	ZAF_NUMID_BORDER
NUMID_C_SCROLL	ZAF_NUMID_C_SCROLL
NUMID_CNR_HEADER	ZAF_NUMID_CORNER_HEADER
NUMID_COL_HEADER	ZAF_NUMID_COLUMN_HEADER
NUMID_GEOMETRY	ZAF_NUMID_GEOMETRY
NUMID_HZ_SCROLL	ZAF_NUMID_HZ_SCROLL
NUMID_MAXIMIZE	ZAF_NUMID_MAXIMIZE
NUMID_MINIMIZE	ZAF_NUMID_MINIMIZE
NUMID_OPT_CLOSE	ZAF_NUMID_OPT_CLOSE
NUMID_OPT_MAXIMIZE	ZAF_NUMID_OPT_MAXIMIZE
NUMID_OPT_MINIMIZE	ZAF_NUMID_OPT_MINIMIZE
NUMID_OPT_MOVE	ZAF_NUMID_OPT_MOVE
NUMID_OPT_RESTORE	ZAF_NUMID_OPT_RESTORE
NUMID_OPT_SIZE	ZAF_NUMID_OPT_SIZE
NUMID_OPT_SWITCH	ZAF_NUMID_OPT_SWITCH
NUMID_ROW_HEADER	ZAF_NUMID_ROW_HEADER
NUMID_SYSTEM	ZAF_NUMID_SYSTEM

ZAF 4 symbol (old)	ZAF 5 replacement
NUMID_TITLE	ZAF_NUMID_TITLE
NUMID_VT_SCROLL	ZAF_NUMID_VT_SCROLL
OBJECTID	ZafNumberID
relative	ZafRegion
S_CURRENT	N_CURRENT
S_DISPLAY_ACTIVE	S_DISPLAY_ACTIVE /*Redisplay() or S_REDISPLAY_REGION */
S_DISPLAY_INACTIVE	S_DISPLAY_INACTIVE /*Redisplay() or S_REDISPLAY_REGION */
S_NON_CURRENT	N_NON_CURRENT
SBF_CORNER	SBF_CORNER /* ScrollType() == ZAF_CORNER_SCROLL */
SBF_HORIZONTAL	SBF_HORIZONTAL /* ScrollType() == ZAF_CORNER_SCROLL */
SBF_SCALE	SBF_SCALE /* ScrollType() */
SBF_SLIDER	SBF_SLIDER /* ScrollType() */
SBF_VERTICAL	SBF_VERTICAL /* ScrollType() == ZAF_VERTICAL_SCROLL */
searchID	searchID /*Use ClassName()*/
SEEK	ZafSeek
SelectsOnDoubleClick	SelectOnDoubleClick
SelectsOnDownClick	SelectOnDownClick
SendsMessageWhenSelected	SendMessageWhenSelected
STF_LOWER_CASE	STF_LOWER_CASE /* LowerCase() */
STF_PASSWORD	STF_PASSWORD /* Password() */
STF_UPPER_CASE	STF_UPPER_CASE /* UpperCase() */
STF_VARIABLE_NAME	STF_VARIABLE_NAME /* VariableName() */
storageError	Error()
TBLF_GRID	TBLF_GRID /* Grid() */
THF_COLUMN_HEADER	THF_COLUMN_HEADER /* HeaderType() == ZAF_COLUMN_HEADER */

ZAF 4 symbol (old)	ZAF 5 replacement
THF_CORNER_HEADER	THF_CORNER_HEADER /* HeaderType() == ZAF_CORNER_HEADER */
THF_GRID	THF_GRID /* Grid() */
THF_ROW_HEADER	THF_ROW_HEADER /*HeaderType() == ZAF_ROW_HEADER*/
TRUE	true
true.	/* */
UCHAR	ZafUInt8
UI_APPLICATION	ZafApplication
UI_ATTACHMENT	ZafAttachment
UI_CONSTRAINT	ZafConstraint
UI_DEVICE	ZafDevice
UI_DEVICE	ZafDevice
UI_DIMENSION_CONSTRAINT	ZafDimensionConstraint
UI_DISPLAY	ZafDisplay
UI_DISPLAY	ZafDisplay
UI_DISPLAY_IMAGE	ZafDisplayImage
UI_ELEMENT	ZafElement
UI_ERROR_STUB	ZafErrorStub
UI_ERROR_SYSTEM	ZafErrorSystem
UI_EVENT	ZafEventStruct
UI_EVENT	ZafEventStruct
UI_EVENT_MANAGER	ZafEventManager
UI_EVENT_MANAGER	ZafEventManager
UI_EVENT_MAP	ZafEventManager
UI_GEOMETRY_MANAGER	ZafGeometryManager
UI_GRAPHICS_DISPLAY	ZafDisplay
UI_HELP_CONTEXT	UI_HELP_CONTEXT /* ZafIChar[lookupString]*/
UI_HELP_STUB	ZafHelpStub
UI_HELP_SYSTEM	ZafHelpSystem
UI_KEY	ZafKeyStruct
UI_LIST	ZafList
UI_LIST_BLOCK	ZafListBlock
UI_MACINTOSH_DISPLAY	ZafDisplay
UI_META_DISPLAY	ZafDisplay
UI_MSC_DISPLAY	ZafDisplay

ZAF 4 symbol (old)	ZAF 5 replacement
UI_MSWINDOWS_DISPLAY	ZafDisplay
UI_NEXTSTEP_DISPLAY	ZafDisplay
UI_OS2_DISPLAY	ZafDisplay
UI_PALETTE	ZafPaletteStruct
UI_PALETTE_MAP	ZafPaletteMap
UI_PATH	ZafPath
UI_PATH_ELEMENT	ZafPathElement
UI_POSITION	ZafPosition
UI_PRINTER	ZafPrinter
UI_QUEUE_BLOCK	ZafQueueBlock
UI_QUEUE_ELEMENT	ZafQueueElement
UI_REGION	ZafRegionStruct
UI_REGION_ELEMENT	ZafRegionElement
UI_REGION_LIST	ZafRegionList
UI_RELATIVE_CONSTRAINT	ZafRelativeConstraint
UI_SCROLL_INFORMATION	ZafScrollStruct
UI_TEXT_DISPLAY	ZafDisplay
UI_WCC_DISPLAY	ZafDisplay
ui_win.hpp	zaf.hpp
UI_WIN.hpp	zaf.hpp
UI_WINDOW_MANAGER	ZafWindowManager
UI_WINDOW_MANAGER	ZafWindowManager
UI_WINDOW_OBJECT	ZafWindowObject
UI_WINDOW_OBJECT	ZafWindowObject
UI_WINDOW_OBJECT::helpSystem	zafHelpSystem
UI_WINDOW_OBJECT::errorSystem	zafErrorSystem
UI_WINDOW_OBJECT::defaultStorage	UI_WINDOW_OBJECT::defaultStorage /*Use zafObjectPersistence*/
UI_XT_DISPLAY	ZafDisplay
UID_CURSOR	ZafCursor
UID_KEYBOARD	ZafKeyboard
UID_MOUSE	ZafMouse
UID_TIMER	ZafTimer
UIW_BIGNUM	ZafBignum
UIW_BORDER	ZafBorder
UIW_BUTTON	ZafButton

ZAF 4 symbol (old)	ZAF 5 replacement
UIW_COMBO_BOX	ZafComboBox
UIW_DATE	ZafDate
UIW_FORMATTED_STRING	ZafFormattedString
UIW_GROUP	ZafGroup
UIW_HZ_LIST	ZafHzList
UIW_ICON::_questionIconName	ZAF_QUESTION_ICON
UIW_ICON::_handIconName	ZAF_HAND_ICON
UIW_ICON::_exclamationIconName	ZAF_EXLAMATION_ICON
UIW_ICON::_asteriskIconName	ZAF_ASTERISK_ICON
UIW_ICON::_applicationIconName	ZAF_APPLICATION_ICON
UIW_ICON	ZafIcon
UIW_IMAGE	ZafImage
UIW_INTEGER	ZafInteger
UIW_MAXIMIZE_BUTTON	ZafMaximizeButton
UIW_MINIMIZE_BUTTON	ZafMinimizeButton
UIW_NOTEBOOK	ZafNotebook
UIW_POP_UP_ITEM	ZafPopUpItem
UIW_POP_UP_MENU	ZafPopUpMenu
UIW_PROMPT	ZafPrompt
UIW_PROMPT	ZafPrompt
UIW_PULL_DOWN_ITEM	ZafPullDownItem
UIW_PULL_DOWN_MENU	ZafPullDownMenu
UIW_REAL	ZafReal
UIW_SCROLL_BAR	ZafScrollBar
UIW_SCROLL_BUTTON	ZafScrollButton
UIW_SPIN_BUTTON	ZafSpinButton
UIW_SPIN_CONTROL	ZafSpinControl
UIW_STATUS_BAR	ZafStatusBar
UIW_STRING	ZafString
UIW_SYSTEM_BUTTON	ZafSystemButton
UIW_TABLE	ZafTable
UIW_TABLE_HEADER	ZafTableHeader
UIW_TABLE_RECORD	ZafTableRecord
UIW_TEXT	ZafText
UIW_TIME	ZafTime

ZAF 4 symbol (old)	ZAF 5 replacement
UIW_TITLE	ZafTitle
UIW_TOOL_BAR	ZafToolBar
UIW_VT_LIST	ZafVtList
UIW_WINDOW	ZafWindow
UsesAvailableRegion	UsesAvailableRegion /* RegionType() == ZAF_AVAILABLE_REGION */
UsesDefaultCellCoordinate	UsesDefaultCellCoordinate /* CoordinateType() == ZAF_CELL */
UsesDefaultMinicell- Coordinate	UsesDefaultMinicellCoordinate /* CoordinateType() == ZAF_MINICELL */
UsesDefaultPixelCoordinate	UsesDefaultPixelCoordinate /*CoordinateType() == ZAF_PIXEL */
UsesNormalHotKeys	NormalHotkeys
UsesOutsideRegion	UsesOutsideRegion /* RegionType() == ZAF_OUTSIDE_REGION */
VirtualGet	VirtualGet /*Use ZafWindowObject::BeginDraw()*/
VirtualPut	VirtualPut /*Use ZafWindowObject::EndDraw()*/
WNF_AUTO_SELECT	WNF_AUTO_SELECT /* AutoSelect() */
WNF_AUTO_SORT	WNF_AUTO_SORT /* AutoSortData() */
WNF_BITMAP_CHILDREN	WNF_BITMAP_CHILDREN /*!OSDraw() */
WNF_NO_WRAP	WNF_NO_WRAP /* !WrappedData() */
WNF_OWNERDRAW_CHILDREN	WNF_OWNERDRAW_CHILDREN /* !OSDraw() */
WNF_SELECT_MULTIPLE	WNF_SELECT_MULTIPLE /* SelectionType() == ZAF_MULTIPLE_SELECTION */
WOAF_ACCEPTS_DROP	WOAF_ACCEPTS_DROP /* AcceptDrop() */
WOAF_COPY_DRAG_OBJECT	WOAF_COPY_DRAG_OBJECT /* CopyDraggable() */
WOAF_DIALOG_OBJECT	WOAF_DIALOG_OBJECT /* ZafDialogWindow */
WOAF_LOCKED	WOAF_LOCKED /* Locked() */
WOAF_MDI_OBJECT	WOAF_MDI_OBJECT /* ZafMDIWindow */
WOAF_MODAL	WOAF_MODAL /* Modal() */

ZAF 4 symbol (old)	ZAF 5 replacement
WOAF_MOVE_DRAG_OBJECT	WOAF_MOVE_DRAG_OBJECT /* MoveDraggable() */
WOAF_NO_DESTROY	WOAF_NO_DESTROY /* !Destroyable() */
WOAF_NO_MOVE	WOAF_NO_MOVE /* !Moveable() */
WOAF_NO_SIZE	WOAF_NO_SIZE /* !Sizeable() */
WOAF_NON_CURRENT	WOAF_NON_CURRENT /* Noncurrent() */
WOAF_NORMAL_HOT_KEYS	WOAF_NORMAL_HOT_KEYS /* NormalHotKeys() */
WOAF_OUTSIDE_REGION	WOAF_OUTSIDE_REGION /* RegionType() == ZAF_OUTSIDE_REGION */
WOAF_TEMPORARY	WOAF_TEMPORARY /* Temporary() */
WOF_AUTO_CLEAR	WOF_AUTO_CLEAR /* AutoClear() */
WOF_BORDER	WOF_BORDER /* Bordered() */
WOF_INVALID	WOF_INVALID /* Invalid() */
WOF_MINICELL	WOF_MINICELL /* CoordinateType() == ZAF_MINICELL */
WOF_NON_FIELD_REGION	WOF_NON_FIELD_REGION /* RegionType() == ZAF_AVAILABLE_REGION */
WOF_NON_SELECTABLE	WOF_NON_SELECTABLE /*Disabled() */
WOF_PIXEL	WOF_PIXEL /*CoordinateType() == ZAF_PIXEL */
WOF_SUPPORT_OBJECT	WOF_SUPPORT_OBJECT /* SupportObject() */
WOF_UNANSWERED	WOF_UNANSWERED /* Unanswered() */
WOF_VIEW_ONLY	WOF_VIEW_ONLY /* ViewOnly() */
WOS_CHANGED	WOS_CHANGED /* Changed() */
WOS_EDIT_MODE	WOS_EDIT_MODE /* EditMode() */
WOS_GRAPHICS	WOS_GRAPHICS /*CoordinateType() */
WOS_INVALID	WOS_INVALID /* Invalid() */
WOS_MAXIMIZED	WOS_MAXIMIZED /* Maximized() */
WOS_MINIMIZED	WOS_MINIMIZED /* Minimized() */
WOS_OWNERDRAW	WOS_OWNERDRAW /* !OSDraw() */
WOS_READ_ERROR	WOS_READ_ERROR /* Error() == ZAF_ERROR_READ */

ZAF 4 symbol (old)	ZAF 5 replacement
WOS_REDISPLAY	WOS_REDISPLAY /* S_REDISPLAY_DATA */
WOS_SELECTED	WOS_SELECTED /* Selected() */
WOS_UNANSWERED	WOS_UNANSWERED /* Unanswered() */
Z_ERROR	ZafError
ZAF_DIALOG_WINDOW	ZafDialogWindow
ZAF_GEOMETRY_MANAGER	ZafGeometryManager
ZAF_MESSAGE_WINDOW	ZafMessageWindow
ZAF_WINDOW	ZafWindow
ZIL_BIGNUM	ZafBignumData
ZIL_CHAR_MAP	ZafCodeSetData
ZIL_COLOR	ZafLogicalColor
ZIL_COMPARE_FUNCTION	ZafCompareFunction
ZIL_DATE	ZafDateData
ZIL_DESIGNER_EVENT	ZafDesignerEvent
ZIL_DEVICE_STATE	ZafDeviceState
ZIL_DEVICE_TYPE	ZafDeviceType
ZIL_DIALOG_EVENT	ZafDialogEvent
ZIL_DLG_ABORT	S_DLG_ABORT
ZIL_DLG_CANCEL	S_DLG_CANCEL
ZIL_DLG_IGNORE	S_DLG_IGNORE
ZIL_DLG_NO	S_DLG_NO
ZIL_DLG_OK	S_DLG_OK
ZIL_DLG_RETRY	S_DLG_RETRY
ZIL_DLG_YES	S_DLG_YES
ZIL_ERA_TABLE	ZafEraStruct
ZIL_EVENT_TYPE	ZafEventType
ZIL_EXIT_FUNCTION	ZafExitFunction
ZIL_EXPORT_CLASS	ZAF_EXPORT
ZIL_EXPORT_CLASS	ZAF_EXPORT
ZIL_FILE	ZafFile
ZIL_FILE_CHAR	char
ZIL_I18N	ZafI18nData
ZIL_IBIGNUM	ZafIBignum
ZIL_ICHAR	ZafIChar
ZIL_ICHAR	ZafIChar
ZIL_IMAGE_HANDLE	OSImageID

ZAF 4 symbol (old)	ZAF 5 replacement
ZIL_IMAGE_TYPE	ZafDeviceType
ZIL_INT16	ZafInt16
ZIL_INT32	ZafInt32
ZIL_INT8	ZafInt8
ZIL_INTERNATIONAL::	<no replacement, not needed>
ZIL_LANGUAGE	ZafLanguageData
ZIL_LANGUAGE_ELEMENT	ZafLanguageData
ZIL_LOCALE	ZafLocaleData
ZIL_LOCALE_ELEMENT	ZafLocaleData
ZIL_LOGICAL_EVENT	ZafLogicalEvent
ZIL_LOGICAL_FONT	ZafLogicalFont
ZIL_LOGICAL_PALETTE	ZafPaletteState
ZIL_LOGICAL_PATTERN	ZafLogicalFillPattern
ZIL_MAXPATHLEN	ZAF_MAXPATHLEN
ZIL_MSG_ABORT	ZAF_DIALOG_ABORT
ZIL_MSG_CANCEL	ZAF_DIALOG_CANCEL
ZIL_MSG_HELP	ZAF_DIALOG_HELP
ZIL_MSG_IGNORE	ZAF_DIALOG_IGNORE
ZIL_MSG_NO	ZAF_DIALOG_NO
ZIL_MSG_OK	ZAF_DIALOG_OK
ZIL_MSG_RETRY	ZAF_DIALOG_RETRY
ZIL_MSG_YES	ZAF_DIALOG_YES
ZIL_NEW_FUNCTION	ZafObjectConstructor
ZIL_NULLP	ZAF_NULLP
ZIL_NUMBER	ZafNumber
ZIL_NUMBERID	ZafNumberID
ZIL_OBJECTID	ZafNumberID
ZIL_PRINTER_MODE	ZafPrinterMode
ZIL_RAW_CODE	ZafRawCode
ZIL_RBIGNUM	ZafRBignum
ZIL_REVISION	ZAF_REVISION
ZIL_SCREENID	ZafScreenIDType
ZIL_STDARG	ZafStandardArg
ZIL_STORAGE	ZafStorage
ZIL_STORAGE_DIRECTORY	ZafStorageDirectory
ZIL_STORAGE_OBJECT	ZafStorageFile
ZIL_STORAGE_OBJECT_READ_ONLY	ZafStorageFile

ZAF 4 symbol (old)	ZAF 5 replacement
ZIL_STORAGE_OBJECT_READ_ONLY	ZafStorageFile
ZIL_STORAGE_READ_ONLY	ZafStorage
ZIL_STORAGE_READ_ONLY	ZafStorage
ZIL_SYSTEM_EVENT	ZafSystemEvent
ZIL_SYSTEM_EVENT	ZafSystemEvent
ZIL_TIME	ZafTimeData
ZIL_UINT16	ZafUInt16
ZIL_UINT32	ZafUInt32
ZIL_UINT8	ZafUInt8
ZIL_USER_EVENT	ZafUserEvent
ZIL_USER_FUNCTION	ZafUserFunction
ZIL_UTIME	ZafUTimeData
ZIL_VERSION	ZAF_VERSION
ZINC_SIGNATURE	ZafSignature

ZMake

“ZMake” is a make utility providing support for portable, compiler-independent make files. Although make utilities provided with various compilers may be used to build ZAF projects, Zinc provides ZMake as an alternate, preferred method. ZMake supports the same concepts and most of the same syntax as compiler-supplied make utilities.

ZMake make files are easier to create and use than traditional make files since much of their internal complexity is off loaded to sub-make files supplied by Zinc. (This is largely a product of make file organization, however, and is not specifically related to ZMake.) By abstracting make syntax slightly, ZMake allows a single make file to support many different compilers. This greatly reduces make file maintenance issues for both Zinc and its customers.

Make files shipped with PC versions of ZAF are named ZMAKE.MAK, the default file name of a ZMake make file. Currently, ZMake is implemented for all compilers supported by ZAF for Microsoft Windows and MS DOS. ZMake32 should be used under 32-bit Microsoft Windows platforms. Support for other environments is planned. Currently, make files for Motif are generated from POSIX.MAK and do not use the utility or concepts discussed in this reference.

Usage

zmake [options] <target>

The command line options for the ZMake utility are as follows:

Option	Description
-f [makefile]	Uses [makefile] instead of the default ZMAKE.MAK
-k	Keeps temporary response files
-p	Shows pre-parser output (for make file debugging)
[macro=myString]	Defines a macro that may be used in the make file

Some examples of using the ZMake utility follow:

Command Line	Description
zmake win16	Builds a ZAF program for 16-bit Microsoft Windows using the make file ZMAKE.MAK.

Command Line	Description
<code>zmake -f makefile.mak win32</code>	Builds a ZAF program for 32-bit Microsoft Windows using the make file MAKEFILE.MAK instead of the default ZMAKE.MAK file.
<code>zmake myapp.exe</code>	Builds the program myapp.exe using the make file ZMAKE.MAK.
<code>zmake myfile.obj</code>	Builds only the object module myfile.obj using the make file ZMAKE.MAK.

Environment Variables

Before ZMake can be executed successfully, environment variables must be defined. These environment variables may be used to define the compiler-specific aspects of the make, thereby allowing the make files themselves to remain compiler-independent.

Note that, like other make utilities, ZMake treats make file macros and environment variables identically. Therefore it is possible to define environment variables inside the make file itself, or to define the make file macros (discussed later) in the OS environment. Zinc has chosen to define the following variables in the environment to maximize compiler independence and ease of use. See [Macros](#) for more information.

The ZMake-compatible make files provided with ZAF utilize the following environment variables:

Environment Variable	Description
ZAF_ROOT	<i>Required.</i> Specifies the top-level directory containing the ZAF 5 installation. This directory is referenced to find source files, include files, libraries, etc.
ZINC_COMPILER	<i>Required.</i> Specifies which compiler and linker ZMake should invoke during the build process. ZINC_COMPILER may be any of the following: BORLAND, MICROSOFT, WATCOM.

Environment Variable	Description
INCLUDE	<i>Required for all compilers except Borland.</i> Specifies a complete include path. When specified, INCLUDE should list the paths to compiler and ZAF headers, plus any other headers required by the project. If not specified (Borland only), the include path specified in the compiler configuration file (turbo.ccfg or bcc32.ccfg) will be used. Note that the INCLUDE environment variable is not specifically used by ZMake, but is required by some compilers.
LIB	<i>Required for all compilers except Borland.</i> Specifies a complete library path. When specified, LIB should list the paths to compiler and ZAF libraries, plus any other libraries required by the project (ChartFolio, for example). If not specified (Borland only), the lib path specified in the linker configuration file (tlink.ccfg or tlink32.ccfg) will be used. Note that the INCLUDE environment variable is not specifically used by ZMake, but is required by some linkers.

Macros

During the build process, ZMake make files expect several symbols to be defined. Some of these symbols may be defined as environment variables (see the preceding section), and others may be defined as make file macros. If a symbol is defined *both* as an environment variable and as a macro, the later macro definition is used.

The following macros are specifically supported:

Macro	Definition
ZAF_ROOT	<i>Required.</i> Normally specified as an environment variable. See Environment Variables for more information.
ZINC_COMPILER	<i>Required.</i> Normally specified as an environment variable. See Environment Variables for more information.

Macro	Definition
PLATFORM	<i>Required.</i> Specifies the target platform supported by the current build option. Multiple build options (and therefore multiple platforms) may be supported by a single make file. Refer to make file examples (below) for sample usage. Valid options: WIN16 (16 bit Microsoft Windows platforms) WIN32 (32 bit Microsoft Windows platforms)
TARGET_TYPE	<i>Required.</i> Specifies the target type supported by the current build option. Valid options: EXE (Executable. ZAF statically linked) EXEDLL (Executable. ZAF dynamically linked) LIB (Library) LIBDLL (Library. Calls functions in ZAF DLL) DLL (Library. Dynamically linked)
TARGET	<i>Required.</i> Specifies the name of the target file.
SOURCE_FILES	<i>Required.</i> Specifies the source code files needed to build the target.
IMPORT_LIBRARY	<i>Required for DLLs only.</i> Specifies the name of the Windows import library that will be generated when the TARGET_TYPE is "DLL".
ADD_OBJECT_FILES	<i>Optional.</i> Specifies the names of additional object files to be used in this build, if any.
ADD_LIBRARIES	<i>Optional.</i> Specifies the names of additional library files to be used in this build, if any.
DEFINITION_FILE	<i>Optional.</i> Specifies the name of the Windows ".def" file to be used in this build, if any.
RESOURCE_FILE	<i>Optional.</i> Specifies the name of the Windows ".res" file to be used in this build, if any.
DEBUG	<i>Optional.</i> Specifies whether the code is compiled with debug information or not. Valid options: off (No debug information - default) on (Build with debug information)
MAKE	<i>Internal.</i> Returns the name of the make utility currently running. This will normally be "ZMAKE".

Example

```
# --- zake.mak -----
# Define macros used by main make file
```

```
# These macros are usually defined by environment variables.
ZAF_ROOT = c:\zaf500
ZINC_COMPILER = borland    (options: borland, microsoft, watcom)

# Options common to all targets
DEBUG = off
SOURCE_FILES = myappl.cpp myapp2.cpp

# 16 bit windows target.
TARGET = myapp.exe
TARGET_TYPE = exe          (options: exe, exedll, lib, libdll, dll)
PLATFORM = win16           (options: win32, win16)
DEFINITION_FILE = myapp.def
RESOURCE_FILE = bitmaps.rc
!include <zaf.mak>          (required. generate rules for this target)

# 32 bit windows target.
TARGET = myapp32.exe
TARGET_TYPE = exe
PLATFORM = win32
DEFINITION_FILE =
RESOURCE_FILE = bitmaps.rc
!include <zaf.mak>

# 32 bit windows target linked to dll.
TARGET = myapp32d.exe
TARGET_TYPE = exedll
PLATFORM = win32
DEFINITION_FILE =
RESOURCE_FILE = bitmaps.rc
!include <zaf.mak>
```

Internal Make Files

Most ZMake users will not need to understand the internal workings of traditional make files and their different syntaxes. Those who need such information should consult the documentation for their specific make utility and study the internal ZAF make files introduced in this section.

ZAF make files for PC platforms (Motif currently uses a different system) are easy to understand and use, in part because Zinc has hidden much of the normal make file complexity in other make files typically unseen by the programmer. In the above example, “!include <zaf.mak>” triggers these other make files to complete the build process.

The following information provides insight into the internal organization of these “internal” make files. This information is considered “advanced” and is not necessary for the full utilization of Zinc Application Framework.

As shipped, the ZMake build process actually interprets and executes four different make files. All internal make files are found in the “\include” directory and should be individually studied by programmers interested in full control over the build process.

The four make files, and the order in which they are referenced, follow:

Make file	Purpose
zmake.mak	The user-supplied make file. (See the example above.) This make file is the one initially invoked by ZMake. This file must include the command <code>"!include <zaf.mak>"</code> to trigger the remainder of the build process.
zaf.mak	The main ZAF make file. The primary purpose of this file is to ensure that all necessary macros are defined and available to subsequent steps in the build process.
opts???.mak	These files specify compiler options unique to each compiler. By placing the compiler options in this file, Zinc ensures that the options used to build the library match the options used to build applications. This is where additional compiler options may be added, such as pre-compiled headers. (Non-matching compile options can cause difficult bugs.) Note: These files have been created to allow non-ZMake users to access them in their own compiler-specific make files.
zaf???.mak	These files specify build rules unique to each linker.

Note: The “???” in the above filenames refers to a three letter abbreviation for each compiler. For example, actual names include: `optsbor.mak` (for Borland), `optsmc.mak` (for Microsoft) and `optswat.mak` (for Watcom).

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.) For comparison purposes, the first notification event is equal to `N_NOTIFY_FIRST` and the last is equal to `N_NOTIFY_LAST`.

- 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.*

<code>N_CHANGE_PAGE</code>	Notifies <code>ZafNotebook</code> that the current notebook page has changed.
<code>N_CLOSE</code>	Notifies <code>ZafWindow</code> that it is about to be closed. If <code>ZafWindow</code> returns 0, it will be closed; otherwise it will not be closed. (Operation may not be cancellable on Motif.)
<code>N_CURRENT</code>	Notifies <code>ZafWindowObject</code> that it has received focus. <code>event.windowObject</code> points to the object losing focus. Calls member <code>UserFunction()</code> , if any.
<code>N_EXIT</code>	Notifies <code>ZafWindowManager</code> that the application is about to exit. <code>ZafWindowManager</code> responds by returning the result of <code>exitFunction()</code> , if any. If <code>exitFunction()</code> returns 0, it will exit; otherwise it will not exit. (Operation may not be cancellable on Motif.)
<code>N_HSCROLL</code>	Notifies <code>ZafWindowObject</code> that its corresponding horizontal <code>ZafScrollBar</code> has scrolled, if the scroll bar is <code>SupportObject()</code> . <code>ZafWindowObject</code> responds by scrolling its data. If the scroll bar is not <code>SupportObject()</code> , then member <code>UserFunction()</code> is called.
<code>N_MAXIMIZE</code>	Notifies <code>ZafWindow</code> that it has been maximized, and that its <code>zafRegion</code> reflects the new size. <code>ZafWindow</code> responds by updating its internal information to correspond with the operating system object.
<code>N_MINIMIZE</code>	Notifies <code>ZafWindow</code> that it has been minimized. <code>ZafWindow</code> responds by updating its internal information to correspond with the operating system object.
<code>N_MOUSE_ENTER</code>	Notifies <code>ZafWindowObject</code> that the mouse has entered its region. <code>ZafWindowManager::mouseObject</code> already points to the object and <code>event.position</code> reflects

	the mouse position. 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. event.windowObject points to the object gaining focus. Calls memberUserFunction(), if any. (Operation may not be cancellable on Motif.)
N_RESET_I18N	Notifies ZafWindowObjects that depend on internationalization (such as ZafBignum, ZafDate, and ZafTime) that the internationalization values have been reset, and such objects should redisplay their data according to the new values.
N_RESTORE	Notifies ZafWindow that it has been restored from a maximized or minimized state, and that its zafRegion reflects the new size. ZafWindow responds by updating its internal information to correspond with the operating system object.
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. For comparison purposes, the first system event is equal to S_SYSTEM_FIRST and the last is equal to S_SYSTEM_LAST.

- 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 subclasses.*

S_ADD_OBJECT	Causes ZafWindow to add event.windowObject as a child.
S_CLOSE	Causes ZafWindowManager to close a window, and any temporary windows above it. If event.windowObject is null, ZafWindowManager closes its top-most non-temporary window; otherwise, it closes the window specified by event.windowObject.
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 ZafWindowManager to maximize a window on the display. If event.windowObject is non-null, ZafWindowManager passes the event to event.windowObject; otherwise, it passes the event to its top-most window.
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 ZafWindowManager to minimize a window on the display. If event.windowObject is non-null, ZafWindowManager passes the event to event.windowObject; otherwise, it passes the event to its top-most window.
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 ZafWindowManager to restore a maximized or minimized window on the display. If event.windowObject is non-null, ZafWindowManager passes the event to event.windowObject; otherwise, it passes the event to its top-most window.
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. The S_CREATE message itself is not propagated to the object's children, but the appropriate messages are sent to the children for creation.
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.
S_SET_FOCUS	Causes ZafWindowObject to gain focus. Normally put on the event manager's queue in NotifyFocus() if an object with invalid data should not lose focus.
S_VALID_CHECK	Causes ZafWindowObject to check the validity of its data. The object returns S_VALID, S_INVALID, or S_ERROR according to the validity of its data. Normally sent to an object from NotifyFocus().

**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_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 ZafWindowObject and some functions to indicate error status.
S_INVALID	Returned by ZafWindowObject in response to S_VALID_CHECK if its data is invalid.
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.
S_VALID	Returned by ZafWindowObject in response to S_VALID_CHECK if its data is valid.

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.) For comparison purposes, the first logical event is equal to L_LOGICAL_FIRST and the last is equal to L_LOGICAL_LAST.

- 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. In Motif, only mapped if ZafRegister-Mouse() was called for the object with true in the <code>rightMouse</code> parameter.
L_CONTINUE_- SELECT	Mapped for the ZafWindowObject under the mouse when the mouse moves while the left button is still depressed. In Motif, only mapped if ZafRegister-Mouse() was called for the window with true in the <code>leftMouse</code> parameter.
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 <code>ZafWindowObject::doubleClickRate</code> .
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_MOUSE1_DOWN	This is the same as L_BEGIN_SELECT.
L_MOUSE1_MOVE	This is the same as L_CONTINUE_SELECT.
L_MOUSE1_UP	This is the same as L_END_SELECT.
L_MOUSE2_DOWN	This is the same as L_BEGIN_ESCAPE.
L_MOUSE2_MOVE	This is the same as L_CONTINUE_ESCAPE.
L_MOUSE2_UP	This is the same as L_END_ESCAPE.
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. In Motif, only mapped if [ZafRegisterMouse\(\)](#) was called for the object with true in the view parameter.

**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, ZafViList, 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

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().

E_KEY	Indicates the ZafKeyboard device.
E_MOUSE	Indicates the ZafMouse device.
E_OSEVENT	Indicates a system event on any native environment, and is assigned to event.type. event.InputType() may return E_KEY or E_MOUSE, as appropriate. The value of this constant is also used for E_MACINTOSH, E_MOTIF, E_MSWINDOWS, E_OS2, and E_XT.

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. For comparison purposes, the first raw device event is equal to E_DEVICE_FIRST and the last is equal to E_DEVICE_LAST.

E_CURSOR	Indicates the ZafCursor device.
E_DEVICE	Indicates all devices.
E_HELP TIPS	Indicates the ZafHelpTips device.

E_TIMER Indicates the ZafTimer 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 ?. For comparison purposes, the first device event is equal to D_DEVICE_FIRST and the last is equal to D_DEVICE_LAST.

- 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 Device Requests: 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).
DM_HORIZONTAL_- SPLIT	Causes ZafMouse to show the horizontal split image (MSWindows, Motif, DOS, OS/2, Macintosh).
DM_VERTICAL_- SPLIT	Causes ZafMouse to show the vertical split 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. For comparison purposes, the first application event is equal to A_APPLICATION_FIRST and the last is equal to A_APPLICATION_LAST.

- 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_CLOSE_WINDOW	Causes the application to close any 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_HELP_CONTEXT	Causes ZafHelpManager to display the help context specified by event.text. (The object handling the event deletes event.text.)

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, or if it’s already on the screen, the window receives focus. (The object handling the event deletes event.text.)
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_RESET_I18N	Causes the application to change its language and locale bases specified in event.text. (The object handling the event deletes event.text.)
A_RESTORE_- WINDOWS	Causes ZafWindowManager to restore all the minimized windows on the display.

Property Matrices

Zinc Application Framework 5 supports a very large number of individual “properties” or attributes that may be manipulated to change object appearance and behavior. These properties are actually data members of each class that are manipulated using accessor methods. Knowledge of these properties, their definitions and interrelationships is vital to full usage of ZAF. This Appendix provides a set of tables that may be used as a convenient reference to “object properties.” In each table a matrix is presented. Each “cell” in the matrix indicates how a specific derived object (listed in the top row of the matrix) supports a single property inherited from its base class (listed in the left column of the matrix). Many possible types and degrees of support are possible.

Most properties are supported in one of several predictable ways. The following chart legend shows the symbols found on the property matrices and their corresponding meanings.

Symbol	For this derived class, the property is ...
“ ”	(blank) undefined
•	supported normally (may be inherited functionality)
Ø	initialized to zero
A	forced to “ZAF_AVAILABLE_REGION”
B	see base class for definition
C	initialized to “ZAF_CELL”
D	supported by DOS only (normally under end user control in other operating environments)
F	forced to “false”
I	forced to “ZAF_INSIDE_REGION”
M	forced to “ZAF_MULTIPLE_SELECTION”
N	forced to “NULL”
O	forced to “ZAF_OUTSIDE_REGION”
S	forced to “ZAF_SINGLE_SELECTION”
T	forced to “true”

In addition to properties and objects, the property matrices show two other pieces of information: support for “Dynamic” property manipulation, and constructor property initialization.

“Dynamic” property manipulation refers to a programmer’s ability to change the state of a property at any time. Some properties may be changed without regard for an object’s current state while others may only be changed when not

being “managed” or displayed. Dynamic properties may be changed at any time and are indicated by the supported symbol (“•”) in the “Dynamic” column.

All ZAF user interface objects have at least three very different constructors. For ease of discussion, these are known as the “Normal” constructor, the “Copy” constructor, and the “Persistent” constructor. The Normal constructor is the one used in straight code—screen coordinates are specified and some properties while the remaining properties are defaulted. The Copy constructor duplicates an existing object—including all of its property states—and returns a pointer to the copy. The Persistent constructor loads an object from a ZAF binary data file and initializes many of its properties based on the contents of the file.

Each of these constructors is shown along with the value of each property as it is initialized by the constructor. “•” in these columns indicates that the property is being copied or read from elsewhere rather than being set to a known default value.

Base Classes

All ZAF user interface objects are ultimately derived from ZafWindow and/or ZafWindowObject. These two base classes therefore encapsulate the majority of the properties that may be manipulated using ZAF. The first two tables presented in this appendix detail these most important classes and will be the ones most commonly consulted.

The first table lists ZafWindowObject properties and indicates how each of its derived classes supports its many properties. The second table lists ZafWindow properties (including those inherited from ZafWindowObject) and indicates how classes derived from ZafWindow support these properties.

Many exceptional cases exist. These are detailed in the footnotes following the ZafWindow matrix.

**Derived
Classes**

ZAF user interface subclasses contain far fewer properties since they inherit most of the functionality they need from their base classes. Unique properties are listed in separate “mini-matrices” for each object along with properties inherited from base classes other than ZafWindow and ZafWindowObject (since these properties are already covered in the base class matrices). Property inheritance, supported value, “Dynamic” property information, and default constructor initializations are shown.

A “B” in a matrix indicates that the base class matrix should be consulted. A “•” indicates that the class supports the property, although the property may be restricted by base class definition.

Derived class or “subclass” matrices follow the base class matrices. No footnotes are supplied or necessary for the derived classes.

Base Property Matrix—ZafWindowObject Classes

ZafWindowObject Property ¹	Dynamic ²	Con- structors			Derived ZafWindowObject																					
		Normal	Copy	Persistent	Prompt	String	FmtString	Bignum	Date	Integer	Real	Time	UTime	Button	Icon	Bitmap	Image	ProgressBar	Splitter	PullDnItem	PopUpItem	Border	TitleBar	SysButton	MaxButton	MinButton
AcceptDrop	•	F	•	•	F	•	•	•	•	•	•	•	•	•	•	F	F	F	F	F	F	F	F	F	F	F
AutomaticUpdate ⁴	•	T	T	T	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
BackgroundColor	•		• ⁵	• ⁵	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	D	D	D	D	D	D	D
Bordered		F	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	F	F	F	F	F	F	F
Changed	•	F	•	F	F	•	•	•	•	•	•	•	•	•		F				•						
CoordinateType		C	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CopyDraggable	•	F	•	•	F	•	•	•	•	•	•	•	•	•	•	F	F	F	F	F	F	F	F	F	F	F
Disabled	•	F	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	F	F	F	F	F	F
EditMode		F	•	F	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•
Error	•	N	•	N	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Focus	•	F	F	F	F	•	•	•	•	•	•	•	•	•	•	F	F	F	F	•	•	F	F	•	F	F
Font	•		• ⁵	• ⁵	•	•	•	•	•	•	•	•	•	•	•	N	N	•	N	D	D	N	D			
HelpContext	•	N	•	•	N	•	•	•	•	•	•	•	•	•	•	N	N	N	N	•	•	N	N	N	N	N
HelpObjectTip	•	N	•	•	•	•	•	•	•	•	•	•	•	•	•	•	N	•	•	•	•	N	N	N	N	N
LinkDraggable	•	F	•	•	F	•	•	•	•	•	•	•	•	•	•	F	F	F	F	F	F	F	F	F	F	F
MoveDraggable	•	F	•	•	F	•	•	•	•	•	•	•	•	•	•	F	F	F	F	F	F	F	F	F	F	F
Noncurrent	•	F	•	•	T	•	•	•	•	•	•	•	•	•	•	T	T	T	T	•	•	T	T	F	T	T
OSDraw		T	•	•	•	•	•	•	•	•	•	•	•	•	•	F	•	•	F	•	•	•	•	•	•	•
Parent		N	N	N	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
ParentDrawBorder ⁷		F	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	F	F	F	F	F	F	F	F	F
ParentDrawFocus ⁸		F	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	F	F	F	F	F	F	F	F	F
ParentPalette		F	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	F	•	F	F	F	F	F	F	F
QuickTip	•	N	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	N	N	N	N	N
Region	•	∅	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
RegionType		I	•	•	I	•	•	•	•	•	•	•	•	•	I	•	•	•	•	I	I	O	A	A	A	A
Selected	•	F	•	•	F	•	•	•	•	•	•	•	•	•	•	•	•	F	F	F	•	F	F	F	F	F
SupportObject		F	•	•											•				F	F	F	T	T	T	T	T
SystemObject		T	•	T																						
Text ⁹	•				•	•	•	•	•	•	•	•	•	•	•	•		•		•	•		•			
TextColor	•		• ⁵	• ⁵	•	•	•	•	•	•	•	•	•	•	•	N	N	•	N	D	D	N	D			
UserFunction	•	N	•	•	N	•	•	•	•	•	•	•	•	•	•	N	N	N	•		•	N	N	N	N	N
UserPaletteData	•	N	• ⁵	• ⁵	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	D	D	D	D	D	D	D
Visible	•	T	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•

Legend to footnotes follows the ZafWindow matrix.

1. “Property” refers to accessor functions inside `ZafWindowObject` and `ZafWindow` that manipulate private data members. Each accessor has both a “Get” and a “Set” variant. For example, “`SetAcceptDrop(true)`” sets a property, while “`if (AcceptDrop())`” checks the status of the property.
2. “Dynamic” indicates the property may be changed dynamically, i.e. after the object has been initialized and has a `screenID`. If the property cannot be changed dynamically then “`SetProperty`” functions must be called prior to adding the object to a managed parent or to the `ZafWindowManager`. Alternatively, the object may be subtracted, properties changed, and then re-added to its parent.
3. Property is set on `ZafTreeList` only, and affects the entire tree including all `ZafTreeItems`.
4. “`AutomaticUpdate`” is defined only for palette manipulation functions (`BackgroundColor`, `TextColor`, `Font`, `UserPaletteData`) plus `Add` and `Subtract`. When “`AutomaticUpdate()==false`” the display is not refreshed when these functions are called. This allows the programmer to fully manipulate the palette, or add or subtract many items on a parent without slow and distracting screen updates. “`SetAutomaticUpdate(true)`” restores the normal display behavior and immediately redisplay the changes made while `AutomaticUpdate` was false.

Note: `AutomaticUpdate` is not defined for other contexts and is therefore not portable outside of palette functions, `Add` and `Subtract`. It may result in unpredictable and/or non-portable behavior under other conditions and should therefore never be left in the “false” state. In general, “`SetAutomaticUpdate(false)`” and “`SetAutomaticUpdate(true)`” should immediately precede and follow the functions it is intended to affect.

5. This accessor function manipulates the `UserPaletteData` member which is an instance of `ZafPaletteData`—a separate class. Initialization, duplication and persistence are handled by `ZafPaletteData` rather than the user interface class.
6. Property is valid in some contexts for this class, but not others. For example, “`SetDisabled(true)`” is valid for proper child window only (i.e. windows without decorations).
7. By default, this property applies only to children of a `ZafStatusBar`. However, it may be exploited for other purposes.
8. By default, this property applies only to children of `ZafHzList`, `ZafVtList`, `ZafTreeList`, and `ZafTreeItem`. However, it may be exploited for other purposes.
9. Proper child windows cannot have decorations (e.g. `TitleBar`, `Border`, etc.) therefore, `Text` is undefined in this case. MDI child windows do have decorations.
10. This property applies only to top-level windows, meaning that the window may not have a parent. This property may not be supported in some environments.

11. Instances of this class are forced to “SetTemporary(true)” by ZafComboBox when they are the ZafComboBox::list member.

Property Matrices—Derived Classes

Bignum

ZafBignum Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowInvalid	String	•	B	B	B	B
AutoClear	String	•	B	B	B	B
BignumData	•	•	•	N	•	•
HzJustify	String	•	B	B	B	B
InputFormatData	String	•	B	B	B	B
Invalid	String	•	B	B	B	B
LowerCase	String	F		B	B	B
OutputFormatData	String	•	B	B	B	B
Password	String	F		B	B	B
RangeData	String	•	B	B	B	B
ReportInvalid	String	•	B	B	B	B
StringData	String	INVALID		B	B	B
Unanswered	String	•	B	B	B	B
UpperCase	String	F		B	B	B
VariableName	String	F		B	B	B
ViewOnly	String	•	B	B	B	B

Bitmap

ZafBitmap Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoSize	•	•		T	•	•
BitmapData	•	•	•	N	•	•

Border

ZafBorder Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
Width (static member)	•	varies				

Button

ZafButton Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowDefault	•	•		F	•	•
AllowToggling	•	•		F	•	•
AutoRepeatSelection	•	•		F	•	•
AutoSize	•	•		T	•	•
BitmapData	•	•	•	N	•	•
ButtonType	•	•		NATIVE	•	•
Depressed	•	internal		F	F	F
Depth	•	•		2	•	•
HotKeyChar	•	•		Ø	•	•
HotKeyIndex	•	•		-1	•	•

ZafButton Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
HzJustify	•	•		CENTER	•	•
SelectOnDoubleClick	•	•		F	•	•
SelectOnDownClick	•	•		F	•	•
SendMessageText	•	•	•	N	•	•
SendMessageWhenSelected	•	•		F	•	•
StringData	•	•	•	N	•	•
Value	•	•	•	Ø	•	•
VtJustify	•	•		CENTER	•	•

ComboBox

ZafComboBox Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoSortData	•	•	•	list	list	list
ViewOnly	•	•		F	•	•

Date

ZafDate Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoClear	String	•	B	B	B	B
DateData	•	•	•	N	•	•
HzJustify	String	•	B	B	B	B
InputFormatData	String	•	B	B	B	B
Invalid	String	•	B	B	B	B
LowerCase	String	F		B	B	B
OutputFormatData	String	•	B	B	B	B
Password	String	F		B	B	B
RangeData	String	•	B	B	B	B
ReportInvalid	String	•	B	B	B	B
StringData	String	INVALID		B	B	B
Unanswered	String	•	B	B	B	B
UpperCase	String	F		B	B	B
VariableName	String	F		B	B	B
ViewOnly	String	•	B	B	B	B

Formatted String

ZafFormattedString Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowInvalid	String	•	B	B	B	B
AutoClear	String	•	B	B	B	B
CompressedData	•	•		N	•	•
DeleteData	•	•		N	•	•
ExpandedData	•	•		StringData	StringData	StringData
FormatData	•	•		InputForm- formatData	InputForm- formatData	InputForm- formatData

ZafFormattedString Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
HzJustify	String	•	B	B	B	B
InputFormatData	String	INVALID		B	B	B
Invalid	String	•	B	B	B	B
LowerCase	String	F		B	B	B
OutputFormatData	String	INVALID		B	B	B
Password	String	F		B	B	B
RangeData	String	INVALID		B	B	B
ReportInvalid	String	•	B	B	B	B
StringData	String	INVALID		B	B	B
Unanswered	String	•	•	B	B	B
UpperCase	String	F		B	B	B
VariableName	String	F		B	B	B
ViewOnly	String	•	B	B	B	B

Group

ZafGroup Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoSelect	•	•		F	•	•
HotKeyChar	•	•		Ø	•	•
HotKeyIndex	•	•		-1	•	•
StringData	•	•	•	N	•	•

HzList

ZafHzList Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoSortData	•	•	•	F	•	•
CellHeight	•	•		parameter	•	•
CellWidth	•	•		parameter	•	•

Icon

ZafIcon Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowDefault	Button	F		B	B	B
AllowToggling	Button	F		B	B	B
AutoRepeatSelection	Button	F		F	F	F
AutoSize	Button	•	B	B	B	B
BitmapData	Button	N		B	B	B
ButtonType	Button	FLAT		FLAT	B	B
Depressed	Button	•	B	B	B	B
Depth	Button	varies		B	B	B
HotKeyChar	Button	Ø		B	B	B
HotKeyIndex	Button	-1		B	B	B
HzJustify	Button	•	B	B	B	B
IconData	•	•	•	N	•	•
IconType	•	•		NATIVE	•	•

ZafIcon Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
SelectOnDoubleClick	Button	•	B	T	B	B
SelectOnDownClick	Button	•	B	B	B	B
SendMessageText	Button	•	B	B	B	B
SendMessageWhenSelected	Button	•	B	B	B	B
StringData	Button	•	B	B	B	B
Value	Button	•	B	B	B	B
VtJustify	Button	•	B	B	B	B

Image

ZafImage Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoSize	•	•		F	•	•
PathID	•	•	•	-1	•	•
PathName	•	•	•	N	•	•
Scaled	•	•		F	•	•
Tiled	•	•		F	•	•
Wallpaper	•	•		F	•	•

Integer

ZafInteger Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowInvalid	String	•	B	B	B	B
AutoClear	String	•	B	B	B	B
HzJustify	String	•	B	B	B	B
InputFormatData	String	•	B	B	B	B
IntegerData	•	•	•	N	•	•
Invalid	String	•	B	B	B	B
LowerCase	String	F		B	B	B
OutputFormatData	String	•	B	B	B	B
Password	String	F		B	B	B
RangeData	String	•	B	B	B	B
ReportInvalid	String	•	B	B	B	B
StringData	String	INVALID		B	B	B
Unanswered	String	•	B	B	B	B
UpperCase	String	F		B	B	B
VariableName	String	F		B	B	B
ViewOnly	String	•	B	B	B	B

Maximize Button

ZafMaximizeButton Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowDefault	Button	F		B	B	B
AllowToggling	Button	F		B	B	B
AutoRepeatSelection	Button	F		B	B	B

ZafMaximizeButton Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoSize	Button	T		B	B	B
BitmapData	Button	internal		B	B	B
ButtonType	Button	3D		3D	B	B
Depressed	Button	internal		B	B	B
Depth	Button	1		1	B	B
HotKeyChar	Button	Ø		B	B	B
HotKeyIndex	Button	-1		B	B	B
HzJustify	Button	CENTER		B	B	B
SelectOnDoubleClick	Button	F		B	B	B
SelectOnDownClick	Button	F		B	B	B
SendMessageText	Button	N		B	B	B
SendMessageWhenSelected	Button	T		T	B	B
StringData	Button	internal		B	B	B
Value	Button	internal		S_MAX- IMIZE	B	B
VtJustify	Button	CENTER		B	B	B

MDIWindow

ZafMDIWindow Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
MDIType	•	•		parameter	•	•

Minimize Button

ZafMinimizeButton Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowDefault	Button	F		B	B	B
AllowToggling	Button	F		B	B	B
AutoRepeatSelection	Button	F		B	B	B
AutoSize	Button	T		B	B	B
BitmapData	Button	internal		B	B	B
ButtonType	Button	3D		3D	B	B
Depressed	Button	internal		B	B	B
Depth	Button	1		1	B	B
HotKeyChar	Button	Ø		B	B	B
HotKeyIndex	Button	-1		B	B	B
HzJustify	Button	CENTER		B	B	B
SelectOnDoubleClick	Button	F		B	B	B
SelectOnDownClick	Button	F		B	B	B
SendMessageText	Button	N		B	B	B
SendMessageWhenSelected	Button	T		T	B	B
StringData	Button	internal		B	B	B
Value	Button	internal		S_MIN- IMIZE	B	B
VtJustify	Button	CENTER		B	B	B

Notebook

ZafNotebook Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
CurrentPage	•	•	•	-1	•	-1
TabHeight	•	•		Ø	•	•
TabText	•	•	•			
TabWidth	•	•		Ø	•	•

PopUpItem

ZafPopUpItem Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowDefault	Button	F		B	B	B
AllowToggling	Button	•		B	B	B
AutoRepeatSelection	Button	F		F	F	F
AutoSize	Button	F		B	B	B
BitmapData	Button	N		B	B	B
ButtonType	Button	FLAT		FLAT	B	B
Depressed	Button	internal		B	B	B
Depth	Button	Ø		Ø	B	B
HotKeyChar	Button	•	B	B	B	B
HotKeyIndex	Button	•	B	B	B	B
HxJustify	Button	LEFT		LEFT	B	B
ItemType	•	•		B	•	•
SelectOnDoubleClick	Button	F		B	B	B
SelectOnDownClick	Button	F		B	B	B
SendMessageText	Button	•	B	B	B	B
SendMessageWhenSelected	Button	•	B	B	B	B
StringData	Button	•	B	B	B	B
Value	Button	•	B	B	B	B
VtJustify	Button	CENTER		B	B	B

ProgressBar

ZafProgressBar Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
ProgressData	•	•	•	N	•	•
ProgressStyle	•	•		NATIVE	•	•
ProgressType	•	•		HORIZON- TAL	•	•
TextStyle	•	•		TEXT - VALUE	•	•

Prompt

ZafPrompt Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoSize	•	•		T	•	•
HotKeyChar	•	•		Ø	•	•
HotKeyIndex	•	•		-1	•	•

ZafPrompt Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
HzJustify	•	•		LEFT	•	•
StringData	•	•	•	N	•	•
TransparentBackground	•	•		F	•	
VtJustify	•	•		TOP	•	•

PullDownItem

ZafPullDownItem Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowDefault	Button	F		B	B	B
AllowToggling	Button	F		B	B	B
AutoRepeatSelection	Button	F		F	F	F
AutoSize	Button	F		B	B	B
BitmapData	Button	N		B	B	B
ButtonType	Button	FLAT		FLAT	B	B
Depressed	Button	internal		B	B	B
Depth	Button	Ø		Ø	B	B
HotKeyChar	Button	•	B	B	B	B
HotKeyIndex	Button	•	B	B	B	B
HzJustify	Button	CENTER		B	B	B
SelectOnDoubleClick	Button	F		B	B	B
SelectOnDownClick	Button	F		B	B	B
SendMessageText	Button	N		B	B	B
SendMessageWhenSelected	Button	F		B	B	B
StringData	Button	•	B	B	B	B
Value	Button	Ø		B	B	B
VtJustify	Button	CENTER		B	B	B

Real

ZafReal Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowInvalid	String	•	B	B	B	B
AutoClear	String	•	B	B	B	B
HzJustify	String	•	B	B	B	B
InputFormatData	String	•	B	B	B	B
Invalid	String	•	B	B	B	B
LowerCase	String	F		B	B	B
OutputFormatData	String	•	B	B	B	B
Password	String	F		B	B	B
RangeData	String	•	B	B	B	B
RealData	•	•	•	N	•	•
ReportInvalid	String	•	B	B	B	B
StringData	String	INVALID		B	B	B
Unanswered	String	•	B	B	B	B
UpperCase	String	F		B	B	B

ZafReal Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
VariableName	String	F		B	B	B
ViewOnly	String	•	B	B	B	B

ScrollBar

ZafScrollBar Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoSize	•	•		T	•	•
ScrollData	•	•	•	N	•	•
ScrollType	•	•		VERTICAL	•	•

Scrolled Window

ZafScrolledWindow Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
HzScrollPos	•	•	•	parameter	•	•
ScrollHeight	•	•	•	parameter	•	•
ScrollWidth	•	•	•	parameter	•	•
VtScrollPos	•	•	•	parameter	•	•

SpinControl

ZafSpinControl Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
Delta	•	•	•	parameter	•	•
ViewOnly	•	•		F	•	•
WrappedData	•	•	•	T	•	•

Splitter

ZafSplitter Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
Live	•	• (F on Motif)	•	T (F on Motif)	• (F on Motif)	• (F on Motif)
NextPaneObject	•	•	•	N	N	N
Position	•	•	•	50 percent	•	•
PreviousPaneObject	•	•	•	N	N	N
SplitterType	•	•		parameter	•	•
Thickness	•	•		3	•	•

String

ZafString Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowInvalid	•	•	•	F	•	•
AutoClear	•	•	•	T	•	•
HzJustify	•	•		LEFT	•	•
InputFormatData	•	•		N	•	•

ZafString Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
Invalid	•	•	•	F	•	•
LowerCase	•	•		F	•	•
OutputFormatData	•	•		N	•	•
Password	•	•		F	•	•
RangeData	•	•	•	N	•	•
ReportInvalid	•	•	•	T	T	•
StringData	•	•	•	N	•	•
Unanswered	•	•	•	F	•	•
UpperCase	•	•		F	•	•
VariableName	•	•		F	•	•
ViewOnly	•	•		F	•	•

SystemButton

ZafSystemButton Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowDefault	Button	F		B	B	B
AllowToggling	Button	F		B	B	B
AutoRepeatSelection	Button	F		B	B	B
AutoSize	Button	T		B	B	B
BitmapData	Button	internal		B	B	B
ButtonType	Button	3D		3D	B	B
Depressed	Button	internal		B	B	B
Depth	Button	1		1	B	B
HotKeyChar	Button	Ø		B	B	B
HotKeyIndex	Button	-1		B	B	B
HzJustify	Button	CENTER		B	B	B
SelectOnDoubleClick	Button	F		B	B	B
SelectOnDownClick	Button	F		B	B	B
SendMessageText	Button	N		B	B	B
SendMessageWhenSelected	Button	F		T	B	B
StringData	Button	internal		B	B	B
SystemButtonType	•	•		NATIVE	•	•
Value	Button	Ø		B	B	B
VtJustify	Button	CENTER		B	B	B

Table

ZafTable Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
CurrentOffset	•	internal	user read- able	-1	-1	-1
Grid	•	•		T	•	•
HeaderHeight	•	•	•	parameter	ZafTable- Header	ZafTable- Header
HeaderWidth	•	•	•	parameter	ZafTable- Header	ZafTable- Header
MaxOffset	•	•	•	-1	•	•

ZafTable Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
ReadFunction	•	•	•	Read-Record	•	Read-Record
RowHeight	•	•		parameter	ZafTable-Record	ZafTable-Record
VirtualRecord	•	•		N	•	•
WriteFunction	•	•	•	Write-Record	•	Write-Record

TableHeader

ZafTableHeader Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
HeaderType	•	•		ROW	•	•
VirtualField	•	•		N	•	•

TableRecord

ZafTableRecord Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
Offset	•	internal	user read-able	-1	-1	-1

Text

ZafText Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoClear	•	•	•	F	•	•
HzJustify	•	•		LEFT	•	•
Invalid	•	•	•	F	•	•
StringData	•	•	•	N	•	•
Unanswered	•	•	•	F	•	•
ViewOnly	•	•		F	•	•
WordWrap	•	•	•	T	•	•

Time

ZafTime Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowInvalid	String	•	B	B	B	B
AutoClear	String	•	B	B	B	B
HzJustify	String	•	B	B	B	B
InputFormatData	String	•	B	B	B	B
Invalid	String	•	B	B	B	B
LowerCase	String	F		B	B	B
OutputFormatData	String	•	B	B	B	B
Password	String	F		B	B	B
RangeData	String	•	B	B	B	B
ReportInvalid	String	•	B	B	B	B
StringData	String	INVALID		B	B	B

ZafTime Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
TimeData	•	•	•	N	•	•
Unanswered	String	•	B	B	B	B
UpperCase	String	F		B	B	B
VariableName	String	F		B	B	B
ViewOnly	String	•	B	B	B	B

Title

ZafTitle Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowDefault	Button	F		B	B	B
AllowToggling	Button	F		B	B	B
AutoRepeatSelection	Button	F		B	B	B
AutoSize	Button	T		B	B	B
BitmapData	Button	N		B	B	B
ButtonType	Button	FLAT		FLAT	B	B
Depressed	Button	internal		B	B	B
Depth	Button	Ø		Ø	B	B
HotKeyChar	Button	Ø		B	B	B
HotKeyIndex	Button	-1		B	B	B
HzJustify	Button	CENTER		B	B	B
SelectOnDoubleClick	Button	T		T	B	B
SelectOnDownClick	Button	T		T	B	B
SendMessageText	Button	N		B	B	B
SendMessageWhenSelected	Button	T		T	B	B
StringData	Button	•	B	B	B	B
Value	Button	internal		B	B	B
VtJustify	Button	CENTER		B	B	B

ToolBar

ZafToolBar Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
DockType	•	•		TOP	•	•
WrapChildren	•	•		T	•	•

TreelItem

ZafTreelItem Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoSortData	•	•	•	F	•	•
Expandable	•	•		F	•	•
Expanded	•	•	•	F	•	•
NormalBitmap	•	•		N	•	•
SelectedBitmap	•	•		N	•	•
StringData	•	•	•	N	•	•

TreeList

ZafTreeList Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoSortData	•	•	•	F	•	•
DrawLines	•	•		T	•	•

UTime

ZafUTime Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AllowInvalid	String	•	B	B	B	B
AutoClear	String	•	B	B	B	B
HzJustify	String	•	B	B	B	B
InputFormatData	String	•	B	B	B	B
Invalid	String	•	B	B	B	B
LowerCase	String	F		B	B	B
OutputFormatData	String	•	B	B	B	B
Password	String	F		B	B	B
RangeData	String	•	B	B	B	B
ReportInvalid	String	•	B	B	B	B
StringData	String	INVALID		B	B	B
Unanswered	String	•	B	B	B	B
UpperCase	String	F		B	B	B
UTimeData	•	•	•	N	•	•
VariableName	String	F		B	B	B
ViewOnly	String	•	B	B	B	B

VtList

ZafVtList Property	Inherited From	Value	Dynamic	Constructor		
				Normal	Copy	Persistent
AutoSortData	•	•	•	F	•	•

ZAF 5 to 4 Class Comparisons

Zinc Application Framework 5 introduces a new API. Naming conventions and parameters have changed during this “once in a lifetime” rewrite. Core architectures and concepts remain largely unchanged from previous versions of ZAF, however.

Programmers using versions of ZAF prior to version 5 may find the following table useful when transitioning to version 5. In the table, each ZAF 5 class is listed along with an indication of similarity to a previous ZAF 4 class. All ZAF 5 classes are different from their previous incarnations, but differences vary significantly in degree.

- “Similar” indicates that the class is functionally very close to the ZAF 4 class.
- “Different” indicates that the class is functionally substantially different from the equivalent ZAF 4 class.
- “New” indicates that the class introduces entirely new functionality for ZAF 5.

Class	Similar	Different	New
ZafApplication	•		
ZafAttachment	•		
ZafBignum	•		
ZafBignumData		•	
ZafBitmapData		•	
ZafBitmapStruct			•
ZafBorder	•		
ZafButton		•	
ZafChart			•
ZafCodeSetData		•	
ZafComboBox	•		
ZafConstraint	•		
ZafCursor	•		
ZafData			•
ZafDataManager			•
ZafDataPersistence			•
ZafDataRecord			•
ZafDate	•		
ZafDateData		•	
ZafDevice	•		

Class	Similar	Different	New
ZafDialogWindow	•		
ZafDimensionConstraint	•		
ZafDiskFile			•
ZafDiskFileSystem			•
ZafDisplay	•		
ZafElement	•		
ZafEraStruct	•		
ZafErrorStub	•		
ZafErrorSystem	•		
ZafEventManager	•		
ZafEventMap	•		
ZafEventStruct	•		
ZafFile	•		
ZafFileDialog			•
ZafFileInfoStruct	•		
ZafFileSystem			•
ZafFormatData			•
ZafFormattedString	•		
ZafGeometryManager	•		
ZafGroup	•		
ZafHelpStub	•		
ZafHelpSystem	•		
ZafHelpTips			•
ZafHzList	•		
ZafI18nData			•
ZafIcon	•		
ZafIconData			•
ZafIconStruct			•
ZafImage	•		
ZafImageStruct			•
ZafInteger	•		
ZafIntegerData			•
ZafKeyboard	•		
ZafKeyStruct	•		

Class	Similar	Different	New
ZafLanguageData	•		
ZafLanguageManager	•		
ZafList	•		
ZafListBlock	•		
ZafLocaleData			•
ZafLocaleStruct	•		
ZafMaximizeButton	•		
ZafMDIWindow	•		
ZafMessageData	•		
ZafMessageStruct	•		
ZafMessageWindow	•		
ZafMinimizeButton	•		
ZafMouse	•		
ZafMouseData			•
ZafMouseStruct			•
ZafNotebook	•		
ZafNotification			•
ZafObjectPersistence			•
ZafPaletteData			•
ZafPaletteMap	•		
ZafPaletteStruct		•	
ZafPath	•		
ZafPathElement	•		
ZafPopUpItem	•		
ZafPopUpMenu	•		
ZafPositionStruct	•		
ZafPrinter	•		
ZafProgressBar			•
ZafPrompt	•		
ZafPullDownItem	•		
ZafPullDownMenu	•		
ZafQueueBlock	•		
ZafQueueElement	•		
ZafRegionStruct	•		

Class	Similar	Different	New
ZafReal	•		
ZafRealData			•
ZafRelativeConstraint	•		
ZafScreenDisplay	•		
ZafScrollBar	•		
ZafScrollData			•
ZafScrollStruct	•		
ZafScrolledWindow			•
ZafSpinControl	•		
ZafStandardArg			•
ZafStatusBar	•		
ZafStorage		•	
ZafStorageFile		•	
ZafString	•		
ZafStringData		•	
ZafSystemButton	•		
ZafTable		•	
ZafTableHeader		•	
ZafTableRecord		•	
ZafText	•		
ZafTime	•		
ZafTimeData		•	
ZafTimer	•		
ZafTitle	•		
ZafToolBar	•		
ZafTreeItem			•
ZafTreeList			•
ZafUTime			•
ZafUTimeData			•
ZafVtList	•		
ZafWindow	•		
ZafWindowManager	•		
ZafWindowObject	•		

Character Maps

This appendix lists the character maps supported by Zinc Application Framework in the file I18n.znc. See [ZafCodeSetData](#) for information on supporting character maps.

ISO 8859-1 Character Maps (Code Sets)
ANSI/Latin 1 Windows (CP 1252)
Arabic Windows (CP 1256)
Canadian-French DOS (CP 863)
Cyrillic DOS (CP 855)
Cyrillic Windows (CP 1251)
Eastern European Windows (CP 1250)
Greek DOS (CP 869)
Greek Windows (CP 1253)
Hebrew Windows (CP 1255)
Iceland DOS (CP 861)
Latin 1 DOS (CP 850)
Macintosh
Nordic DOS (CP 865)
Portuguese DOS (CP 860)
Slavic DOS (CP 852)
Turkish DOS (CP 857)
Turkish Windows (CP 1254)
United States DOS (CP 437)

Unicode Character Maps (Code Sets)
ANSI/Latin 1 Windows
Arabic Windows
BIG 5
Canadian-French DOS
Cyrillic DOS
Cyrillic Windows
Eastern European Windows

Unicode Character Maps (Code Sets)

EUC-JIS
GB 2312
Greek DOS
Greek Windows
Hebrew Windows
IBM 5550
Iceland DOS
KSC 5601
Latin 1 DOS
Macintosh
Nordic DOS
Portuguese DOS
Shift-JIS
Slavic DOS
Turkish DOS
Turkish Windows
United States DOS

ISO Country Codes

This appendix lists the ISO country codes. Zinc will maintain compatibility with the ISO definitions as they are updated or, in certain cases, before they are officailly adopted if it is evident that a proposed standard will be adopted. Please be aware that the inclusion of a country code in this table does not imply support for that country code by Zinc Application Framework. This table is the complete ISO table.

These codes are used by Zinc for identifying a particular country, or if necessary, a locale within a country. The locale identified by these codes will affect the formatting of dates and times, and the display of symbols (such as monetary symbols). The codes are from the ISO3166 standard.

“†” Indicates that support for the country is already included in the I18n.znc file. See [ZafLocaleData](#) for information on supporting countries.

Country	ISO Code
AFGHANISTAN	AF
ALBANIA	AL
ALGERIA	DZ
AMERICAN SAMOA	AS
ANDORRA	AD
ANGOLA	AO
ANGUILLA	AI
ANTARCTICA	AQ
ANTIGUA AND BARBUDA	AG
ARGENTINA	AR
ARMENIA	AM
ARUBA	AW
AUSTRALIA	AU
AUSTRIA	AT [†]
AZERBAIJAN	AZ
BAHAMAS	BS
BAHRAIN	BH
BANGLADESH	BD
BARBADOS	BB
BELARUS	BY
BELGIUM	BE

Country	ISO Code
BELIZE	BZ
BENIN	BJ
BERMUDA	BM
BHUTAN	BT
BOLIVIA	BO
BOSNIA AND HERZEGOWINA	BA
BOTSWANA	BW
BOUVET ISLAND	BV
BRAZIL	BR
BRITISH INDIAN OCEAN TERRITORY	IO
BRUNEI DARUSSALAM	BN
BULGARIA	BG
BURKINA FASO	BF
BURUNDI	BI
CAMBODIA	KH
CAMEROON	CM
CANADA	CA [†]
CAPE VERDE	CV
CAYMAN ISLANDS	KY
CENTRAL AFRICAN REPUBLIC	CF
CHAD	TD
CHILE	CL
CHINA	CN [†]
CHRISTMAS ISLAND	CX
COCOS (KEELING) ISLANDS	CC
COLOMBIA	CO
COMOROS	KM
CONGO	CG
COOK ISLANDS	CK
COSTA RICA	CR
COTE D'IVOIRE	CI
CROATIA (local name: Hrvatska)	HR
CUBA	CU
CYPRUS	CY

Country	ISO Code
CZECH REPUBLIC	CZ
DENMARK	DK [†]
DJIBOUTI	DJ
DOMINICA	DM
DOMINICAN REPUBLIC	DO
EAST TIMOR	TP
ECUADOR	EC
EGYPT	EG
EL SALVADOR	SV
EQUATORIAL GUINEA	GQ
ERITREA	ER
ESTONIA	EE
ETHIOPIA	ET
FALKLAND ISLANDS (MALVINAS)	FK
FAROE ISLANDS	FO
FIJI	FJ
FINLAND	FI [†]
FRANCE	FR [†]
FRANCE, METROPOLITAN	FX
FRENCH GUIANA	GF
FRENCH POLYNESIA	PF
FRENCH SOUTHERN TERRITORIES	TF
GABON	GA
GAMBIA	GM
GEORGIA	GE
GERMANY	DE [†]
GHANA	GH
GIBRALTAR	GI
GREECE	GR [†]
GREENLAND	GL
GRENADA	GD
GUADELOUPE	GP
GUAM	GU
GUATEMALA	GT

Country	ISO Code
GUINEA	GN
GUINEA-BISSAU	GW
GUYANA	GY
HAITI	HT
HEARD AND MCDONALD ISLANDS	HM
HONDURAS	HN
HONG KONG	HK
HUNGARY	HU
ICELAND	IS
INDIA	IN
INDONESIA	ID
IRAN (ISLAMIC REPUBLIC OF)	IR
IRAQ	IQ
IRELAND	IE
ISRAEL	IL
ITALY	IT [†]
JAMAICA	JM
JAPAN	JP [†]
JORDAN	JO
KAZAKHSTAN	KZ
KENYA	KE
KIRIBATI	KI
KOREA, DEMOCRATIC PEOPLE'S REPUBLIC OF	KP
KOREA, REPUBLIC OF	KR [†]
KUWAIT	KW
KYRGYZSTAN	KG
LAO PEOPLE'S DEMOCRATIC REPUBLIC	LA
LATVIA	LV
LEBANON	LB
LESOTHO	LS
LIBERIA	LR
LIBYAN ARAB JAMAHIRIYA	LY
LIECHTENSTEIN LI	LIE
LITHUANIA	LT

Country	ISO Code
LUXEMBOURG	LU
MACAU	MO
MACEDONIA, THE FORMER YUGOSLAV REPUBLIC OF	MK
MADAGASCAR	MG
MALAWI	MW
MALAYSIA	MY
MALDIVES	MV
MALI	ML
MALTA	MT
MARSHALL ISLANDS	MH
MARTINIQUE	MQ
MAURITANIA	MR
MAURITIUS	MU
MAYOTTE	YT
MEXICO	MX [†]
MICRONESIA, FEDERATED STATES OF	FM
MOLDOVA, REPUBLIC OF	MD
MONACO	MC
MONGOLIA	MN
MONTSERRAT	MS
MOROCCO	MA
MOZAMBIQUE	MZ
MYANMAR	MM
NAMIBIA	NA
NAURU	NR
NEPAL	NP
NETHERLANDS	NL [†]
NETHERLANDS ANTILLES	AN
NEW CALEDONIA	NC
NEW ZEALAND	NZ
NICARAGUA	NI
NIGER	NE
NIGERIA	NG
NIUE	NU

Country	ISO Code
NORFOLK ISLAND	NF
NORTHERN MARIANA ISLANDS	MP
NORWAY	NO [†]
OMAN	OM
PAKISTAN	PK
PALAU	PW
PANAMA	PA
PAPUA NEW GUINEA	PG
PARAGUAY	PY
PERU	PE
PHILIPPINES	PH
PITCAIRN	PN
POLAND	PL
PORTUGAL	PT
PUERTO RICO	PR
QATAR	QA
REUNION	RE
ROMANIA	RO
RUSSIAN FEDERATION	RU
RWANDA	RW
SAINT KITTS AND NEVIS	KN
SAINT LUCIA	LC
SAINT VINCENT AND THE GRENADINES	VC
SAMOA	WS
SAN MARINO	SM
SAO TOME AND PRINCIPE	ST
SAUDI ARABIA	SA
SENEGAL	SN
SEYCHELLES	SC
SIERRA LEONE	SL
SINGAPORE	SG
SLOVAKIA (Slovak Republic)	SK
SLOVENIA	SI
SOLOMON ISLANDS	SB

Country	ISO Code
SOMALIA	SO
SOUTH AFRICA	ZA
SPAIN	ES [†]
SRI LANKA	LK
ST. HELENA	SH
ST. PIERRE AND MIQUELON	PM
SUDAN	SD
SURINAME	SR
SVALBARD AND JAN MAYEN ISLANDS	SJ
SWAZILAND	SZ
SWEDEN	SE [†]
SWITZERLAND	CH
SYRIAN ARAB REPUBLIC	SY
TAIWAN, PROVINCE OF CHINA	TW [†]
TAJIKISTAN	TJ
TANZANIA, UNITED REPUBLIC OF	TZ
THAILAND	TH
TOGO	TG
TOKELAU	TK
TONGA	TO
TRINIDAD AND TOBAGO	TT
TUNISIA	TN
TURKEY	TR
TURKMENISTAN	TM
TURKS AND CAICOS ISLANDS	TC
TUVALU	TV
UGANDA	UG
UKRAINE	UA
UNITED ARAB EMIRATES	AE
UNITED KINGDOM	GB [†]
UNITED STATES	US [†]
UNITED STATES MINOR OUTLYING ISLANDS	UM
URUGUAY	UY
UZBEKISTAN	UZ

Country	ISO Code
VANUATU	VU
VATICAN CITY STATE (HOLY SEE)	VA
VENEZUELA	VE
VIET NAM	VN
VIRGIN ISLANDS (BRITISH)	VG
VIRGIN ISLANDS (U.S.)	VI
WALLIS AND FUTUNA ISLANDS	WF
WESTERN SAHARA	EH
YEMEN	YE
YUGOSLAVIA	YU
ZAIRE	ZR
ZAMBIA	ZM
ZIMBABWE	ZW

ISO Language Codes

This appendix lists the ISO language codes. Zinc will maintain compatibility with the ISO definitions as they are updated or, in certain cases, before they are officailly adopted if it is evident that a proposed standard will be adopted. Please be aware that the inclusion of a language code in this table does not imply support for that language code by Zinc Application Framework. This table is the complete ISO table.

These codes are used by Zinc for identifying a particular language. The language identified by these codes will be used when displaying text on objects in the library. The codes are from the ISO639 standard.

“*” Signifies a proposed language code. “†” Indicates that support for the language is already included in the I18n.znc file. See [ZafLanguageData](#) for information on supporting languages.

Language	ISO Code
Abkhazian	ab
Afar	aa
Afrikaans	af
Albanian	sq
Amharic	am
Arabic	ar
Armenian	hy
Assamese	as
Aymara	ay
Azerbaijani	az
Bashkir	ba
Basque	eu
Bengali (Bangla)	bn
Bhutani	dz
Bihari	bh
Bislama	bi
Breton	br
Bulgarian	bg
Burmese	my
Byelorussian	be
Cambodian	km

Language	ISO Code
Catalan	ca [†]
Chinese	zh
Corsican	co
Croatian	hr
Czech	cs
Danish	da [†]
Dutch	nl [†]
English	en [†]
Esperanto	eo
Estonian	et
Faeroese	fo
Farsi	fa
Fiji	fj
Finnish	fi [†]
French	fr [†]
Frisian	fy
Galician	gl
Georgian	ka
German	de [†]
Greek	el [†]
Greenlandic	kl
Guarani	gn
Gujarati	gu
Hausa	ha
Hebrew	iw, he
Hindi	hi
Hungarian	hu
Icelandic	is
Indonesian	in, id
Interlingua	ia
Interlingue	ie
Inuktitut	iu
Inupiak	ik

Language	ISO Code
Irish	ga
Italian	it [†]
Japanese	ja [†]
Javanese	jw
Kannada	kn
Kashmiri	ks
Kazakh	kk
Kinyarwanda	rw
Kirghiz	ky
Kirundi	rn
Korean	ko [†]
Kurdish	ku
Laothian	lo
Latin	la
Latvian (Lettish)	lv
Lingala	ln
Lithuanian	lt
Macedonian	mk
Malagasy	mg
Malay	ms
Malayalam	ml
Maltese	mt
Manx Gaelic	gv [*]
Maori	mi
Marathi	mr
Moldavian	mo
Mongolian	mn
Nauru	na
Nepali	ne
Norwegian	no [†]
Occitan	oc
Oriya	or
Oromo (Afan)	om
Pashto (Pushto)	ps

Language	ISO Code
Polish	pl
Portuguese	pt
Punjabi	pa
Quechua	qu
Rhaeto-Romance	rm
Romanian	ro
Russian	ru
Samoan	sm
Sangro	sg
Sanskrit	sa
Scots Gaelic	gd
Serbian	sr
Serbo-Croatian	sh
Sesotho	st
Setswana	tn
Shona	sn
Sindhi	sd
Singhalese	si
Siswati	ss
Slovak	sk
Slovenian	sl
Somali	so
Spanish	es [†]
Sundanese	su
Swahili	sw
Swedish	sv [†]
Tagalog	tl
Tajik	tg
Tamil	ta
Tatar	tt
Telugu	te
Thai	th
Tibetan	bo
Tigrinya	ti

Language	ISO Code
Tonga	to
Tsonga	ts
Turkish	tr
Turkmen	tk
Twi	tw
Uighur	ug
Ukrainian	uk
Urdu	ur
Uzbek	uz
Vietnamese	vi
Volapk	vo
Welsh	cy
Wolof	wo
Xhosa	xh
Yiddish	ji, yi
Yoruba	yo
Zulu	zu

X Resources

As part of the internal implementation of Zinc Application Framework 5 for Motif, ZAF derives from Motif “xm*” classes and registers its own versions. Knowledge of this internal implementation may be useful for programmers who wish to change colors, fonts, etc. in Motif resource files rather than hard-coding them into an application.

In general, a Motif application that wants to use resources should do so as defined in the O’Reilly & Associates Motif manuals:

- Vol IV, Chapter 2.3.3—The App-defaults file
- Vol IV, Chapter 10—Resource Management

These chapters describe the types of operations and files that can be overridden, and from what class. This information is also available in various end-user Motif or system administration manuals.

The table below lists ZAF 5 window object class names and the corresponding Motif or custom class names that are registered with Motif. These names may be used as resource tags when overriding default widget operations. Also listed are Motif classes used as an immediate base class by ZAF.

Sample usage is presented following the table.

ZAF Class Name	Motif Class Name	ZAF class derives from
ZafWindowObject	“ZafWindowObject”	xmPrimitiveClassRec
ZafWindow	“ZafWindowShell” (shell)	topLevelShellClassRec
	“ZafWindowFrame” (frame)	xmBulletinBoardClassRec
	“ZafWindow” (client)	xmBulletinBoardClassRec
ZafButton	“ZafButton”	xmPushButtonClassRec
ZafComboBox	“ZafComboBox”	xmBulletinBoardClassRec
ZafDialogWindow	see ZafWindow	
ZafFileDialog	“ZafWindowShell”	xmFileSelectionBoxWidget
ZafFormattedString	“ZafFormattedString”	xmTextFieldClassRec
ZafGroup	see ZafString	
ZafHzList	see ZafVtList	
ZafIcon	uses Icon registration	
ZafMDIWindow	see ZafWindow	
ZafNotebook	“ZafNotebook”	xmBulletinBoardClassRec

ZAF Class Name	Motif Class Name	ZAF class derives from
ZafPullDownMenu	“xmRowColumn”	xmRowColumnClassRec
ZafPullDownItem	“xmCascadeButton”	xmCascadeButtonClassRec
ZafPopUpMenu	“popupShell”	xmRowColumnWidget- ClassRec
ZafPopUpItem	“ZafToggleItem” (no children)	xmToggleButtonClassRec
	“ZafCascadeItem” (with children)	xmCascadeButtonClassRec
ZafPrompt	“ZafPrompt”	xmLabelClassRec
ZafScrolledWindow	two “ZafWindow”s	
ZafScrollBar	“ZafScrollBar” (scroll- bar)	xmScrollBarClassRec
	“ZafSlider” (slider)	xmScaleClassRec
ZafSpinControl	“ZafSpinControl”	xmBulletinBoardClassRec
ZafStatusBar	see ZafWindow	
ZafString	“ZafString”	xmTextFieldClassRec
ZafTable	see ZafWindow	
ZafTableHeader	see ZafWindow	
ZafTableRecord	see ZafWindow	
ZafToolBar	see ZafWindow	
ZafText	“ZafText”	xmTextClassRec
ZafTreeList	“ZafTreeList”	xmListClassRec
ZafVtList	“ZafList”	xmListClassRec

When overriding operations for a program named “hello”, for example, a resource file named “Hello” or “XHello” should be created. In this file a user might include overrides such as:

```
*ZafPrompt.fontList: *-helvetica-bold-r-*-140-*-*-*-*-*
*ZafPrompt.foreground: blue
*ZafString.background: yellow
*ZafWindow.background: green
```

When adding definitions to resource files, note that Motif allows resources to affect derived classes. Therefore, overrides for a base xmBulletinBoardClassRec can affect ZAF classes that use the xmBulletinBoardClassRec definition. For example:

```
*xmLabel.foreground: black
*xmBulletinBoard.fontlist: *-times-normal-r-*-140-*-*-*-*-*
```

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

BROKEN STATEMENTS

If a statement cannot fit on one line on the screen, it should be broken with the subsequent lines of the statement indented one space farther over than the first line. It should be split after a comma or logic symbol if possible. Several examples of this calling convention are shown below:

```

if (veryLongBooleanA && veryLongBooleanB && veryLongBooleanC &&
    veryLongBooleanD)
    DoSomething();

for (int veryLongLoopCounter = 0;
    veryLongLoopCounter < 10;
    veryLongLoopCounter++)
{
    if (CheckTheVeryLongLoopCounter(veryLongLoopCounter) ==
        veryLongConstant)
        DoSomethingElse();
}

```

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 Z_BORDER.CPP's file layout is shown below:

```
// Zinc Application Framework - Z_BORDER.CPP
// COPYRIGHT (C) 1990-1997. 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)
{
}

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) 1990-1997. 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. When a member must extend to the next line(s), each subsequent line should be indented an additional space. 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 space 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

- AbortJob
 - Printer 394
- AcceptDrop
 - WindowObject 614
- ActivateObject
 - TableRecord 524
- ActiveModel
 - Chart 75
- Add
 - List 297
 - PopUpItem 380
 - PullDownItem 413
 - SystemButton 503
 - Timer 540
 - Window 586
- AddColor
 - Display 149
- AddCompareFunction
 - ObjectPersistence 362
- AddDataConstructor
 - DataPersistence 113
- AddDataGroup
 - Chart 75
- AddDataPoint
 - Chart 75
- AddDepthItem
 - TreeList 560
- AddFileSystem
 - DataPersistence 113
- AddFont
 - Display 150
- AddGenericObjects
 - Window 587, 588
- AddNotification
 - Notification 352
- AddObjectConstructor
 - ObjectPersistence 362
- AddPalette
 - PaletteData 368
- AddStaticModule
 - I18nData 254
- AddUserCallback
 - ObjectPersistence 362
- AddUserObject
 - ObjectPersistence 363
- allFilesFilter
 - FileDialog 208
- Allocate
 - LanguageManager 294
- AllocateData
 - DataManager 109
- AllocateFile
 - DataPersistence 114
- AllowDefault
 - Button 61
- AllowInvalid
 - String 481
- AllowModifyCollate
 - PrintDialog 391
- AllowModifyCopies
 - PrintDialog 391
- AllowModifyRange
 - PrintDialog 391
- AllowToggling
 - Button 62
- altDigits
 - LocaleStruct 304
- AltPressed
 - Keyboard 286
- Append
 - StringData 492
- Application 22
- argc
 - Application 24
- argv
 - Application 24
- Array
 - BitmapData 51
 - IconData 264
 - MouseData 335
- array
 - ImageStruct 274
- Assign
 - PositionStruct 388
 - RegionStruct 430
- Attachment 27
- attributes
 - FileInfoStruct 211
- AutoClear
 - String 481
 - Text 528
- AutomaticUpdate
 - Window 588
 - WindowObject 617
- AutoRepeatSelection
 - Button 62
- AutoSelect
 - Group 234
- AutoSize
 - Bitmap 49
 - Button 63
 - Image 269
 - Prompt 408
 - ScrollBar 445
- AutoSortData
 - ComboBox 92

HzList 249
 TreeItem 553
 TreeList 561
 VtList 581

available3D
 ZafMSWindowsApp .. 341

AxisColor
 Chart 76

B

Background
 Display 152

BackgroundColor
 WindowObject 619

BasisYear
 UTimeData 573

Beep
 ErrorSystem 182

BeginDraw
 Display 152
 WindowObject 620

beginGregorian
 LocaleStruct 304

BeginJob
 Printer 394

BeginPage
 Printer 394

Bignum 32
 Property Matrix B-779

BignumData 37
 Bignum 34

BinaryMode
 File 201

Bitmap 47

Display 152
 Property Matrix B-779

BitmapData 50
 Bitmap 49
 Button 63

BitmapStruct 53

blankString
 LanguageManager 294

BlinkRate
 Cursor 100

Blocked
 EventManager 187

Border 55
 Property Matrix B-779
 Window 589

Bordered
 WindowObject 621

bottom
 RegionStruct 431

BottomOffset
 Table 512

BroadcastEvent
 Window 589

Button 58
 Property Matrix B-779

ButtonType
 Button 64

C

canonicalLocale
 LocaleData 302

CaptureMouse
 WindowManager 605

CellHeight

Display 153
 HzList 250

CellWidth
 Display 153
 HzList 250

Center
 RelativeConstraint 436
 WindowManager 605

Changed
 StorageFile 476
 Window 590
 WindowObject 622

ChangeExtension
 DiskFileSystem 142

Char
 StringData 492

Chart 72

ChartStub 81

ChartType
 Chart 76

ChDir
 DiskFileSystem 142
 FileSystem 214
 Storage 472

ClassID
 Element 172
 WindowObject 623

classID
 WindowObject 623

ClassName
 Element 172
 WindowObject 623

className
 WindowObject 623

Clear
 BignumData 39
 BitmapData 51

-
- CodeSetData 84
 - Data 104
 - DataRecord 120
 - DateData 128
 - IconData 264
 - IntegerData 281
 - LanguageData 290
 - LocaleData 302
 - MouseData 335
 - PaletteData 368
 - RealData 427
 - ScrollData 448
 - TimeData 538
 - UTimeData 573
 - ClearCompareFunctions
 - ObjectPersistence 363
 - ClearDataAll
 - Chart 76
 - ClearDataConstructors
 - DataPersistence 114
 - ClearDataGroup
 - Chart 77
 - ClearFileSystems
 - DataPersistence 114
 - ClearImage
 - Button 65
 - ClearMessageFlags
 - MessageWindow 323
 - ClearNotifications
 - Notification 352
 - ClearObjectConstructors
 - ObjectPersistence 363
 - ClearText
 - Button 65
 - ClearUserCallbacks
 - ObjectPersistence 363
 - ClearUserObjects
 - ObjectPersistence 363
 - ClientRegion
 - Window 591
 - ClipRegion
 - Display 153
 - Close
 - DiskFileSystem 143
 - FileSystem 214
 - Storage 472
 - CodeSetAllocate
 - CodeSetData 84
 - CodeSetData 82
 - CodeSetFree
 - CodeSetData 84
 - CodeSetName
 - CodeSetData 84
 - colorBackground
 - PaletteStruct 371
 - colorForeground
 - PaletteStruct 371
 - ColorInfo
 - Display 154
 - colorTable
 - ScreenDisplay 440
 - column
 - PositionStruct 388
 - columns
 - Display 154
 - ComboBox 89
 - Property Matrix B-780
 - commonControlsAvailable
 - ZafMSWindowsApp .. 341
 - Compare
 - StringData 493
 - CompareAscending
 - Window 591
 - CompareDescending
 - Window 591
 - CompareFunction
 - List 297
 - CompressedData
 - FormattedString 223
 - Constraint 94
 - Control
 - Application 24
 - DialogWindow 135
 - Convert 729
 - Convert Utility
 - ZAF 4 729
 - ConvertCoordinates
 - PositionStruct 388
 - RegionStruct 430
 - WindowObject 623
 - converted
 - EventStruct 195
 - ConvertRegion
 - WindowObject 624
 - convertText
 - ZafMSWindowsApp .. 341
 - ConvertToDrawRegion
 - WindowObject 624
 - ConvertToObjectPosition
 - WindowObject 624
 - ConvertToOSBitmap
 - Display 155
 - ConvertToOSChar
 - CodeSetData 85
 - ConvertToOSIcon
 - Display 155

ConvertToOSMouse Display 155	CoordinateType Display156 WindowObject627	Ctl3dUnregister ZafMSWindowsApp .. 342
ConvertToOSPosition WindowObject 625	coordinateType PositionStruct388 RegionStruct430	CtrlPressed Keyboard 286
ConvertToOSRegion WindowObject 625	CopyDraggable WindowObject634	currencySymbol LocaleStruct 305
ConvertToOSSString CodeSetData 85	CornerScrollBar Window591	Current List 298 PopUpItem 381 ProgressBar402 PullDownItem 414 ScrollData448 SystemButton504
ConvertToOSWChar CodeSetData 85	Count List298 PopUpItem381 PullDownItem414 SystemButton504	current ScrollStruct450
ConvertToOSWString CodeSetData 85	CountryCodeToZafLocale I18nData254	CurrentClassID DataPersistence 114
ConvertToScreenPosition WindowObject 624	Create File201 Storage472	CurrentClassName DataPersistence 114
ConvertToZafBitmap Display 155	CreateSubclassedWindow ZafMSWindowsApp ..341	currentDirectoryName FileSystem 214
ConvertToZafChar CodeSetData 86	creditLeftParen LocaleStruct304	CurrentFile DataPersistence 115
ConvertToZafEvent WindowObject 626	creditRightParen LocaleStruct305	CurrentFileSystem DataPersistence 115
ConvertToZafIcon Display 155	Ctl3dAutoSubclass ZafMSWindowsApp ..342	CurrentLanguage DataPersistence 115
ConvertToZafMouse Display 155	Ctl3dEnabled ZafMSWindowsApp ..341	CurrentOffset Table512
ConvertToZafPosition WindowObject 625	ctl3dModule ZafMSWindowsApp ..341	CurrentPage Notebook 348
ConvertToZafRegion WindowObject 625	Ctl3dRegister ZafMSWindowsApp ..341	Cursor99
ConvertToZafString CodeSetData 86	Ctl3dSubclassCtl ZafMSWindowsApp ..342	CursorOffset String481 Text528
ConvertXValue Display 156		CursorPosition
ConvertYValue Display 156		
CoordinateStruct 97		

- Text 528
- ## D
- Data 102
StorageFile 476
- data
EventStruct 195
- DataGroup
Chart 77
- DataGroupBackgroundColor
Chart 77
- DataGroupFillPattern
Chart 77
- DataGroupFont
Chart 77
- DataGroupForegroundColor
Chart 78
- DataGroupLineStyle
Chart 78
- DataManager 108
- DataPersistence 110
- DataPoint
Chart 78
- DataRecord 119
- Date 121
Property Matrix B-780
- DateData 126
Date 124
- dateSeparator
LocaleStruct 305
- dateStringInputFormat
LocaleStruct 305
- dateStringOutputFormat
LocaleStruct 305
- dateTimeStringInputFormat
LocaleStruct 305
- dateTimeStringOutputFormat
LocaleStruct 305
- Day
UTimeData 573
- DayName
UTimeData 573
- DayOfWeek
UTimeData 573
- DayOfYear
UTimeData 573
- DaysInMonth
UTimeData 573
- DaysInYear
UTimeData 574
- DeactivateObject
TableRecord 524
- decimalSeparator
LocaleStruct 305
- Decrement
ProgressBar 402
ScrollData 448
String 481
- DefaultButton
Window 592
- DefaultEventRoute
WindowManager 605
- defaultFilterName
Image 269
- DefaultLeadByte
CodeSetData 87
- DefaultMessageFlag
MessageWindow 323
- DefaultUserFunction
WindowObject 630
- DefaultValidateFunction
String 482
- defDigits
LocaleStruct 306
- DeleteData
FormattedString 224
- DeleteDriveNames
DiskFileSystem 143
- DeleteRecord
Table 512
- Delta
ProgressBar 402
ScrollData 448
SpinControl 458
- delta
ScrollStruct 450
- Depressed
Button 65
- Depth
Button 65
- DepthCurrent
TreeItem 553
TreeList 561
- DepthFirst
TreeItem 553
TreeList 561
- DepthLast
TreeItem 554
TreeList 562
- DepthNext
TreeItem 554
- DepthPrevious
TreeItem 554
- DerivedAccess

File	202	DimensionConstraint	136	dragObject WindowManager	605
Destroy		direction		dragStartPosition ZafMSWindowsApp ..	342
List	298	EraStruct	180	dragTest ZafMSWindowsApp ..	342
PopUpItem	381	Directory		Draw	
PullDownItem	414	FileDialog	209	WindowObject	635
SystemButton	504	FileInfoStruct	211	DrawBackground WindowObject	635
Window	592	dirSepStr		DrawBorder WindowObject	635
Destroyable		CodeSetData	87	DrawContext Display	157
Data	105	Disabled		DrawFocus WindowObject	636
Window	593	WindowObject	631	DrawLines TreeList	562
DestroyColor		DiskFile	140	DrawShadow WindowObject	636
Display	156	DiskFileSystem	142	dropDownCombo ZafMSWindowsApp ..	342
ZafDisplay	149	Display	147	Duplicate	
DestroyEvent		EventStruct	196	Data	105
EventManager	187	WindowObject	632	WindowObject	638
DestroyFont		display		Dynamic Properties	
Display	156	Device	130	Property Matrices ...	B-773
ZafDisplay	150	WindowObject	632	DynamicOSText StringData	493
DestroyOSBitmap		DisplayContext		DynamicOSWText StringData	493
Display	156	Display	157	DynamicPtrCast	709
DestroyOSIcon		DisplayHelp		DynamicText StringData	493
Display	156	HelpStub	237		
DestroyOSMouse		HelpSystem	240		
Display	156	DisplayMode			
DestroyZafBitmap		Display	157		
Display	157	DisplayType			
DestroyZafIcon		Display	158		
Display	157	DockType			
DestroyZafMouse		ToolBar	548		
Display	157	double			
Device	129	BignumData	41		
DeviceState		RealData	428		
Device	130	DragDropEvent			
DeviceType		WindowObject	633		
Device	131	Draggable			
DialogWindow	133	WindowObject	634		

E

- EditMode
 - WindowObject 639
- Element 170
- Ellipse
 - Display 158
- Encompassed
 - RegionStruct 430
- endDate
 - EraStruct 180
- EndDraw
 - WindowObject 620
- EndJob
 - Printer 395
- EndPage
 - Printer 395
- eraFormat
 - EraStruct 180
- eraName
 - EraStruct 180
- EraStruct 180
- eraTable
 - LocaleStruct 306
- eraTableLength
 - LocaleStruct 306
- Error
 - Application 25
 - Data 105
 - File 202
 - FileSystem 214
 - WindowObject 639
- ErrorMessage
 - ErrorStub 182
 - ErrorSystem 185
- errorString
- EventManager 294
- ErrorStub 182
- ErrorSystem 184
- EvaluateIsA
 - Element 173
- Event
 - Attachment 29
 - Bignum 35
 - Button 65
 - Constraint 96
 - Cursor 100
 - Data 107
 - Date 125
 - DimensionConstraint 138
 - EventManager 187
 - GeometryManager 230
 - HelpSystem 240
 - HelpTips 244
 - Integer 278
 - Keyboard 286
 - Mouse 330
 - PopUpItem 381
 - PullDownItem 414
 - Real 425
 - RelativeConstraint 436
 - String 482
 - Text 528
 - Time 535
 - Timer 541
 - UTime 569
 - Window 593
 - WindowManager 606
 - WindowObject 641
- event
 - QueueElement 421
- EventManager 186
 - EventStruct 196
- eventManager
 - WindowObject 646
- EventMap 191
- Events
- A_CLOSE_WINDOW .A-771
- A_HELP_CONTEXT A-771
- A_MINIMIZE_WINDOWS A-772
- A_OPEN_DOCUMENT A-772
- A_OPEN_WINDOW A-772
- A_PRINT_DOCUMENT A-772
- A_RESET_I18N A-772
- A_RESTORE_WINDOWS A-772
- D_ACTIVATE A-770
- D_DEACTIVATE .. A-770
- D_DEINITIALIZE .A-770
- D_HIDE A-770
- D_INITIALIZE A-770
- D_OFF A-770
- D_ON A-770
- D_POSITION A-770
- D_STATE A-769
- DC_INSERT A-770
- DC_OVERSTRIKE .A-770
- DH_HELP_TIPS_TIMER .A-770
- DH_SET_HELP_OBJECT A-769
- DH_UPDATE_HELP_OBJECT A-769
- DM_BOTTOM_LEFT_CORNER A-769
- DM_BOTTOM_RIGHT_CORNER A-769
- DM_BOTTOM_SIDE .. A-769
- DM_CANCEL A-769
- DM_CROSS_HAIRS A-769
- DM_DRAG A-770
- DM_DRAG_COPY .A-770
- DM_DRAG_COPY_MULTIPLE A-771
- DM_DRAG_LINK .A-771
- DM_DRAG_LINK_MULTIPLE A-771
- DM_DRAG_MOVE A-771
- DM_DRAG_MOVE_MULTIPLE A-771
- DM_EDIT A-769
- DM_LEFT_SIDE ... A-769

DM_MOVE A-769	L_LASTA-765	N_NON_CURRENT A-757
DM_RIGHT_SIDE . A-769	L_LEFTA-765	N_RESET_I18N A-757
DM_SELECT A-770	L_MARK_BOL ...A-766	N_SIZEA-756, A-757
DM_TOP_LEFT_CORNER A-770	L_MARK_DOWN ..A-766	N_TIMER A-757
DM_TOP_RIGHT_CORNER A-770	L_MARK_EOLA-766	N_VSCROLL A-757
DM_TOP_SIDE A-770	L_MARK_LEFT ...A-766	S_ADD_OBJECT ..A-758
DM_VIEW A-770	L_MARK_PGDN ...A-767	S_BEGIN_DRAG ..A-760
DM_WAIT A-770	L_MARK_PGUP ...A-767	S_CLOSE A-758
E_CURSOR A-768	L_MARK_RIGHT ..A-767	S_CLOSE_TEMPORARY A-758
E_DEVICE A-768	L_MARK_UPA-767	S_COMPUTE_SIZE A-760
E_HELPTIPS A-768	L_MARK_WORD_LEFT A-767	S_CONTINUE A-758
E_KEY A-768	L_MARK_WORD_RIGHT A-767	S_COPY A-758
E_MOUSE A-768	L_MAXIMIZEA-767	S_COPY_DATA ...A-758
E_OSEVENT A-768	L_MDI_MOVE_MODE A- 767	S_CREATE A-761
E_TIMER A-769	L_MDI_NEXT_WINDOW A-765	S_CURRENT A-761
L_ALT_KEY A-766	L_MDI_SIZE_MODE ..A- 767	S_CUT A-758
L_BACKSPACE ... A-764	L_MINIMIZEA-767	S_DECREMENT ...A-758
L_BEGIN_ESCAPE A-764	L_MOVE_MODE ..A-767	S_DEINITIALIZE ..A-761
L_BOL A-764	L_NEXTA-765	S_DESTROYA-761
L_CANCEL A-764	L_NEXT_WINDOW A-767	S_DLG_ABORT ...A-762
L_CLOSE A-764	L_NONEA-767	S_DLG_CANCEL ..A-762
L_CLOSE_TEMPORARY A-764	L_PASTEA-765	S_DLG_IGNORE ..A-762
L_CONTINUE_ESCAPE . A-764	L_PGDNA-765	S_DLG_NO A-762
L_CONTINUE_SELECT . A-764	L_PGUPA-765	S_DLG_OK A-763
L_COPY A-764	L_PREVIOUSA-765	S_DLG_RETRY ...A-763
L_CUT A-764	L_RESTOREA-767	S_DLG_YES A-763
L_DELETE A-764	L_RIGHTA-765	S_DRAG_COPY ...A-761
L_DELETE_EOL .. A-766	L_SELECTA-765	S_DRAG_DEFAULT ..A- 761
L_DELETE_WORD A-766	L_SIZE_MODE ...A-767	S_DRAG_LINKA-761
L_DOUBLE_CLICK A-764	L_SYSTEM_MENU A-767	S_DRAG_MOVE ..A-761
L_DOWN A-764	L_TOGGLE_EXPANDED A-767	S_DROP_COPY ...A-761
L_END_ESCAPE .. A-764	L_UPA-765	S_DROP_DEFAULT A-761
L_END_SELECT .. A-764	L_VIEWA-766	S_DROP_LINKA-761
L_EOL A-765	L_WORD_LEFT ..A-768	S_DROP_MOVE ...A-761
L_EXIT A-764	L_WORD_RIGHT ..A-768	S_END_DRAGA-761
L_EXTEND_FIRST A-766	N_CHANGE_PAGE A-756	S_ERROR A-763
L_EXTEND_LAST A-766	N_CLOSEA-756	S_EXIT A-758
L_EXTEND_NEXT A-766	N_CURRENTA-756	S_HELP A-758
L_EXTEND_PREVIOUS . A-766	N_EXITA-756	S_HLP_CLOSEA-761
L_FIRST A-765	N_HSCROLLA-756	S_HLP_SELECT_TOPIC A-761
L_HELP A-765	N_MOUSE_ENTER A-756	S_HLP_SHOW_INDEX A- 761
L_INSERT_TOGGLE .A- 766	N_MOUSE_LEAVE A-757	S_HLP_SHOW_TOPIC A- 762
	N_MOVEA-757	S_HLP_UPDATE_NAME A-762

- S_HOT_KEY A-762
 - S_HSCROLL A-758
 - S_HSCROLL_CHECK A-758
 - S_HSCROLL_COMPUTE A-758
 - S_HSCROLL_SET .. A-758
 - S_INCREMENT ... A-758
 - S_INITIALIZE A-762
 - S_MAXIMIZE A-759
 - S_MDI_CASCADE_WINDOWS A-759
 - S_MDI_CLOSE A-759
 - S_MDI_MAXIMIZE A-759
 - S_MDI_MINIMIZE A-759
 - S_MDI_MOVE_MODE A-759
 - S_MDI_NEXT_WINDOW A-759
 - S_MDI_RESTORE .. A-759
 - S_MDI_SIZE_MODE .. A-759
 - S_MDI_TILE_WINDOWS A-759
 - S_MINIMIZE A-759
 - S_MOVE_MODE .. A-759
 - S_NEXT_WINDOW A-759
 - S_NO_OBJECT A-763
 - S_NON_CURRENT A-762
 - S_PASTE A-759
 - S_REDISPLAY A-759
 - S_REDISPLAY_DATA A-759
 - S_REDISPLAY_DEFAULT A-762
 - S_REDISPLAY_REGION A-759
 - S_REGISTER_OBJECT A-762, A-763
 - S_RESTORE A-760
 - S_SET_DATA A-760
 - S_SIZE A-760
 - S_SIZE_MODE A-760
 - S_SUBTRACT_OBJECT A-760
 - S_UNKNOWN A-763
 - S_VSCROLL A-760
 - S_VSCROLL_CHECK A-760
 - S_VSCROLL_COMPUTE A-760
 - S_VSCROLL_SET .. A-760
 - EventStruct193
 - eventType
 - EventMap191
 - ExitFunction
 - WindowManager606
 - Expandable
 - TreeItem554
 - Expanded
 - TreeItem554
 - ExpandedData
 - FormattedString224
 - Export
 - PositionStruct389
 - ExportPixel
 - RegionStruct431
 - ExportPoint
 - RegionStruct431
- ## F
- File200
 - FileDialog209
 - fileCreator
 - DiskFile140
 - FileDialog207
 - FileInfoStruct211
 - FileSystem213
 - fileType
 - DiskFile140
 - FillPattern
 - Display159
 - fillPattern
 - PaletteStruct372
 - Filter
 - FileDialog209
 - Find
 - Element174
 - List298
 - FindClose
 - DiskFileSystem143
 - FileSystem215
 - Storage472
 - FindFirst
 - DiskFileSystem143
 - FileSystem215
 - Storage473
 - FindNext
 - DiskFileSystem143
 - FileSystem215
 - Storage473
 - First
 - List298
 - PopUpItem382
 - PullDownItem414
 - SystemButton504
 - FirstFileSystem
 - DataPersistence115
 - FirstPage
 - Notebook348
 - Flush
 - Table513
 - Focus
 - WindowObject647
 - FocusObject
 - PopUpItem382
 - PullDownItem415
 - Window593
 - WindowObject648
 - focusObject
 - WindowManager607

FocusOffset
 Table 513

Font
 Display 159
 WindowObject 649

font
 PaletteStruct 373

FontInfo
 Display 159

fontList
 X Resources G-812

fontTable
 ScreenDisplay 440

Foreground
 Display 160

FormatData 218
 FormattedString 225

Formatted String
 Property Matrix B-780

FormattedString 221

FormattedText
 BignumData 40
 DateData 128
 FormatData 219
 IntegerData 282
 RealData 428
 TimeData 538
 UTimeData 574

fractionDigits
 LocaleStruct 306

FrameJumpProc
 ZafMSWindowsApp .. 343

FrameProc
 ZafMSWindowsApp .. 343

Free
 LanguageManager 294

Full
 ListBlock300

FullPath
 FileDialog209

G

GdiDisplay227

GeometryManager228
 Window593

Get
 EventManager187
 List299
 PopUpItem382
 PullDownItem415
 SystemButton504

GetClassName
 ObjectPersistence363

GetCompareFunction
 ObjectPersistence364

GetCompareFunctionName
 ObjectPersistence364

GetCWD
 DiskFileSystem143
 FileSystem216
 Storage473

GetDataConstructor
 DataPersistence116

GetDriveNames
 DiskFileSystem144

GetFile
 FileDialog209

GetMessage
 LanguageData290, 291

GetMessageData
 LanguageData291

GetObject
 Data 107
 DataRecord 120
 EventManager 188
 PopUpItem 382
 PullDownItem 415
 WindowObject 650

GetObjectConstructor
 ObjectPersistence 364

GetPalette
 PaletteData 369

GetUserCallback
 ObjectPersistence 364

GetUserCallbackName
 ObjectPersistence 364

GetUserObject
 ObjectPersistence 365

GetUserObjectName
 ObjectPersistence 365

Grid
 Table 513

Group 232
 Property Matrix B-781

grouping
 LocaleStruct 306

H

HeaderBackgroundColor
 Table 513

HeaderHeight
 Table 514

HeaderTextColor
 Table 514

HeaderType
 TableHeader 520

-
- HeaderWidth
 - Table 514
 - Height
 - BitmapData 52
 - IconData 264
 - MouseData 335
 - RegionStruct 431
 - height
 - ImageStruct 274
 - HelpContext
 - EventStruct 196
 - WindowObject 651
 - HelpObject
 - HelpTips 244
 - helpObject
 - WindowManager 607
 - HelpObjectTip
 - WindowObject 652
 - HelpStub 237
 - HelpSystem 238
 - HelpTips 242
 - HelpTipsType
 - HelpTips 245
 - hInstance
 - ZafMSWindowsApp .. 342
 - HorizontalScrollBar
 - Window 594
 - HotKeyChar
 - Button 66
 - Group 235
 - MessageData 317
 - Prompt 409
 - hotKeyChar
 - MessageStruct 319
 - HotKeyIndex
 - Button 66
 - Group 235
 - MessageData 317
 - Prompt 409
 - hotKeyIndex
 - MessageStruct 319
 - HotKeyText
 - ZafMSWindowsApp .. 342
 - HotSpotX
 - MouseData 335
 - hotSpotX
 - MouseStruct 338
 - HotSpotY
 - MouseData 335
 - hotSpotY
 - MouseStruct 338
 - Hour
 - UTimeData 574
 - HzJustify
 - Button 66
 - Prompt 410
 - String 483
 - Text 529
 - HzList 247
 - Property MatrixB-781
 - HzScrollPos
 - ScrolledWindow 454
 - HzShift
 - PositionStruct 389
 - RegionStruct 431
 - I**
 - I18nAllocate
 - I18nData 255
 - I18nData 252
 - I18nFree
 - I18nData 256
 - I18nName
 - I18nData 255
 - Icon 257
 - Display 160
 - Property MatrixB-781
 - IconData 262
 - Icon 259
 - IconImage
 - MessageWindow 323
 - IconStruct 265
 - IconType
 - Icon 260
 - Image 267
 - Property MatrixB-782
 - ImageData 272
 - ImageStruct 274
 - ImageType
 - Cursor 101
 - Mouse 330
 - Import
 - PositionStruct 389
 - ImportPixel
 - RegionStruct 431
 - ImportPoint
 - RegionStruct 431
 - Increment
 - ProgressBar 403
 - ScrollData 448
 - String 483
 - Index
 - List 299
 - PopUpItem 382
 - PullDownItem 415
 - SystemButton 505
 - IndexMethod

Chart 78

InitialDelay
 HelpTips 245
 WindowObject 653

InitializeOSBitmap
 Display 160

InitializeOSIcon
 Display 160

InitializeOSMouse
 Display 160

InitializeWrappers
 ZafMSWindowsApp .. 343

InputFormatData
 String 483

InputType
 EventStruct 196

Insert
 StringData 494

InsertRecord
 Table 514

Installed
 Device 131

intCurrencySymbol
 LocaleStruct 306

Integer 275
 Property Matrix B-782

IntegerData 280
 Integer 278

integerStringInputFormat
 LocaleStruct 306

integerStringOutputFormat
 LocaleStruct 306

internalHandle
 FileInfoStruct 211

Interval

Timer541

intFractionDigits
 LocaleStruct307

Invalid
 String484
 Text529

IsA
 Element174
 WindowObject654

IsoToUnicode
 CodeSetData87

ItemType
 PopUpItem382

IValue
 Bignum35
 BignumData40

J

JobSetup
 Printer395

JulianDay
 UTimeData574

JumpProc
 ZafMSWindowsApp ..343

K

Key
 EventStruct196

Keyboard285

KeyStruct288

L

Language
 LanguageManager 294

LanguageAllocate
 LanguageManager 295

LanguageData289, 317

LanguageFree
 LanguageManager 295

LanguageManager 293

LanguageName
 LanguageData 291
 LanguageManager 295

LanguageStruct 319

Last
 List 299
 PopUpItem 383
 PullDownItem 415
 SystemButton 505

LastFileSystem
 DataPersistence 116

LastPage
 Notebook 348

LeapYear
 UTimeData 574

left
 RegionStruct 431

Length
 DiskFile 140
 File 202
 StorageFile 476
 StringData 494

length
 FileInfoStruct 211

Line
 Display 161

-
- line
 - PositionStruct 388
 - lines
 - Display 154
 - LineStyle
 - Display 161
 - lineStyle
 - PaletteStruct 372
 - lineTable
 - ScreenDisplay 441
 - LinkDraggable
 - WindowObject 634
 - LinkMain
 - Application 25
 - List 296
 - list
 - ComboBox 92
 - ListBlock 300
 - ListIndex
 - Element 175
 - Live
 - Splitter 463
 - Load
 - Path 375
 - LocaleAllocate
 - LocaleData 303
 - LocaleData 301
 - LocaleFree
 - LocaleData 303
 - LocaleName
 - LocaleData 303
 - LocaleStruct 304
 - Lock
 - LanguageData 291
 - Locked
 - LanguageData 291
 - Window 594
 - LogicalEvent
 - WindowObject 655
 - LogicalPalette
 - WindowObject 659
 - logicalValue
 - EventMap 191
 - long
 - BignumData 40
 - IntegerData 282
 - UTimeData 574, 575
 - LowerCase
 - String 485
- ## M
- Main
 - Application 25
 - MakeFullPath
 - DiskFileSystem 144
 - MapTable
 - PaletteData 369
 - Margins
 - Printer 396
 - MarksXAxis
 - Chart 79
 - MarksYAxis
 - Chart 79
 - Maximize Button
 - Property Matrix B-782
 - MaximizeButton 310
 - Window 594
 - Maximized
 - Window 595
 - Maximum
 - DimensionConstraint .. 138
 - ProgressBar 403
 - ScrollData 448
 - maximum
 - ScrollStruct 450
 - MaxLength
 - StringData 494
 - MaxOffset
 - Table 514
 - mblen
 - CodeSetData 87
 - mbstowcs
 - CodeSetData 87
 - MDIChildJumpProc
 - ZafMSWindowsApp .. 343
 - MDIFrameJumpProc
 - ZafMSWindowsApp .. 344
 - MDIType
 - MDIWindow 316
 - MDIWindow 313
 - Property Matrix B-783
 - MemberUserFunction
 - WindowObject 663
 - memberUserFunction
 - WindowObject 663
 - menu
 - PopUpItem 384
 - PullDownItem 415
 - SystemButton 505
 - Merge
 - DataPersistence 116
 - ObjectPersistence 365
 - Message
 - MessageWindow 324

MessageFlags	Window596	mouseObject
MessageWindow 324	Mode	WindowManager 607
Messages	Display161	MouseStruct 337
LanguageData 292	modeTable	mouseTimerID
MessageWindow 320	ScreenDisplay441	ZafMSWindowsApp .. 344
MilliSecond	modifiers	Moveable
UTimeData 574	EventMap192	Window597
miniDenominatorX	EventStruct197	MoveDraggable
Display 153	monDecimalSeparator	WindowObject 634
miniDenominatorY	LocaleStruct307	MoveEvent
Display 153	monGrouping	WindowObject 665
Minimize Button	LocaleStruct307	
Property Matrix B-783	MonoBackground	
MinimizeButton 326	Display162	N
Window 595	monoBackground	name
Minimized	PaletteStruct373	FileInfoStruct 212
Window 596	MonoForeground	native3D
MinimizeIcon	Display162	ZafMSWindowsApp .. 344
Window 596	monoForeground	NativeMapping
Minimum	PaletteStruct373	CodeSetData 88
DimensionConstraint . 138	monoTable	negativeSign
ProgressBar 403	ScreenDisplay441	LocaleStruct 307
ScrollData 448	Month	negCurrencyPrecedes
minimum	UTimeData575	LocaleStruct 307
ScrollStruct 450	MonthName	negSignPrecedes
miniNumeratorX	UTimeData575	LocaleStruct 307
Display 153	monThousandsSeparator	negSpaceSeparation
miniNumeratorY	LocaleStruct307	LocaleStruct 308
Display 153	Motif class names	NewHelpTipDelay
Minute	X ResourcesG-811	HelpTips 245
UTimeData 574	Mouse329	Next
MkDir	Display162	Device 131
DiskFileSystem 144	MouseData333	Element 176
FileSystem 216	MouseEventRoute	WindowObject 665
Storage 473	WindowManager605	NextPaneObject
Modal		

- Splitter 463
 - Noncurrent
 - WindowObject 666
 - NormalBitmap
 - TreeItem 555
 - NormalHotKeys
 - Window 597
 - Notebook 346
 - Property Matrix B-784
 - Notification 350
 - NotifyCount
 - Notification 354
 - Timer 541
 - NotifyFocus
 - WindowObject 667
 - NotifyMessage
 - Timer 542
 - NotifySelection
 - WindowObject 670
 - NumberID
 - Element 177
 - numberID
 - FileInfoStruct 212
 - MessageStruct 319
- ## O
- Object
 - Constraint 96
 - ObjectFromHandle
 - ZafMSWindowsApp .. 344
 - objectHighWord
 - ZafMSWindowsApp .. 344
 - objectLowWord
 - ZafMSWindowsApp .. 344
 - ObjectPersistence 358
 - ObjNumberID
 - Constraint 96
 - Offset
 - Attachment 29
 - TableRecord 524
 - offset
 - EraStruct 180
 - oldFocusObject
 - WindowManager 608
 - Open
 - DiskFileSystem 144
 - FileSystem 216
 - Storage 474
 - OpenCreate
 - File 203
 - Storage 474
 - operator 205, 578, 578
 - List 299
 - operator -
 - BignumData 42
 - List 299
 - PopUpItem 384
 - PullDownItem 413
 - StringData 497
 - SystemButton 505
 - Timer 542
 - UTimeData 576
 - Window 599, 602
 - operator --
 - BignumData 42
 - IntegerData 283
 - RealData 429
 - RegionStruct 433
 - UTimeData 576
 - operator !=
 - BignumData 45
 - PositionStruct 389
 - RegionStruct 433
 - ScrollStruct 451
 - StringData 497
 - UTimeData 578
 - operator %
 - BignumData 43
 - operator %=
 - BignumData 45
 - IntegerData 284
 - RealData 429
 - operator ()
 - Window 602
 - operator *
 - BignumData 43
 - PaletteData 369
 - operator *=
 - BignumData 44
 - IntegerData 283
 - RealData 429
 - operator +
 - BignumData 42
 - List 297
 - PopUpItem 380
 - PullDownItem 413
 - StringData 497
 - SystemButton 504
 - Timer 541
 - UTimeData 576
 - Window 586, 602
 - operator ++
 - BignumData 43
 - IntegerData 283
 - RealData 429
 - RegionStruct 433
 - UTimeData 576
 - operator +=
 - IntegerData 283
 - RealData 429
 - RegionStruct 433
 - StringData 498
 - UTimeData 577
 - operator /
 - BignumData 43

operator /=	
BignumData	45
IntegerData	283
RealData	429
operator <	
BignumData	45
StringData	498
operator <=	
BignumData	46
StringData	498
operator -=	
BignumData	44
IntegerData	283
RealData	429
RegionStruct	433
StringData	497
UTimeData	577
operator =	
BitmapData	52
IconData	264
IntegerData	283
MouseDown	336
RealData	429
ScrollData	449
StringData	498
UTimeData	576
operator ==	
BignumData	45
PositionStruct	389
RegionStruct	432
ScrollStruct	451
StringData	499
UTimeData	578
operator >	
BignumData	46
StringData	499
UTimeData	578
operator >=	
BignumData	46
StringData	499
UTimeData	578
operator >>	

File	203
operator []	
StringData	499
OppositeSide	
Attachment	29
RelativeConstraint	437
Origin	
Display	163
OSDraw	
WindowObject	672
osEvent	
EventStruct	197
OSI18nName	
I18nData	255
OSLanguageToZafLanguage	
I18nData	254
osPalette	
PaletteStruct	374
OSScreenID	
WindowObject	673
OutputFormatData	
String	485
OutputFormatText	
String	485
Overlap	
RegionStruct	432
Owner	
Window	597
P	
PageHeight	
Printer	396
PageWidth	
Printer	396

Palette	
Display	163
palette	
PaletteMap	370
PaletteData	367
PaletteMap	370
PaletteState	
WindowObject	674
PaletteStruct	371
PaperHeight	
Printer	396
PaperOrientation	
Printer	397
PaperWidth	
Printer	396
Parent	
WindowObject	675
parentDirectoryName	
FileSystem	216
ParentDrawBorder	
WindowObject	676
ParentDrawFocus	
WindowObject	676
Password	
String	486
Path	375
PathElement	376
PathElement	376
PathID	
Image	270
PathName	
Image	270
patternTable	
ScreenDisplay	441

- Pixel
 - Display 163
- pixelsPerInchX
 - Display 153
- pixelsPerInchY
 - Display 153
- Poll
 - Cursor 101
 - Device 132
 - HelpTips 246
 - Keyboard 286
 - Mouse 332
 - Timer 542
- PollDevices
 - EventManager 188
- Polygon
 - Display 164
- PopLanguage
 - DataPersistence 116
- PopLevel
 - DataPersistence 117
- PopUpItem 378
 - Property Matrix B-784
- PopUpMenu 385
- posCurrencyPrecedes
 - LocaleStruct 308
- Position
 - EventStruct 197
 - PositionStruct 389
 - Splitter 464
- PositionStruct 388
- positiveSign
 - LocaleStruct 308
- posSignPrecedes
 - LocaleStruct 308
- posSpaceSeparation
 - LocaleStruct 308
- postSpace
 - Display 165
- preSpace
 - Display 165
- Previous
 - Device 131
 - Element 176
 - WindowObject 677
- PreviousPaneObject
 - Splitter 463
- PrintDialog 390
- Printer 392
 - PrinterType
 - Printer 397
- PrintJobStruct 399
- PrintSetup
 - Printer 397
- ProgressBar 400
 - Property Matrix B-784
- ProgressData
 - ProgressBar 403
- ProgressStyle
 - ProgressBar 404
- ProgressType
 - ProgressBar 404
- Prompt 406
 - Property Matrix B-784
- Properties B-773
- Property Matrices B-773
 - Bignum B-779
 - Bitmap B-779
 - Border B-779
 - Button B-779
 - ComboBox B-780
 - Date B-780
- Formatted String B-780
- Group B-781
- HzList B-781
- Icon B-781, B-782
- Integer B-782
- Maximize Button ... B-782
- MDIWindow B-783
- Minimize Button ... B-783
- Notebook B-784
- PopUpItem B-784
- ProgressBar B-784
- Prompt B-784
- PullDownItem B-785
- Real B-785
- ScrollBar B-786
- Scrolled Window ... B-786
- SpinControl B-786
- Splitter B-786
- String B-786
- SystemButton B-787
- Table B-787
- TableHeader B-788
- TableRecord B-788
- Text B-788
- Time B-788
- Title B-789
- ToolBar B-789
- TreeItem B-789
- TreeList B-790
- UTime B-790
- VtList B-790
- Window B-776
- WindowObject B-775
- PullDownItem 411
 - Property Matrix B-785
- PullDownMenu 417
 - Window 597
- PushLanguage
 - DataPersistence 116
- PushLevel
 - DataPersistence 117
- Put
 - EventManager 188

Q

QueueBlock 420
QueueElement 421
QueueEvents
 Timer 542
QuickTip
 WindowObject 677

R

RangeData
 String 486
RangeText
 String 486
Ratio
 RelativeConstraint 437
rawCode
 EventMap 191
 EventStruct 197
Read
 File 203
ReadData
 DiskFile 141
 File 204
 StorageFile 477
ReadFromBeginning
 EventManager 188
ReadFromEnd
 EventManager 189
ReadOnly
 File 204
 FileInfoStruct 212
 Storage 474
ReadRecord

 Table 515
ReadWrite
 File 204
 Storage 474
Real 422
 Property Matrix B-785
RealData 426
 Real 425
realStringInputFormat
 LocaleStruct 308
realStringOutputFormat
 LocaleStruct 308
RearrangeArgs
 StandardArg 465
Record
 Table 515
Rectangle
 Display 165
RectangleXORDiff
 Display 165
Redisplay
 WindowObject 679
RedisplayData
 WindowObject 679
Reference
 Attachment 30
ReferenceNumberID
 Attachment 30
Region
 Button 67
 EventStruct 197
 RegionStruct 432
 WindowObject 679
RegionCopy
 Display 166
RegionStruct 430

RegionType
 WindowObject 680
RelativeConstraint 434
Remove
 DiskFileSystem 144
 FileSystem 217
 Storage 474
 StringData 494
Rename
 DiskFileSystem 145
 FileSystem 217
 Storage 474
Rep 730
Rep Utility
 ZAF 4 730
Rep4to5 731
Rep4to5 Utility
 ZAF 4 731
RepeatDelay
 WindowObject 653
Repopulate
 Table 515
ReportError
 ErrorStub 183
ReportInvalid
 String 487
ResetI18n
 I18nData 256
ResetOSBitmap
 Display 167
ResetOSIcon
 Display 167
ResetOSMouse
 Display 167
ResetStorage
 HelpSystem 241

-
- RestoreDisplayContext
 - Display 157
 - RestoreDrawContext
 - Display 157
 - right
 - RegionStruct 431
 - RmDir
 - DiskFileSystem 145
 - FileSystem 217
 - Storage 475
 - rootDirectoryName
 - FileSystem 217
 - RootObject
 - WindowObject 683
 - route
 - EventStruct 198
 - RowHeight
 - Table 516
 - RValue
 - Bignum 35
 - BignumData 41
- S**
- Save
 - Storage 475
 - SaveAs
 - Storage 475
 - Scale
 - Display 167
 - Scaled
 - Image 270
 - ScreenDisplay 439
 - ScreenID
 - EventStruct 198
 - screenID
 - WindowObject 683
 - Scroll
 - EventStruct 198
 - ScrollBar 442
 - Property Matrix B-786
 - ScrollData 446
 - ScrollBar 445
 - Scrolled Window
 - Property Matrix B-786
 - ScrolledWindow 452
 - ScrollEvent
 - WindowObject 685
 - ScrollHeight
 - ScrolledWindow 454
 - ScrollStruct 450
 - ScrollType
 - ScrollBar 445
 - ScrollWidth
 - ScrolledWindow 454
 - Second
 - UTimeData 575
 - Seek
 - DiskFile 141
 - File 204
 - StorageFile 477
 - SeekNextRecord
 - Table 516
 - SeekPreviousRecord
 - Table 516
 - SeekRandomRecord
 - Table 516
 - Selected
 - WindowObject 686
 - SelectedBitmap
 - SelectedItem 555
 - SelectionType
 - HzList 250
 - TreeList 562
 - VtList 582
 - Window 598
 - SelectOnDoubleClick
 - Button 68
 - SelectOnDownClick
 - Button 68
 - SendMessage
 - Button 68
 - SendMessageText
 - Button 69
 - SendMessageWhenSelected
 - Button 68
 - SetAcceptDrop
 - WindowObject 614
 - SetAllowDefault
 - Button 61
 - SetAllowInvalid
 - String 481
 - SetAllowModifyCollate
 - PrintDialog 391
 - SetAllowModifyCopies
 - PrintDialog 391
 - SetAllowModifyRange
 - PrintDialog 391
 - SetAllowToggling
 - Button 62
 - SetAutoClear
 - String 481
 - Text 528
 - SetAutomaticUpdate
 - Window 588
 - WindowObject 617

SetAutoRepeatSelection Button 62	Cursor100	FormattedString 223
SetAutoSelect Group 234	SetBordered WindowObject621	SetCoordinateType Display 156 Table512 WindowObject 627
SetAutoSize Bitmap 49 Button 63 Image 269 Prompt 408 ScrollBar 445	SetBottomOffset Table512	SetCopyDraggable WindowObject 634
SetAutoSortData ComboBox 92 HzList 249 TreeItem 553 TreeList 561 VtList 581	SetButtonType Button64	SetCurrent List 298 ProgressBar 402 ScrollData 448
SetAxisColor Chart 76	SetCellHeight HzList250	SetCurrentOffset Table 512
SetBackground Display 152	SetCellSize Display153	SetCurrentPage Notebook 348
SetBackgroundColor ComboBox 92 HzList 250 TreeList 563 VtList 581 WindowObject 619	SetCellWidth HzList250	SetCursorOffset String 481 Text 528
SetBasisYear UTimeData 573	SetCenter RelativeConstraint436	SetCursorPosition Text 528
SetBignum Bignum 36 BignumData 41	SetChanged Window590 WindowObject622	SetDataConstructors DataPersistence 118
SetBignumData Bignum 34	SetChar StringData492	SetDataGroupBackground- Color Chart 77
SetBitmap BitmapData 52	SetChartType Chart76	SetDataGroupFillPattern Chart 77
SetBitmapData Bitmap 49 Button 63	SetClipRegion Display153	SetDataGroupFont Chart 78
SetBlinkRate	SetCodeSetName CodeSetData84	SetDataGroupForegroundCol- or Chart 78
	SetColumnText Table512	SetDataGroupLineStyle Chart 78
	SetCompareFunction List297	SetDate DateData 128
	SetCompareFunctions ObjectPersistence365	
	SetCompressedData FormattedString223	
	SetCompressedText	

-
- SetDateData
 - Date 124
 - SetDefaultButton
 - Window 592
 - SetDefaultEventRoute
 - WindowManager 605
 - SetDefaultLeadByte
 - CodeSetData 87
 - SetDefaultMessageFlag
 - MessageWindow 323
 - SetDeleteData
 - FormattedString 224
 - SetDeleteText
 - FormattedString 224
 - SetDelta
 - ProgressBar 402
 - ScrollData 448
 - SpinControl 458
 - SetDepressed
 - Button 65
 - SetDepth
 - Button 65
 - SetDestroyable
 - Data 105
 - Window 593
 - SetDevice
 - EventStruct 195
 - SetDeviceImage
 - EventManager 189
 - SetDevicePosition
 - EventManager 189
 - SetDeviceState
 - Device 130
 - EventManager 189
 - SetDirectory
 - FileDialog 209
 - SetDisabled
 - WindowObject 631
 - SetDisplay
 - EventStruct 196
 - SetDisplayContext
 - Display 157
 - SetDockType
 - ToolBar 548
 - SetDrawContext
 - Display 157
 - SetDrawLines
 - TreeList 562
 - SetDrive
 - DiskFileSystem 145
 - SetEditMode
 - WindowObject 639
 - SetError
 - Application 25
 - Data 105
 - DataPersistence 115
 - File 202
 - FileSystem 214
 - WindowObject 639
 - SetEventManager
 - EventStruct 196
 - SetExitFunction
 - WindowManager 607
 - SetExpandable
 - TreeItem 554
 - SetExpanded
 - TreeItem 554
 - SetExpandedData
 - FormattedString 224
 - SetExpandedText
 - FormattedString 225
 - SetFile
 - FileDialog 209
 - SetFillPattern
 - Display 159
 - SetFilter
 - FileDialog 209
 - SetFocus
 - WindowObject 647
 - SetFocusOffset
 - Table 513
 - SetFont
 - ComboBox 92
 - Display 159
 - HzList 251
 - VtList 582
 - WindowObject 649
 - SetForeground
 - Display 160
 - SetFormatData
 - FormattedString 225
 - SetFormatText
 - FormattedString 225
 - SetGrid
 - Table 513
 - SetHeaderBackgroundColor
 - Table 513
 - SetHeaderHeight
 - Table 514
 - SetHeaderTextColor
 - Table 514
 - SetHeaderType
 - TableHeader 520
 - SetHeaderWidth
 - Table 514
 - SetHelpContext
 - EventStruct 196
 - WindowObject 651

SetHelpObject	HelpTips245	LocaleData303
HelpTips244	WindowObject653	
SetHelpObjectTip	SetInputFormat	SetLocked
WindowObject652	String483	Window594
SetHelpTipsType	SetInputFormatData	SetLowerCase
HelpTips245	String483	String485
SetHotKey	SetInteger	SetMapTable
Button66	Integer279	PaletteData369
Group235	IntegerData282	SetMargins
MessageData317	SetIntegerData	Printer396
Prompt409	Integer278	SetMarksXAxis
SetHotSpot	SetInterval	Chart79
MouseData335	Timer541	SetMarksYAxis
SetHzJustify	SetInvalid	Chart79
Button67	String484	SetMaximized
Prompt410	Text529	Window595
String483	SetItemType	SetMaximum
Text529	PopUpItem382	DimensionConstraint ..138
SetHzScrollPos	SetJulianDay	ProgressBar403
ScrolledWindow454	UTimeData574	ScrollData448
SetI18nName	SetKey	SetMaxLength
I18nData255	EventStruct196	StringData494
SetIcon	SetLanguage	SetMaxOffset
IconData264	LanguageManager294	Table514
SetIconData	SetLanguageName	SetMDIType
Icon260	LanguageData291	MDIWindow316
SetIconImage	LanguageManager295	SetMessage
Icon260	SetLineStyle	LanguageData292
MessageWindow323	Display161	MessageWindow324
SetIconType	SetLinkDraggable	SetMessageFlags
Icon260	WindowObject634	MessageWindow324
SetImageType	SetLive	SetMessages
Cursor101	Splitter463	LanguageData292
Mouse330	SetLocale	SetMinimized
SetIndexMethod	LocaleData302	Window596
Chart79	SetLocaleName	SetMinimum
SetInitialDelay		DimensionConstraint ..138

-
- ProgressBar 403
 - ScrollData 448
 - SetModal
 - Window 596
 - SetMode
 - Display 161
 - SetMonoBackground
 - Display 162
 - SetMonoForeground
 - Display 162
 - SetMouse
 - MouseData 335
 - SetMouseEventRoute
 - WindowManager 605
 - SetMoveable
 - Window 597
 - SetMoveDraggable
 - WindowObject 634
 - SetNativeMapping
 - CodeSetData 88
 - SetNewHelpTipDelay
 - HelpTips 245
 - SetNextPaneObject
 - Splitter 463
 - SetNoncurrent
 - WindowObject 666
 - SetNormalBitmap
 - TreeItem 555
 - SetNormalHotKeys
 - Window 597
 - SetNotifyMessage
 - Timer 542
 - SetNumberID
 - Element 177
 - SetObject
 - Constraint 96
 - SetObjectConstructors
 - ObjectPersistence 365
 - SetObjNumberID
 - Constraint 96
 - SetOffset
 - Attachment 29
 - TableRecord 524
 - SetOppositeSide
 - Attachment 29
 - RelativeConstraint 437
 - SetOrigin
 - Display 163
 - SetOSDraw
 - WindowObject 672
 - SetOSText
 - StringData 495
 - SetOSWText
 - StringData 495
 - SetOutputFormat
 - String 485
 - SetOutputFormatData
 - String 485
 - SetOwner
 - Window 597
 - SetPageHeight
 - Printer 396
 - SetPageWidth
 - Printer 396
 - SetPalette
 - Display 163
 - SetPaperHeight
 - Printer 396
 - SetPaperOrientation
 - Printer 397
 - SetPaperWidth
 - Printer 396
 - SetParent
 - WindowObject 675
 - SetParentDrawBorder
 - WindowObject 676
 - SetParentDrawFocus
 - WindowObject 676
 - SetParentPalette
 - WindowObject 676
 - SetPassword
 - String 486
 - SetPath
 - PathElement 376
 - SetPathID
 - Image 270
 - SetPathName
 - Image 270
 - SetPosition
 - EventStruct 197
 - Splitter 464
 - SetPreviousPaneObject
 - Splitter 463
 - SetPrinterType
 - Printer 397
 - SetProgressData
 - ProgressBar 403
 - SetProgressStyle
 - ProgressBar 404
 - SetProgressType
 - ProgressBar 404
 - SetQueueEvents
 - Timer 542
 - SetQuickTip
 - WindowObject 677

SetRange String 486	EventStruct198 ScrollData449	SetSprintf StandardArg 466
SetRangeData String 486	SetScrollData ScrollBar445	SetSscanf StandardArg 466
SetRatio RelativeConstraint 437	SetScrollHeight ScrolledWindow454	SetStaticArray ImageStruct 274
SetReadFunction Table 515	SetScrollType ScrollBar445	SetStaticData LanguageData 292 PaletteData 369 StringData 496
SetReal Real 425 RealData 428	SetScrollWidth ScrolledWindow454	SetStaticHandle BitmapStruct 54 IconStruct 266 MouseStruct 339
SetRealData Real 425	SetSelected String487 WindowObject686	SetStep ProgressBar 404 ScrollData 449
SetReference Attachment 30	SetSelectedBitmap TreeItem555	SetStretch Attachment30 RelativeConstraint437
SetReferenceNumberID Attachment 30	SetSelectionType HzList250 TreeItem555 TreeList562 VtList582 Window598	SetStringData Button 70 Group 235 Prompt 410 String 487 Text 529 TreeItem 556
SetRegion Button 67 EventStruct 197 WindowObject 679	SetSelectOnDoubleClick Button68	SetStringID Element 178
SetRegionType WindowObject 680	SetSelectOnDownClick Button68	SetSupportObject WindowObject687
SetRepeatDelay WindowObject 653	SetSendMessageText Button69	SetSystemButtonType SystemButton505
SetReportInvalid String 487	SetSendMessageWhenSelect- ed Button68	SetSystemObject WindowObject687
SetRowHeight Table 516	SetShowing ScrollData449	SetTabHeight Notebook 349
SetScale Display 167	SetSizeable Window598	SetTabHotKey
SetScaled Image 270	SetSplitterType Splitter464	
SetScreenID EventStruct 198		
SetScroll		

-
- Notebook 349
 - SetTabText
 - Notebook 349
 - SetTabWidth
 - Notebook 349
 - SetTemporary
 - Window 601
 - SetText
 - Button 70
 - String 487
 - StringData 496
 - Text 530
 - TreeItem 556
 - Window 601
 - WindowObject 687
 - SetTextColor
 - ComboBox 92
 - HzList 251
 - TreeList 563
 - VtList 582
 - WindowObject 689
 - SetTextStyle
 - ProgressBar 405
 - SetTitle
 - Chart 79
 - SetTextXAxis
 - Chart 79
 - SetTextYAxis
 - Chart 80
 - SetThickness
 - Splitter 464
 - SetTiled
 - Image 270
 - SetTime
 - TimeData 538
 - SetTimeData
 - Time 535
 - SetTopOffset
 - Table 517
 - SetTransparentBackground
 - Prompt 410
 - SetType
 - Attachment 30
 - DimensionConstraint 139
 - RelativeConstraint 437
 - SetUnanswered
 - String 488
 - Text 530
 - SetUpdate
 - Notification 354
 - SetUpperCase
 - String 488
 - SetUserCallbacks
 - ObjectPersistence 365
 - SetUserFunction
 - WindowObject 691
 - SetUserObjects
 - ObjectPersistence 366
 - SetUserPalette
 - HelpTips 246
 - SetUserPaletteData
 - WindowObject 693
 - SetUTime
 - UTimeData 575
 - SetUTimeData
 - UTime 569
 - SetValue
 - Button 71
 - SetVariableName
 - String 489
 - SetVersion
 - Storage 475
 - SetViewCurrent
 - TreeList 563
 - SetViewOnly
 - ComboBox 93
 - String 489
 - Text 531
 - SetVirtualField
 - TableHeader 521
 - SetVirtualRecord
 - Table 517
 - SetVisible
 - WindowObject 699
 - SetVoidData
 - EventStruct 198
 - SetVtJustify
 - Button 67
 - Prompt 410
 - SetVtScrollPos
 - ScrolledWindow 455
 - SetWallpaper
 - Image 271
 - SetWidth
 - Border 57
 - SetWindow
 - EventStruct 199
 - SetWindowManager
 - EventStruct 199
 - SetWindowObject
 - EventStruct 199
 - SetWordWrap
 - Text 531
 - SetWrapChildren
 - ToolBar 548
 - SetWrappedData
 - SpinControl 459
 - SetWriteFunction
 - Table 517

SetZoneOffset	StaticHandle	PopUpItem	384
UTimeData	BitmapStruct	PullDownItem	416
576	IconStruct	SystemButton	505
ShiftPressed	MouseStruct	Timer	542
Keyboard	339	Window	599
287	StatusBar		
shiftState	468	SubtractCompareFunction	
KeyStruct	Step	ObjectPersistence	366
288	ProgressBar		
Showing	404	SubtractDataConstructor	
ScrollData	ScrollData	DataPersistence	118
449	449		
showing	Storage	SubtractFileSystem	
ScrollStruct	471	DataPersistence	118
450	StorageFile		
Sizeable	476	SubtractNotification	
Window	streq	Notification	352
598	717		
skipGregorian	Stretch	SubtractObjectConstructor	
LocaleStruct	Attachment	ObjectPersistence	366
308	30		
Sort	RelativeConstraint	SubtractStaticModule	
List	437	I18nData	254
299	String		
PopUpItem	Property Matrix	SubtractUserCallback	
384	B-786	ObjectPersistence	366
PullDownItem	StringData		
416	Button	SubtractUserObject	
SystemButton	Group	ObjectPersistence	366
505	235		
SpinControl	Prompt	support	
Property Matrix	String	Window	599
B-786	Text		
Splitter	TreeItem	SupportCurrent	
Property Matrix	555	Window	599
B-786	StringID		
SplitterType	Element	SupportDestroy	
Splitter	178	Window	599
464	stringID		
StandardArg	FileInfoStruct	SupportFirst	
465	MessageStruct	Window	599
startDate	319		
EraStruct	StripFullPath	SupportLast	
181	DiskFileSystem	Window	599
state	145		
PaletteMap	strneq	SupportObject	
370	720	WindowObject	687
StaticArray	strnicmp		
ImageStruct	721	SystemButton	501
274	strepc	Property Matrix	B-787
StaticData	722	Window	600
LanguageData	rstrip		
292	723		
PaletteData	Subtract		
369	List		
StringData	299		
496			

SystemButtonMenu
 Window 600

SystemButtonType
 SystemButton 505

SystemObject
 WindowObject 687

T

TabHeight
 Notebook 349

TabHotKeyChar
 Notebook 349

TabHotKeyIndex
 Notebook 349

Table 507
 Property Matrix B-787
 TableHeader 521
 TableRecord 524

TableHeader 518
 Property Matrix B-788

TableRecord 522
 Property Matrix B-788

TabText
 Notebook 349

TabWidth
 Notebook 349

Tell
 DiskFile 141
 File 205
 StorageFile 477
 Table 512

TempName
 DiskFileSystem 145

Temporary
 File 205

Window 601

Text 525
 Button 70
 Display 168
 Property Matrix B-788
 String 487
 StringData 496
 Text 530
 TreeItem 556
 Window 601
 WindowObject 687

text
 EventStruct 198
 MessageStruct 319

TextBlock
 Printer 398

TextColor
 WindowObject 689

TextLine
 Printer 398

TextMode
 File 205

TextSize
 Display 169

TextStyle
 ProgressBar 405

TextTitle
 Chart 79

TextXAxis
 Chart 79

TextYAxis
 Chart 80

Thickness
 Splitter 464

thousandsSeparator
 LocaleStruct 309

Tiled

Image 270

Time 532
 Property Matrix B-788

time12StringOutputFormat
 LocaleStruct 309

TimeData 536
 Time 535

TimeName
 UTimeData 575

Timer 539

timeSeparator
 LocaleStruct 309

TimeStamp
 LocaleData 303

timeStringInputFormat
 LocaleStruct 309

timeStringOutputFormat
 LocaleStruct 309

Title 543
 Property Matrix B-789
 Window 601

ToggleExpanded
 TreeItem 554

ToggleSelected
 WindowObject 686

ToolBar 546
 Property Matrix B-789

top
 RegionStruct 431

topLevelShellClassRec
 X Resources G-811

TopOffset
 Table 517

Touching
 RegionStruct 432

TransparentBackground
 Prompt 410

TreeItem 550
 Property Matrix B-789

TreeList 558
 Property Matrix B-790
 TreeItem 556

Type
 Attachment 30
 DimensionConstraint . 139
 RelativeConstraint 437

type
 EventStruct 198
 PaletteMap 370

U

Unanswered
 String 488
 Text 530

UnicodeToIso
 CodeSetData 88

UnLock
 LanguageData 291

Update
 Notification 354

UpdateData
 Notification 356

UpdateObjects
 Notification 356

UpperCase
 String 488

userFlags
 WindowObject 690

UserFunction
 WindowObject 691

userFunction
 WindowObject 691

UserInformation
 WindowObject 692

userObject
 WindowObject 693

UserPaletteData
 WindowObject 693

userPaletteData
 WindowObject 693

userStatus
 WindowObject 698

UTime 565
 Property Matrix B-790

UTimeData 570
 UTime 569

V

Validate
 String 488

ValidName
 DiskFileSystem 146

Value
 Button 71
 Integer 279
 IntegerData 282
 Real 425
 RealData 428
 UTimeData 575

value
 KeyStruct 288

VariableName
 String 489

Version
 Storage 475

VerticalScrollBar
 Window 602

ViewCount
 TreeList 563

ViewCurrent
 TreeItem 556
 TreeList 563

ViewFirst
 TreeItem 556
 TreeList 564

ViewLast
 TreeItem 557
 TreeList 564

ViewLevel
 TreeItem 557

ViewNext
 TreeItem 557

ViewOnly
 ComboBox 93
 String 489
 Text 531

ViewPrevious
 TreeItem 557

VirtualField
 TableHeader 521

VirtualRecord
 Table 517

Visible
 WindowObject 699

VoidData
 EventStruct 198

vsprintf
 StandardArg 466

vsscanf
 StandardArg 467

VtJustify

Button 66
 Prompt 410
 VtList 579
 Property Matrix B-790
 VtScrollPos
 ScrolledWindow 455
 VtShift
 PositionStruct 389
 RegionStruct 432

W

Wallpaper
 Image 270
 wcstombs
 CodeSetData 88
 Width
 BitmapData 52
 Border 57
 IconData 264
 MouseData 336
 RegionStruct 432
 width
 ImageStruct 274
 WildStremp 726
 Window 583
 EventStruct 199
 Property Matrix B-776
 WindowManager 603
 EventStruct 199
 windowManager
 WindowObject 700
 WindowObject 609
 EventStruct 199
 Property Matrix B-775
 windowsPlatform

ZafMSWindowsApp ..345
 windowsVersion
 ZafMSWindowsApp ..345
 WordWrap
 Text531
 WrapChildren
 ToolBar548
 WrappedData
 SpinControl459
 Write
 File205
 WindowObject701
 WriteData
 DiskFile141
 File206
 StorageFile477
 WriteRecord
 Table517

X

X ResourcesG-811
 xmBulletinBoardClassRec
 X ResourcesG-811
 xmCascadeButtonClassRec
 X ResourcesG-812
 xmFileSelectionBoxWidget
 X ResourcesG-811
 xmLabelClassRec
 X ResourcesG-812
 xmPrimitiveClassRec
 X ResourcesG-811
 xmPushButtonClassRec
 X ResourcesG-811
 xmRowColumnClassRec

X ResourcesG-812
 xmRowColumnWidgetClass-
 Rec
 X ResourcesG-812
 xmScaleClassRec
 X ResourcesG-812
 xmScrollBarClassRec
 X ResourcesG-812
 xmTextFieldClassRec
 X ResourcesG-811

Y

Year
 UTimeData 576

Z

ZAF 5 to 4 Class Comparisons
 C-791
 ZAF 4C-791
 ZafAbs 705
 ZafCoordinate 147
 ZafCrNIToCr 706
 ZafCrNIToNI 707
 ZafCrToCrNI 708
 ZafIChar
 StringData 496
 ZafLanguageToZafLocale
 I18nData 255
 ZafLocaleToZafLanguage
 I18nData 254
 ZafMax 710

ZafMessageStruct	
LanguageData	292
ZafMin	711
ZafMSWindowsApp	340
ZafNIToCrNI	712
zafRegion	
WindowObject	679
ZafRegisterMouse	714
ZafStrColl	715
ZafStrdup	716
ZafStricmp	718
ZafStrlwr	719
ZafStrupr	724
ZafStrXFrm	725
ZMake	746
ZoneOffset	
UTimeData	576